APPLICATIONS OF SUPERCOMPUTERS IN CHEMICAL ENGINEERING

M.A. Stadtherr

Dept. Chem. Eng., 208 Adams Lab. Box C-3
University of Illinois
1209 West California Street, Urbana, Illinois 61801, USA

ABSTRACT

In the first part of this paper we briefly discuss the following: 1. The basic architectural concepts used in supercomputers and other advanced-architecture machines; 2. Applications of these machines with emphasis on those in chemical engineering; 3. Challenges in developing algorithms and codes that take best advantage of advanced computer architectures. In the second part of the paper we use a specific example, namely sparse matrix methods for equation-based process flowsheeting, to illustrate a few of the considerations that may arise in applying advanced computer architectures.

ADVANCED COMPUTER ARCHITECTURES

Advanced computer architectures provide the potential for much higher computational speeds than typical "conventional" computers, as well as access to larger central memories. For more detail there are two good introductions to some aspects of advanced computer architectures by Levine (1982) and Lerner (1985), as well as a recent overview by Vegeais et al. (1986). A review of various algorithms that exploit these architectures is given in Ortega and Voigt (1985). Also, tutorial volumes compiled by Hwang (1984) and by Kuhn and Padua (1981) comprise a variety of papers describing both advanced architectures and related algorithms.

Today, the best known and most widely used advanced-architecture machines are the so-called supercomputers, as typified by machines such as the Cray-1, the CDC Cyber 205, the Cray X-MP, and the Cray-2. To relate supercomputer speed to that of some familiar conventional machines, we note that the Cray-1's peak speed is more than an order of magnitude faster than that of large mainframe computers like the CDC Cyber 175 or the IBM 3081, roughly three orders of magnitude faster than the VAX 11/780, a popular superminicomputer, and roughly four orders of magnitude faster than the IBM PC/AT. Depending on the number of processors available, the Cray X-MP and Cray-2 can increase computational throughput by yet another order of magnitude or more relative to the Cray-1. It should be emphasized that these figures reflect approximate peak speeds, which may not be realized in practice unless the program and algorithm used are able to take advantage of the supercomputer architecture. One very popular benchmark for comparing computer performance is the solution of linear equation systems using the LINPACK routines (Dongarra, 1986).

While supercomputers have been defined as the fastest computers available at any specific time, or as computers that are only one generation behind the computing requirements of leading-edge efforts in science and engineering, the characteristic that distinguishes today's supercomputers from other computers is the extensive use of some form of parallelism. The need for parallelism can be easily seen by noting that since computers were first being manufactured, most of the increases in computational speed have been due simply to increased clock speed. However, it appears that a limit to the clock speed is now being approached. For computers with extremely fast clock speeds, the dimensions of the machine become an important consideration. Since electrical signals cannot travel faster than the speed of light, a very high clock speed requires that the maximum distance a signal must travel be very small. For example, a signal can travel only about 30 centimeters in one nanosecond. Therefore, a computer with a one ns clock (which would be roughly a factor of four faster than the clock cycle of a processor in a state-of-the-art supercomputer such as the Cray-2) must be no more than about a cubic foot in size. This, of course, causes tremendous wiring and cooling problems. Because of this, it has become necessary for computer manufacturers to look to the parallel processing of instructions and data to significantly increase the speed of their computers. This parallelism generally manifests itself in some form of vector processing or multiprocessing architecture, as described below. For instance, machines such as the Cray X-MP or Cray-2 use a vector multiprocessing architecture.

Despite all of the current interest in supercomputers, their availability is still limited by their high cost, typically in excess of $5 million and perhaps much more. This may put the supercomputer out of the price range of all but the largest concerns. However, machines have now become available which cost one or two orders of magnitude less than this, and which, by using advanced computer architectures, offer substantial performance improvements compared to similarly priced conventional machines. Some of these advanced-architecture machines use architectures very similar to the Cray-1, and may even be software compatible with the Cray-1. Others use much different architectural concepts. Assuming that their architectures can be effectively exploited by the user, all of these machines, often called "minisupercomputers" or even "personal supercomputers," appear to offer significantly better price/performance ratios than the conventional technology.

As noted above, the key element in advanced computer architectures is parallelism. There are many different ways in which a computer can be made to operate in parallel. In order to easily distinguish between these differences, several

different sets of categories have been proposed. One simple way to classify advanced architectures is as vector processors, multiprocessors, and vector multiprocessors (Bucher, 1983).

The vector processing category includes "SIMD" and "pipelined" computers because both facilitate the processing of identical operations on large vectors or arrays of numbers. Pipelining is perhaps best explained using an assembly-line analogy. For example, a floating-point operation involves several steps. Without pipelining, all the steps needed to complete one operation would be performed before starting the first step of the next operation. Thus, in effect, the computer works on one operation at a time. On the other hand, to perform a particular operation in a highly pipelined computer, there are several "stations," each of which performs only one step of the overall operation. Since, as in an assembly line, all stations are working concurrently, the computer can perform each step of the operation on different data at the same time. Examples of this type of computer are the Cyber 205 and the Cray-1; both are highly pipelined machines, although they operate somewhat differently. For instance, the Cyber 205 operates most efficiently on very long vectors, while the Cray-1 can operate efficiently even on relatively short vectors. SIMD computers are composed of an array of separate processors, each performing the same operation at the same time but on different data. Perhaps the best known high-performance machines of this type are the ILLIAC IV, now retired, and the Massively Parallel Processor (MPP) built for NASA by Goodyear Aerospace, primarily for image processing. The ILLIAC IV, which had an array of 64 processors, could, for instance, add two 64-element vectors in about the same time as a scalar addition, since all 64 scalar additions needed to add the vectors could be done in parallel.

The multiprocessing category covers architectures that use an array of scalar processors, each capable of executing different instructions at the same time on different data. Since the processors can execute different instructions, they are not necessarily synchronized. Two examples of commercial multiprocessors are the BBN Butterfly and the Intel iPSC. There are also many experimental prototype machines of this type, as well as other machines in various stages of commercialization.

The vector multiprocessing category is essentially a combination of vector processing and multiprocessing. There are a number of processors that can run in parallel, and each of these processors is itself a powerful vector processor, generally of the pipelined type. Most state-of-the-art supercomputers, such as the Cray X-MP and the Cray-2 fall into this category, and plans for newer machines suggest that this architecture will continue to dominate the high end of the supercomputer market for at least the short-to-medium term. Currently available state-of-the-art machines have up to four vector processors in parallel, and this number is expected to grow significantly over the next several years.

Applications

For the chemical engineer, most of the opportunities provided by advanced- architecture machines can be grouped into three overlapping categories: 1. The opportunity to solve problems involving the modeling and analysis of complex physical phenomena which were previously intractable, or at least computationally infeasible; 2. The opportunity to greatly increase engineering design productivity in areas requiring large-scale computation; 3. The opportunity to use complex models in real-time applications. We discuss each of these categories very briefly here. More detail on supercomputer applications in chemical engineering has been provided recently by Stadtherr and Vegeais (1985a).

There are a number of well-established applications in the first category, including petroleum reservoir simulation, weather forecasting, quantum chemistry, high energy physics, astrophysics, etc. Some typical features of problems in this category are: 1. They require computationally intensive solution methods such as finite element methods, finite difference methods, or Monte Carlo simulation; 2. They may be computationally intractable except on supercomputers (for example, problems that could take months to solve on a Vax may require only hours on a Cray); 3. Using the supercomputer, they can be solved with much higher resolution, in more dimensions, and with models that more accurately represent the true chemistry and physics. Some typical applications of this type in chemical engineering include turbulent flow phenomena, polymer rheology, combustion, chemical vapor deposition, etc. Chemical engineering problems in this area often involve rapid changes in a phenomena with respect to position or time, or an interacting combination of phenomena, e.g., simultaneous chemical reaction, fluid flow, heat transfer, or mass transfer. The supercomputer, while perhaps still not able to solve all problems to the resolution desired, provides the chemical engineer the opportunity to model such phenomena in much more detail and with greater accuracy than previously possible.

The second category is the use of advanced computer architectures to increase the productivity of design engineers. There are well established applications of this sort in nuclear engineering, automotive engineering, aerospace engineering, and electrical engineering. Problems in this category can often be solved using conventional computers, but may require several minutes, or much more, of computer time. By using the speed of advanced computer architectures, design productivity can be greatly increased (for example problems that could require several minutes on a Vax may require a second or less on a Cray). This means the engineer can consider many more design alternatives, and, since the engineer can get very rapid feedback on the results of design changes, there is much better man-machine interaction in the design process. In chemical engineering there have been relatively few applications of this sort so far. A few recently described ones are: batch distillation (Crico, 1986) and interconnected systems of distillation columns (Senior, 1986). Likely areas of major impact in the future include both steady- and unsteady-state process flowsheeting and optimization.

The third category is currently the least developed of the three. Real-time applications include robotics, avionics, speech recognition, image processing, and process control. Applications in this category generally require

either that data be sampled at an extremely high rate or that a complex model or computationally intensive procedure be used to process the data (so that without advanced architecture speed, the computer would fall behind real time). The most likely chemical engineering application is process control. Real time simulations of entire plant complexes using advanced computer architectures will provide a powerful tool for on-line, model-based optimization of complete plant operations.

Algorithms and Codes

As noted above, the potential speed of advanced computer architectures may not be (and usually is not) realized in practice unless the programs and algorithms used are able to take advantage of the architecture. A factor used to measure the performance of an algorithm or program in this regard is "speedup." On a multiprocessor, speedup is defined as the time it takes to complete a job using only one processor, divided by the time the job requires using P processors. Ideally speedup would be P, indicating that the algorithm could be performed in P independent parts of equal size. Often, the speedup is divided by the number of processors and is then called the efficiency.

It has been shown, based on a very simple model of parallel processing, that if the fraction of code that can be performed in parallel on P processors is f, then the maximum speedup S is $P/(P - fP + f)$, a relationship known as Amdahl's law (Amdahl, 1967). Speedup is a measure that can also be used with vector computers. In this case it is the ratio of the time it takes for a job to execute without vectorization to the time it takes for a job to execute with vectorization. For vector computers, one can obtain an equation identical to Amdahl's law. In this case, however, f refers to the fraction of the code that vectorizes, and P refers to ratio of peak vector speed to peak scalar speed. It can be seen from Amdahl's law that significant speedup requires that substantial portions of the code be run in parallel. Note, for instance, that with 64 processors, just 5% non-parallelized code will result in a maximum speedup of 15.42 (or a maximum efficiency of only 24%). Even very small amounts of non-parallelized or non-vectorized code can cause very significant losses in overall efficiency. Another way of looking at this, however, is to note that once a relatively high f has been obtained, even small increases in the amount of parallelization or vectorization can yield very significant rewards. On the other hand, when one is working at smaller values of f, small increments in f will not have a significant impact.

In a vector processor, to obtain a high speedup factor one must be concerned with writing code so that as much of the code as possible consists of vector operations. Vectorizing compilers can now do a good job of producing code that is vectorized; unfortunately, the compiler does not know as much about the purpose of the code as does the programmer. Because of this, it is still necessary for the programmer to write code in such a way that the compiler can recognize as much vectorizable code as possible. Vectorization can be inhibited by many different things. First, compilers generally vectorize only DO loops. Loops created with IF statements will not vectorize and should be avoided. Vectorization may also not take place if certain statements are within the loops. For example, loops with IF statements, subroutine calls, or statements with recursion will normally not vectorize nor will loops with irregular addressing. Strategies that avoid some of these problems vary from simply removing a statement from a loop to splitting a loop into multiple loops to using special vector functions such as gather/scatter. Even when a loop can be vectorized, it may not execute as fast as it might if the code were altered. For example, loops may often be speeded up by unrolling the loop to reduce the number of memory references. Nested loops may often be speeded up by changing the order of the loops so that the longest loop is vectorized. It should be noted that codes that vectorize on one computer may not vectorize on another vector computer, or may not show as much of a speed increase if they do vectorize. For example, the Cyber 205 operates most efficiently on very long vectors, while the Cray-1 can operate efficiently on shorter vectors.

For a multiprocessor, the goal is essentially to use solution algorithms and write code that allows a job to be split into many separate tasks, some of which can be executed simultaneously. On machines where each processor shares some global memory this often results in some sort of task queue. The next task in the queue is begun when a processor becomes available. On a multiprocessor without global memory, however, it is often necessary to specify not only the tasks but to determine which processor should do each task. It is also necessary to determine in which local memory variables will be kept.

Strategies for using multiprocessors exhibit varying degrees of synchronization. The most synchronized approach would be to use a multiprocessor as an SIMD computer by programming so that all processors are performing the same operation at the same time. A slightly less synchronized method would be for all processors to start different tasks at the same time. As processors complete their tasks they become idle until the last processor finishes its task. They may all begin on their next task then. This method is often easy to program as it involves a sequence of parallel steps. In such an algorithm it becomes important to keep the time for the completion of a task about the same for all processors in order to minimize the amount of time that processors remain idle while waiting for the other processors to finish. A more efficient method is to decompose the problem but not require that all tasks start simultaneously. This allows processors that have completed their tasks to begin new tasks immediately, if the necessary data are available. In this case it is necessary to synchronize only so a processor does not attempt to use an operand that has not yet been calculated by another processor. This synchronization can be done by several different methods. One is by causing a processor to wait until a certain necessary event has occurred. Another is to cause a processor to wait for the completion of a certain task. In this case, the size of the separate tasks in the code (or granularity) becomes a critical factor in obtaining maximum efficiency. In the case of very small granularity, machines are being developed for which this synchronization can be done on the machine level, in what is known as a data flow machine. For larger granularity the synchronization is usually done by the programmer

or the compiler.

Another major concern to the multiprocessor programmer is how to structure the data in the multiprocessor. This is normally not a problem for a shared-memory multiprocessor, but is extremely important in machines with local memory, such as computers with a hypercube architecture. If not done properly, the memory transfer time could dominate the total execution time of the program.

To execute efficiently, vector multiprocessors obviously require algorithms and programs that exploit both vector operations and parallelism. However, the need for long vectors and the need for several independent tasks can sometimes be competing demands. For example, in the implementation of a nested dissection algorithm on a vector computer, the factorization time can be decreased by stopping the dissection process short of completion (George et al., 1978). The incomplete dissection yields longer vectors than the original scheme, but it also yields fewer submatrices that can be factored in parallel.

In order to illustrate a few of the considerations that may arise in applying advanced computer architectures, we now turn to a specific example, sparse matrix methods for equation-based process flowsheeting.

EQUATION-BASED FLOWSHEETING

Equation-based (EB) process flowsheeting can provide an efficient alternative to traditional sequential modular flowsheeting packages, and is today attracting increasing commercial interest and use.

The basic idea behind EB flowsheeting is very simple. A process is modeled by a large set of nonlinear equations, which is then solved simultaneously, using Newton-Raphson or some related technique. Reviews of the EB approach have been provided by Shacham, et al. (1982), Perkins (1984), and Stadtherr and Vegeais (1985c).

The periodic solution of such a large sparse set of linear equations is a key step in the solution of the set of nonlinear flowsheeting equations. The sparse matrices that arise in these problems do not have a highly regular structure nor do they have desirable numerical properties such as diagonal dominance or positive definiteness. Because of this, a direct and general-purpose sparse matrix solver is generally required. Unfortunately the direct solution of large sparse sets of linear equations without regular structure is a problem that has not been particularly amenable to advanced computer architectures.

The problem with conventional codes is that in order to eliminate storage of the zeros in the sparse matrix, extensive use is made of indirect addressing. Experience has been (Duff and Reid, 1982) that these sparse codes do not vectorize well due to this indirect addressing. Furthermore, there is little recoding that can be done to improve the situation. What is needed is a rethinking of the algorithm involved. On multiprocessors there has been rather little experience with these codes; but again it appears that a rethinking of algorithms is required. Thus we seek to develop strategies for effectively using vector processing and multiprocessing architectures in solving the sparse matrix problems that arise in EB flowsheeting. As was shown by Stadtherr and Wood (1984a,b), flowsheeting matrices do have a structure that can be exploited on sequential computers by a sparse matrix solver. This structure arises from the fact that a process is composed simply of unit operations with few interconnections between them. This results in a matrix that is generally block-banded with several off-band blocks. The question we will look at briefly here, and in more detail elsewhere (Vegeais and Stadtherr, 1987a,b) is how this structure can be exploited on vector or parallel processors. It should also be pointed out that although the solution of the sparse matrix problems is a key computational step in EB flowsheeting, there are other computationally intensive steps as well, especially the evaluation of physical properties. From Amdahl's law, it is clear that in order to effectively speed up an EB flowsheeting package, the vectorization or parallelization of all parts of the package must be considered.

Vector Processing Considerations

On vector computers, it is desirable to operate on data in contiguously or regularly indexed vectors. There are at least two ways this may be done in a sparse matrix algorithm. The first is by the use of gather and scatter and the second is to treat parts of the sparse matrix as if they were full submatrices.

In the gather/scatter method, the needed indexed data is gathered into contiguous locations. Vector operations are then performed on this data. Finally, the results are scattered back into their original indexed locations. Most experience has been that this takes little advantage of the possible speedup of vector processors (Petersen, 1983). However, as will be discussed later, some recent results (Lewis and Simon, 1986) using gather/scatter in hardware on the Cray X-MP/24 suggest that gather/scatter offers potentially much better results than previously thought.

In order to treat parts of the sparse matrix as dense, some matrix structure must be located. Two forms that often occur are dense blocks and diagonal bands. These can be taken advantage of by block-oriented solvers (Calahan, 1979) and frontal solvers (Duff, 1984), respectively. Both of these types of structures can be seen in process flowsheeting matrices.

The block structure in flowsheeting matrices arises from the modular nature of chemical processes. The blocks that occur are themselves sparse, however. Treating them as full can add somewhat to the amount of storage required and will result in performing many wasted operations on zero elements. Another problem in applying this approach to process flowsheeting matrices is that it is often necessary to perform row or column interchanges in order to preserve numerical stability. These interchanges require the redefinition of the blocks and interrupt vector processing.

The frontal approach originated as a band solver for finite element problems (e.g., Irons, 1970). In this method, operations are confined to a relatively small submatrix, called a frontal matrix, that can be regarded as full. This matrix moves down the diagonal as the solution

proceeds.

As mentioned before, process flowsheeting matrices are block-banded with some off-band blocks. While the matrices are not strictly banded, Stadtherr and Vegeais (1985b) found the frontal approach to be effective in solving flowsheeting matrices. These results and subsequent results from Vegeais and Stadtherr (1987a) show that, by using the frontal approach, a percent vectorication of about 90% can be obtained with only about 30% wasted operations on zeros. The performance of the frontal approach depends strongly on the ordering of the rows in the matrix. Vegeais and Stadtherr (1987a) have considered a number of different reordering schemes for flowsheeting matrices, and have found that a cheap and effective reordering can be found by simply reversing the row order obtained from the BLOKS reordering algorithm of Stadtherr and Wood (1984a).

As mentioned above, the addition of hardware gather/scatter may affect the performance of direct sparse matrix solvers on vector computers. Hardware gather/scatter allows vector memory access to randomly indexed vectors, such as those that arise in general direct sparse matrix solvers. This feature is now available on new Cray computers.

Lewis and Simon (1986) have compared the relative execution times for the factorization of seven test sparse matrix problems on a Cray X-MP/24 with hardware gather/scatter and without hardware gather/scatter. The test matrices chosen were all symmetric and positive definite. Six of the seven matrices were from the finite element method while the seventh arose from an electric power network problem. For the six finite element problems, fairly good speedups of 5.55-7.79 were observed. For the seventh problem however a speedup of only 1.34 was observed. At least part of the reason for this appears to be that the electric power problem is more sparse than the other problems. This results in shorter vectors and, therefore, a smaller speedup. Since flowsheeting matrices are not symmetric and positive definite, it is hard to draw any direct conclusions from these results regarding the likely impact of hardware gather/scatter in solving flowsheeting problems. It is our observation, however, that typical flowsheeting matrices seem to more closely resemble power network matrices than those from finite element problems.

Multiprocessing Considerations

Multiprocessing involves identifying independent tasks to be executed simultaneously. For general sparse matrices this has been discussed by Calahan (1973), Conrad and Wallach (1977), and Peters (1985). All essentially considered permuting the matrix to bordered diagonal form (BoDF). Figure 1 shows a matrix in BoDF. This form has been considered because the pivoting on elements of D, the elimination of S, and the updating of submatrix T can be done in parallel. There is a possibility of memory conflicts when updating T, however.

With a matrix in BoDF, each processor can be assigned a pivot in the diagonal submatrix D. Each processor can divide the elements in its row of R by the pivot element for that row independently. Also, the elimination of the elements in S can proceed in parallel because each processor can eliminate all the variables in its own column. The corresponding necessary updates in T can also be computed in parallel.

Although the updates of T can be calculated independently, it is possible that there could be conflicts involving more than one processor attempting to add the update to an element of T at the same time. If R and S are sparse, however, this will be rare. If R and S are rather dense, though, this conflict will cause some loss of efficiency on many machines. However, some machines, like the NYU Ultracomputer (Patt, et al., 1984) are able to handle this problem through the use of a smart switching network and a "fetch and add" command. The fetch and add instruction is intended primarily for use with indices that are likely to be accessed often during the execution of a program. With the fetch and add instruction, the value of the element in memory can be incremented as the element is being fetched. In addition, if more than one processor attempts to access the same element in memory, the requests and increments can be combined so that no delay is required for this memory conflict. The fetch and add instruction could be used to add the updates to the T submatrix. The fetch part is actually unnecessary in this case.

After pivoting on the elements in the diagonal submatrix D, the updated submatrix T still must be eliminated. The overall solution proceeds recursively:

STEP 1
-Reorder (Vegeais and Stadtherr, 1987b)
-Do parallel pivoting on D
STEP 2
-Reorder remaining submatrix $T-SD^{-1}R$
-Do parallel pivoting on diagonal part
STEP 3
-Etc.

This procedure has two drawbacks. First, it does not take advantage of the block structure of the flowsheeting matrices. Also, each step includes a reordering. This reordering is not done in parallel which could significantly slow down the overall computation rate. In order to avoid these problems, a bordered block diagonal form (BoBDF) can be used.

Figure 2 shows a matrix in BoBDF. This matrix can be solved in a method analogous to the solution of the BoDF. Instead of each processor working on a diagonal element, each processor works on a block. The blocks in flowsheeting matrices are not of equal size, however, and this could cause an unbalanced work load across the processors, resulting in a low speedup. The task granularity is too large in this case.

To reduce granularity, each of the diagonal blocks can be reordered into BoDF (Figure 3). This allows pivots from the diagonal parts from all the blocks to be performed at the same time. Also, the reordering step is able to be done in parallel since each diagonal block can be ordered independently.

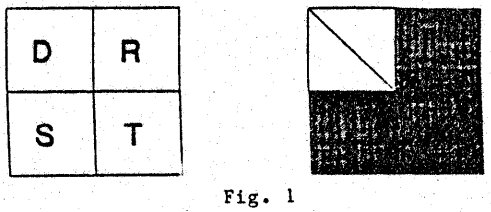

Fig. 1

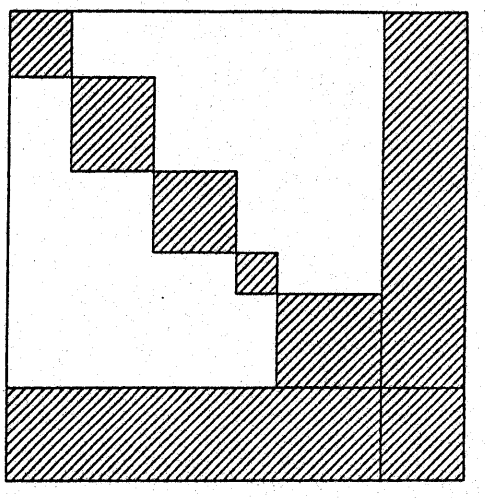

Fig. 2

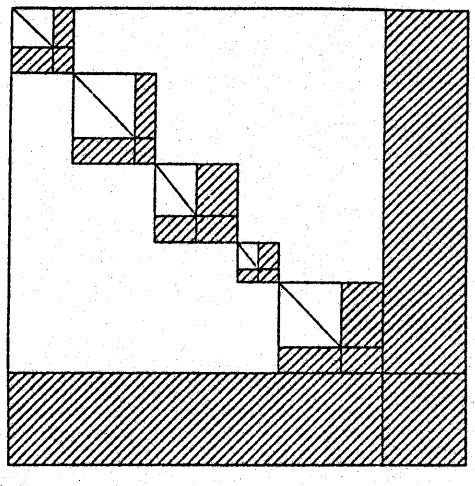

Fig. 3

Table 1 shows some typical results of solving flowsheeting matrices in parallel. It can be seen that about half of the rows of the matrix can be used as parallel pivots initially. It should be pointed out that, in obtaining these results, the block structure was not explicitly used as described, although it was still exploited. Much more detailed results are presented in Vegeais and Stadtherr (1987b).

As the pivoting proceeds, the number of parallel pivots that can be found generally decreases with each step. This is because the submatrix is smaller and generally more dense at each succeeding step. As the lower right hand block fills up, it is necessary at some point to switch to a full matrix solver. It appears that this should be done after five or six parallel pivoting steps.

Unlike general sparse solvers on parallel computers, efficient parallel full matrix solvers have been developed. For example, Neta and Tai (1985) report a speedup of 7.96 on a 20 x 20 full matrix on an 8 processor multiprocessor simulator. Geist (1985) reports a speedup of about 12 in solving a 128 x 128 matrix on a 16 processor hypercube. Crowther, et al. (1985) report a speedup of 119 in solving a 1200 x 1200 matrix on a 128 processor BBN Butterfly.

The speedup of the sparse portion of the procedure is shown in Table 2. These numbers look very encouraging. Using Amdahl's law we can calculate that the percent parallelized code is about 70% to 75% after the first five parallel pivots have been completed. At this range of f, a small increase in parallelization can produce a large gain in speedup.

Table 1

Parallel Pivoting on a Matrix with Flowsheeting Structure

(N=372)

Step	Number of Parallel Pivots	Density of Remaining Matrix
1	181	.09
2	47	.25
3	2	.27
4	1	.27
5	55	.76
6-23	1	.77-.86
24	5	.88
25-68	1	
etc.		

Table 2

(N=372)
Cumulative Speedup
Processors

Step	2	4	8	16	∞
1	1.5	2.3	3.6	5.1	8.5
2	1.4	2.3	4.5	7.0	15.3
3	1.4	2.2	3.5	5.4	9.0
4	1.4	2.2	3.5	5.4	9.0
5	1.6	2.1	2.7	3.1	3.8
Switch to Full Solver					

CONCLUDING REMARKS

The ever-increasing demand for more computing power has manifested itself not only in the popularity of single and multiple vector processors with extremely fast clock cycles, but in the advent of innovative configurations of arrays of microprocessors as well. The two predominant trends in advanced architectures are the use of a few extremely powerful processors in parallel and the use of very many microprocessors in parallel. Whether there will continue to be a market for both classes of machines is still unresolved. The former type of machine has firmly established itself as a valuable scientific tool. However, developments within the latter class indicate that its members may be economical alternatives to expensive machines like the Cray-2. In either case, an understanding of the parallel nature of both the architecture and the algorithm is necessary to fully exploit the machine's capabilities. And as computational requirements continue to grow, the use of parallel architectures in all branches of science and engineering seems inevitable.

References

Amdahl, G. M., Validity of the Single-processor Approach to Achieving Large Scale Computing Capabilities, AFIPS Conf. Proc., 30, 483 (1967).

Bucher, I. Y., The Computational Speed of Supercomputers, Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer System, p. 151 (1983).

Calahan, D. A., Parallel Solution of Sparse Simultaneous Linear Equations, in Proceedings of the 11th Annual Allerton Conference on Circuits and System Theory, (1973).

Calahan, D. A., A Block Oriented Sparse Equation Solver for the Cray-1, in Proceedings of the 1979 Conference on Parallel Processing, IEEE Computer Society Press, Silver Spring, MD (1979).

Conrad, V. and Y. Wallach, Parallel Optimally Ordered Factorization, in Proceedings 1977 Power Industry Applications Conference (1977).

Crico, A. M., ALAMBIC--A Vectorized Batch Rectification Simulator Running on Supercomputers, Paper #55e, AIChE Annual Meeting, Miami Beach (1986).

Crowther, W., J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, Performance Measurements on a 128-node Butterfly Parallel Processor, in Proceedings of the 1985 International Conference on Parallel Processing, IEEE Computer Society Silver Spring, MD (1985).

Dongarra, J. J., Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment, Technical Memorandum No. 23, Argonne National Laboratory, Argonne, IL (1986).

Duff, I. S., Design Features of a Frontal Code for Solving Sparse Unsymmetric Linear Systems Out-of-core, SIAM J. Sci. Stat. Comput., 5, 270 (1984).

Duff, I. S. and J. K. Reid, Experience of Sparse Matrix Codes on the Cray-1, Computer Physics Communications, 26, 293 (1982).

Geist, G. G., Efficient Parallel LU Factorization with Pivoting on a Hypercube Multiprocessor, Report ORNL-6211, Engineering Physics and Mathematics Division, Mathematical Sciences Section, Oak Ridge National Laboratory (1985).

George, A., W. G. Poole, Jr., and R. G. Voigt, Analysis of Dissection Algorithms for Vector Computers, Comp. Math. Appls., 4, 287 (1978).

Hwang, K., Supercomputers: Design and Applications, IEEE Computer Society Press, Silver Spring, Maryland (1984).

Irons, B. M., A Frontal Solution Program for Finite Element Analysis, International J. for Numer. Methods in Eng., 2, 5 (1970).

Kuhn, R. H. and D. A. Padua, Tutorial on Parallel Processing, IEEE Computer Society Press, Silver Spring, Maryland (1981).

Lerner, E. J., Parallel Processing Gets Down to Business, High Technology, 5(7), 20 (1985).

Levine, R. D., Supercomputers, Scientific American, 246(1), 118 (1982).

Lewis, J. G. and H. D. Simon, The Impact of Hardware Gather/Scatter on Sparse Gaussian Elimination, Mathematics & Modeling Technical Report ETA-TR-33, Boeing Computer Services (1986).

Ortega, J. M. and R. G. Voigt, Solution of Partial Differential Equations on Vector and Parallel Computers, SIAM Review, 27, 149 (1985).

Neta, B. and H.-M Tai, LU Factorization on Parallel Computers, Comp. & Math. with Appls., 11(6), 573 (1985).

Patt, Y. N., R. G. Sheldon, M. Shebanow, C. Ponder, and W.-M. Hwu, A Comparison of Several Evolving (University) Supercomputer Architectures, in Proceedings of the 1984 4th Jerusalem Conference on Information (1984).

Perkins, J. D., Equation-oriented Flowsheeting, in Proceedings of the Second International Conference on Foundations of Computer-Aided Process Design (eds. A. W. Westerberg and H. H. Chien), CACHE (1984).

Peters, F. J., Parallelism and Sparse Linear Equations, in Sparsity & its Applications, Cambridge University Press (1985).

Petersen, W. P., Vector FORTRAN for Numerical Problems on Cray-1, Communications of the ACM, 26(11), 1008 (1983).

Senior, P. R., Simulation of Large Dynamic Systems, Paper #108g, presented at AIChE Annual Meeting, Miami Beach (1986).

Shacham, M., S. Macchietto, L. F. Stutzman, and P. Babcock, Equation Oriented Approach to Process Flowsheeting, Computers and Chemical Engineering, 6, 79 (1982).

Stadtherr, M. A. and J. A. Vegeais, Advantages of Supercomputers for Engineering Applications, Chem. Eng. Prog., 81(9), 21 (1985a).

Stadtherr, M. A. and J. A. Vegeais, Process Flowsheeting on Supercomputers, IChemE Symp. Ser., 92, 67 (1985b).

Stadtherr, M. A. and J. A. Vegeais, Recent Progress in Equation-based Process Flowsheeting, in Proceedings of the 1985 Summer Computer Simulation Conference, Society for Computer Simulation (1985c).

Stadtherr, M. A. and E. S. Wood, Sparse Matrix Methods for Equation-based Chemical Process Flowsheeting--I. Reordering Phase, Computers and Chemical Engineering, 8, 9 (1984a).

Stadtherr, M. A. and E. S. Wood, Sparse Matrix Methods for Equation-based Chemical Process Flowsheeting--II. Numerical Phase, Computers and Chemical Engineering, 8, 19 (1984b).

Vegeais, J. A., A. B. Coon, and M. A. Stadtherr, Advanced Computer Architectures: An Overview, Chem. Eng. Prog., 82(12), 23 (1986).

Vegeais, J. A. and M. A. Stadtherr, Vector Processing Strategies for Sparse Matrix Problems in Equation-Based Flowsheeting, submitted for publication (1987a).

Vegeais, J. A.. and M. A. Stadtherr, Parallel Processing Strategies for Sparse Matrix Problems in Equation-Based Flowsheeting, submitted for publication (1987b).