ADVANTAGES OF SUPERCOMPUTERS (AND OTHER ADVANCED COMPUTER ARCHITECTURES) FOR CHEMICAL ENGINEERING APPLICATIONS

Mark A. Stadtherr

Chemical Engineering Department

University of Illinois

1209 W. California Street

Urbana, Illinois 61801

U.S.A.

In 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. A basis for comparing the performance of such machines; 4. Challenges in developing algorithms and codes that take best advantage of advanced computer architectures.

These 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.

Advanced Computer Architectures

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). For example, Dongarra reports that in solving a 100 x 100 set of linear equations using LINPACK, with the basic linear algebra subroutines (BLAS) in Fortran, the Cray X-MP (using four processors) executes at a rate of 49 MFLOPS (millions of floating point operations per second), the Cray X-MP (using one processor) at 24 MFLOPS, the IBM 3081 K and CDC Cyber 175 at 2.1 MFLOPS, the VAX 11/780 (with floating point accelerator) at 0.14 MFLOPS, and the IBM PC/AT (with 80287 math coprocessor) at 0.012 MFLOPS. A problem this small is actually too small for a supercomputer to achieve its peak speed, and the LINPACK algorithms and code were not designed with a supercomputer architecture in mind. For instance, Dongarra also reports that on a 1000 x 1000 problem using different code, the Cray X-MP (using 4 processors) executes at a rate of 713 MFLOPS.

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. Perhaps the most widely used taxonomy involves the four categories originally proposed by Flynn and described in some detail by Hwang and Briggs (1984). These four categories are: single-instruction single-data-stream (SISD), single-instruction multiple-data-stream (SIMD), multiple-instruction single-data-stream (MISD), and multiple-instruction multiple-data-stream (MIMD).

In the SISD machine (which includes most conventional computers) only one instruction is executed at a time, and the instruction operates on only one data stream. In an SIMD computer there is an array of separate processors, each capable of doing the same instruction at the same time on a different data stream. For example, a single instruction could compute the sum of two vectors. Assuming enough processors were available, all the additions of the corresponding elements of the operand vectors could be done simultaneously. In this case the processing is synchronous; that is, all processors must begin to execute the instruction at the same time. It should also be noted that various schemes can be used to interconnect the processors, and to connect them to memory. In a MIMD computer, there is again an array of separate processors, and again the processors may be connected to each other and to memory in a variety of ways. In this case, however, each processor is capable of doing <u>different</u> operations on different data streams. Also, in a MIMD machine, processing is usually asynchronous; that is, the different processors may start their different operations at different times. Finally, in the MISD machine, a single data stream is operated on by a sequence of different

processors, each capable of different instructions. This last category is considered by some to be not very useful, because it is regarded as impractical and there are no real implementations. A second, and more serious, problem with this classification scheme is that there is no good agreement on where to categorize "pipelined" machines. They have been categorized by various people as SISD, SIMD, and MISD. For these reasons, it is perhaps better (and simpler) to classify advanced computer architectures into three categories: vector processing, multiprocessing, and vector multiprocessing (Bucher, 1983).

The vector processing category includes SIMD machines, as well as "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. Rudimentary forms of pipelining can also be found in some conventional machines.

As discussed above, 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 MIMD architectures that use an array of scalar processors, each capable of executing different instructions at the same time on different data. Today the term "parallel processing" is generally regarded as synonymous with multiprocessing. It should be recognized however that "parallel processing" is a term whose meaning has evolved over many years, and which thus may be misleading in some circumstances. 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. Some conventional machines can also be thought of as "loosely-coupled" multiprocessors (Hwang and Briggs, 1984).

The vector multiprocessing category ("parallel vector processing") 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. There are currently several unresolved issues regarding multiprocessing and vector multiprocessing architectures. Some of these are briefly discussed here.

One issue concerns the number and power of the processors to be used. To look at the two extremes, one can either use a very large number of inexpensive and relatively slow microprocessors, or a relatively small number of expensive and very fact vector processors. The latter extreme is firmly entrenched commercially in machines such as the Cray X-MP, but the former extreme may provide economical alternatives.

Another issue involves memory access. Global memory is directly accessible by all processors. However, when there is a large number of processors, this provides the potential for conflicts caused by different processors trying to access the same memory bank at the same time. Local memory is directly accessible by only a single processor, so memory conflicts are not a problem. However, in this case, if one processor needs data stored in another processor's memory, then that data must be transferred from one processor to the other. Such interprocessor communication can be quite slow relative to the computational speed and thus may become a bottleneck. There are of course compromises between these two extremes. For instance, one can use a set of separate memories that can be dynamically assigned, through some sort of switching network, to different processors while a program is executing. Or, some hierarchy of memories could be used: there could be some global memory shared by all processors, some memories shared by clusters of processors, and then local memories assigned to each individual processor.

A third issue involves the schemes used to interconnect the processors. Such schemes may either be "static", with fixed interconnections between processors, or "dynamic", with interconnections that are switchable while a

program is executing. In designing a static interconnection scheme there tend to be two competing goals. First of all, one would like to have efficient data transfer between any two processors. In practice this usually means that sending data from one processor to another should require going through a path involving relatively few other processors. From this standpoint of this goal, one would ideally like to have every processor connected directly to every other processor. However, with a large number of processors this is infeasible, and so a second goal must be to use reasonably few direct processor to processor connections. Various static interconnections have been used; the most popular one today is probably the hypercube, or n-cube. The number of processors in an n-cube is 2^n, each is connected directly to n other processors, and the worst path for interprocessor communication goes through only n interconnections. A 2-cube can be easily visualized as a square, with a processor at each vertex and each edge representing a connection. To make a 3-cube, two 2-cubes are connected at corresponding vertices, thus forming what is easily visualized as a three dimensional cube. In general to form an n-cube, one takes two (n-1)-cubes and connects corresponding vertices. Overall, the hypercube is regarded as a compromise between the number of interconnections and efficient data transfer. One commercial example is the Intel i-PSC, currently available in up to a 10-cube. Dynamic interconnection schemes utilize some sort of switching system to change the connections between processors, or between processors and memory, while a program is executing. These systems may be relatively simple ones such as the crossbar scheme, or more sophisticated ones such as the banyan network or omega network (Vegeais et al., 1986). In general the more sophisticated networks require fewer switches to interconnect a given number of processors and memories than the crossbar scheme. One commercial example of a machine using such a network

is the BBN Butterfly.

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 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 (e.g., Stadtherr and Vegeais, 1985b).

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.

Performance

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. In fact for f less than about 80%, the value of P tends to make relatively little difference in the overall speed-up S. Unfortunately, many large general-purpose scientific codes, NASTRAN for example, do not vectorize more than about 50-60%, nor do they parallelize significantly better.

As noted by Levesque (1986), Amdahl's law can be observed in the marketplace. If one looks at a pair of supercomputers from Fujitsu, the VP-100 and VP-400, one can observe that even though the VP-400 is four times faster in vector mode than the VP-100, many application programs do not vectorize sufficiently to utilize this speed. Thus, the slower (and cheaper) VP-100 has been commercially more successful. Similarly, Levesque (1986) has compared the computational and marketplace performance of three supercomputers from the mid-1970s. The base case for this comparison is the CDC 7600, which was regarded as the fastest conventional mainframe in the mid-1970s. The three supercomputers compared are the ILLIAC IV and STAR 100, both early supercomputers, slower than the 7600 by a factor of four in scalar mode, but with vector speeds 16-20 times faster than the 7600, and the Cray-1, just introduced in the mid-1970s, with a scalar speed twice that of the 7600, but a vector speed only ten times as fast as the 7600. Using Amdahl's law it is easy to show that, because of their slow scalar speeds, the ILLIAC IV and STAR 100 will not outperform the CDC 7600 until the percent vectorization reaches about 77%. On the other hand, because of its higher scalar speed, the Cray-1 will always outperform the 7600, and will outperform the ILLIAC IV and STAR 100 at percentages of vectorization less than about 98%. Based on Amdahl's

law, it is not surprising that the ILLIAC IV and STAR 100 were commercially unsuccessful, while the Cray-1 was. The Cray-1 was so successful in this marketplace because it offered both higher scalar speed and vector processing, even though its vector speed was not the best available.

These arguments can be extended to today's multiprocessing architectures. For instance, for a parallelization of 60%, Amdahl's law shows that there is little to be gained in overall speed-up by having more than about 6-8 processors. Even for 90% parallelization, which is rare in most scientific computing today, there is little to be gained by having more than about 30-35 processors. This suggests that large arrays of parallel processors may not be useful as a general-purpose computing resource. However, as emphasized by Levesque (1986), the situation regarding parallel processors on a large number of processors is not as bad as it might seem. For instance, today there are some applications for which very high levels of parallelization (approaching 100%) are possible; among these are signal processing, image processing, and some finite difference codes. Furthermore, in the future we can expect numerical algorithms and codes that take much better advantage of parallelism than today. In part this may be done using programming languages with built-in parallelism, so that compilers can optimize object code to best utilize parallel processing (today's Fortran is difficult for compilers to optimize for parallelism).

Algorithms and Codes

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, nested loops may often be speeded up by changing the order of the loops so that the longest loop is vectorized. Also, loops may often be speeded up by "unrolling" them to reduce the number of memory references. The concept of unrolling is demonstrated in this example. Consider the following code, which computes the product of the M x N matrix A and the vector X (assuming Y(J) has been initialized to zero for all J):

```
      DO 200  I=1,N
          DO 100   J=1,M
 50         Y(J)=Y(J)+A(J,I)*X(I)
100       CONTINUE
200   CONTINUE
```

The inner loop in this code is vectorizable (though it may take a special compiler directive to do it), and thus Y(J) and A(J,i), i=1,...,N, will be operated on as vectors. The execution of this loop thus requires three vector memory references: two vector loads (on the right-hand-side of line 50) and one vector store (left-hand-side of 50). On the Cray-1, there is only one path to and from memory, thus moving data to and from memory can become a bottleneck. The data transfer rate (or bandwidth) to and from memory on the Cray-1 is 80 million words per second (Mword/sec), thus an upper bound on the computation of the inner loop is 80/3 Mword/sec. Since computing each element (word) of Y(J) requires two floating point operations, the upper bound in MFLOPS is 2*(80/3) ≈ 53 MFLOPS. Now consider unrolling this inner loop to a "depth" of four, as shown in the following code:

```
      DO 200  I=4,N,4
           DO 100  J=1,M
50            Y(J)=((((Y(J)+A(J,I-3)*X(I-3))
                              +A(J,I-2)*X(I-2))
                              +A(J,I-1)*X(I-1))
                              +A(J,I)*X(I))
100        CONTINUE
200   CONTINUE
```

Again the inner loop can be made to vectorize. (The extra parentheses in 50 ensures that the compiler produces code that performs the operations in the most efficient order. Also, if N is not a multiple of four, some additional lines of code will be needed.) Now the inner loop requires five vector loads and one vector store, so the upper bound on execution rate is 80/6

Mwords/sec. But there are now eight floating point operations required to compute each element of Y(J) so the upper bound in MFLOPS is 8*(80/6) ≈ 107 MFLOPS. So we have doubled the maximum possible execution rate. Note that this is essentially the result of reducing the total number of memory references. In the original code the inner loop was executed N times, and each time required three memory references, for a total of 3*N. In the unrolled code the inner loop was executed N/4 times, and each time required six memory references, for a total of 1.5*N. In this particular example, where data transfer to and from memory is the rate-limiting step, the halving of memory references naturally results in a doubling of the maximum execution rate. In other cases, such as on the Cray X-MP, where there are more paths to and from memory, data transfer to and from memory may not be rate limiting; nevertheless, the unrolling of loops may still be a good programming practice, since by reducing the number of memory references the likelihood of memory conflicts will be reduced, and also this reduces the computational overhead due to the start-up of vector operations.

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 or all of which can be executed simultaneously (multitasking). 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 (microtasking), 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 (macrotasking) 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.

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. It may be necessary to think not only of new algorithms, but new programming languages. 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).

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

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).

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).

Hwang, K. and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

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).

Levesque, J. M., Effective Utilization of Parallel Vector Processors, Paper #91b, AIChE Annual Meeting, Miami Beach (1986).

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

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

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).

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