Parallel Processing Strategies for Chemical Process Flowsheeting

James A. Vegeais and Mark A. Stadtherr
Dept. of Chemical Engineering, University of Illinois, Urbana, IL 61801

The potential for using the sequential-modular, simultaneous-modular, and equation-based approaches to process flowsheeting on multiprocessing (parallel processing) computer architectures is assessed. The simultaneous-modular and equation-based problem formulations appear to be the most promising, but the sparse matrix techniques currently used, especially for the latter, do not perform well on parallel machines. We consider two different decomposition schemes for parallelizing the sparse matrix problems that must be solved. These are the bordered-diagonal form and the bordered-block-diagonal form. The latter takes advantage of the inherent block structure of the flowsheeting matrix and represents a promising strategy for solving flowsheeting problems on parallel machines, especially for large problems.

Introduction

Parallel computing architectures represent the inevitable future of high-performance computing, since the speed of single processors is limited by the speed of light. Parallel processing architectures can now be found in relatively high-priced supercomputers, as well as relatively low-priced workstations.

Historically, the need to solve complex problems in science and engineering has led to the development of larger and faster computers, which in turn has led to the formulation of still more complex problems, the development of still faster computers, and so on. Process flowsheeting has not been a large driving force in the development of today's computer power. Flowsheeting practitioners, however, have always taken advantage of advancements in computer power by formulating and solving new and more complex problems. We will undoubtedly continue to see this happen.

For process flowsheeting problems, the advantages of using advanced computer architectures, such as vector processing and parallel processing, have been recognized (for example, Stadtherr and Vegeais, 1985a; McRae et al., 1988; Haley and Sarma, 1989; Vegeais and Stadtherr, 1989; McRae, 1990). These include greater design productivity and cost effectiveness, improved man-machine interaction, and the ability to solve much more complex problems. The difficulty is that to achieve these benefits we must be able to use problem formulations, solution algorithms, and computer codes that effectively exploit these architectures. For process flowsheeting, it appears that the best way to do this is to assess the problem formulations now used and to rethink the solution algorithms used. These are the main problems to be addressed in this article.

Thus, we will consider here the use of parallel processing on each of the three standard problem formulations for the process flowsheeting problem: sequential modular (SeqM), simultaneous modular (SimM), and equation based (EB). We will conclude that the last two appear to be the best suited to parallel computer architectures, but there is a strong need to rethink the sparse matrix methods used, since the usual techniques do not parallelize well. This problem is discussed in detail, as are some promising approaches to its solution.

Similar analyses from the standpoint of vector processing have been performed earlier by Stadtherr and Vegeais (1985b), Zitney and Stadtherr (1988), and Vegeais and Stadtherr (1990).

Background

There are many kinds of parallel computing architectures. Basic details concerning these are available elsewhere (for example, Kuhn and Padua, 1981; Hwang, 1984; Hwang and Briggs, 1984; Vegeais et al., 1986) and will not be discussed here. We do discuss here, however, some measures of algorithm

Correspondence concerning this article should be addressed to M. A. Stadtherr.
Current address of J. A. Vegeais: BP Research, 4440 Warrensville Center Road, Cleveland, OH 44128.

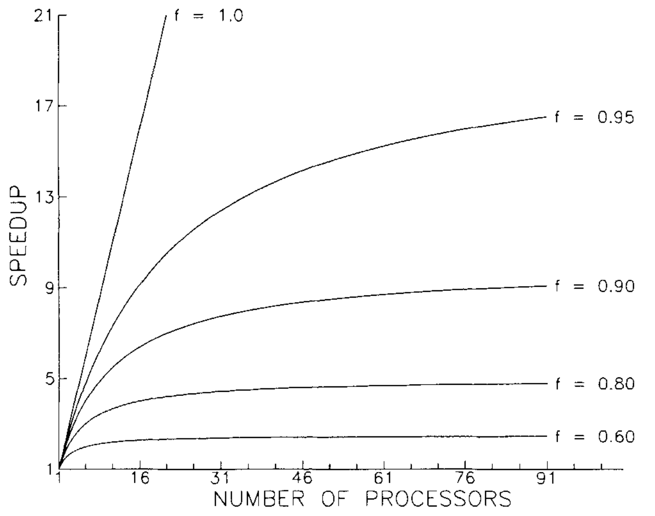

Figure 1. Amdahl's law for several values of fraction of utilized code parallelized, *f*.

performance on parallel machines, as well as some programming considerations.

The performance of an algorithm or code on a multiprocessor is often quantified in terms of its speedup. Speedup is defined here as the ratio of the time it takes for a job to execute on one processor of a machine to the time it takes for that job to execute with P processors on the same machine. Using this definition of speedup, we can also define a parallel efficiency as the speedup divided by the number of processors. There are other useful definitions of speedup as well.

A simple, but important, relationship between percentage parallelization and potential speedup is given by the well-known Amdahl's law (Amdahl, 1967). This essentially provides an upper bound on the speedup possible from a given percentage parallelization. Amdahl's Law has the form $S = P/[P(1-f)+f]$, where S represents an upper bound on the speedup, f is the fraction of utilized code performed in parallel, and P is the number of processors. This relationship is derived from a simple model of parallel processing whose assumptions include: independence of parallel tasks, a negligible amount of overhead time incurred in initiating the parallel tasks, and negligible amounts of time for data organization and interprocessor communication and synchronization. Ideally then, if a code were executed completely in parallel ($f=1$), we would get a speedup of P; however, in practice this is rarely possible (Faber et al., 1986).

As emphasized by Levesque (1986), the lesson of Amdahl's law is that even with an infinite number of processors, if there is just a small amount of code that cannot be run in parallel, the potential speedup will be greatly limited. Figure 1, a plot of Amdahl's law with S vs. P for several values of f, shows this very clearly. For instance, even if 95% of a code is able to run in parallel and an infinite number of processors are available, the maximum speedup possible is 20. Another effect, however, has been noted by Gustafson (1988) with regards to Amdahl's law. Gustafson points out that Amdahl's law assumes a fixed problem size, but that for most applications f actually increases with problem size. That is, as large and even larger problems are attacked, the speedup tends to increase.

In programming a multiprocessor one can distinguish between what might be called "low-level" and "high-level" parallelism. By low level we mean parallel tasks that can, in principle, be identified and executed automatically without user intervention. This could be done on the machine level, for example in a so-called data flow machine (Dennis, 1980; Hwang and Briggs, 1984) or by the compiler. Typically such tasks will be relatively small (fine-grained) on the DO-loop level or lower. By high level we mean opportunities for parallelism that usually must be recognized by the programmer or algorithm developer, typically based on knowledge of a specific problem or class of problems that cannot be imparted to a compiler. Generally the parallel tasks on this level will be large (course-grained) in comparison to those on the low level; thus there are opportunities for low-level parallelism within high-level tasks. In this article we will be seeking strategies for high-level parallelism on the process flowsheeting problem.

Effect of Problem Formulation

The effectiveness with which one can exploit parallel processing architectures depends considerably on how a problem is formulated. Some problem formulations may be much more amenable to parallelization than others. In this section we discuss problem formulations for process flowsheeting and assess their potential with regard to parallel processing.

There are basically three problem formulations for the process flowsheeting problem: sequential modular (SeqM), simultaneous modular (SimM), and equation based (EB). Since these are all well-known we will make no attempt to describe them here. Most commercial codes remain SeqM overall, but increasingly incorporate SimM or EB features for complex units. The EB approach, such as that implemented in the commercial code SPEEDUP, for example, is now recognized as a powerful tool for solving complex dynamic simulation problems. We now consider how each of these three basic problem formulations can be expected to perform in a parallel processing environment.

Sequential-modular approach

The SeqM approach does not appear to be a good candidate for exploiting parallelism. The approach, as its name implies, is inherently sequential. A unit module cannot begin execution until the previous module has completed execution. Thus, unless there are sections of the flowsheet that are independent, only one module can be executing at a time.

It may be possible to find some parallelism within the unit modules, however, so that several processors are at work on the one module that is executing. For example, it is possible that thermodynamic properties for all compounds in the module could be calculated in parallel, or some form of low-level parallelism could be identified by the compiler. Finally it might be possible to change the algorithms used by the modules so that a greater degree of parallelism is obtained. A difficulty based on Amdahl's law is that to realize a substantial overall speedup by trying to exploit parallelism within the modules, it would be necessary to be able to operate in parallel within virtually all of the unit modules. For flowsheets using only modules that could be internally parallelized, some good overall speedup might be obtained, but for other flowsheets the overall performance is likely to be poor.

Simultaneous-modular approach

In this approach, the same computational modules are used as in the SeqM approach. Thus the same comments made above about the potential for parallelism within the modules apply. In this case, however, there are further opportunities for parallelization. There are two variations of the SimM approach: a tear-stream formulation (for example, Biegler, 1985; Chen and Stadtherr, 1985) and a connecting-stream formulation (for example, Jirapongphan et al., 1980; Trevino-Lozano et al., 1985). Since the two formulations differ in their potential for parallelism, we will discuss them separately.

Tear-Stream Formulation. In this case the unit modules are used (in the so-called "module level" of the computation) to compute a relatively small set of simultaneous equations which is then solved for tear stream variables (in the "flowsheet level" of the computation). To calculate the Jacobian of the flowsheet-level system several sequences of modules must be called. So again, there is an inherent sequential character to this formulation. However, there are some possibilities for parallelism.

Clearly if the module sequences are independent, each can be assigned to a different processor or group of processors and their computation done in parallel. The difficulty is that normally there is some dependence among module sequences: in other words, there exists a precedence order in which the sequences should be solved. When there is a dependence between two of the module sequences, either the processor computing the dependent sequence must wait until the other processor calculates the values of the needed variables or the processor can proceed with its calculations using the values of the needed variables from a previous iteration. In the first case, the processor is idle during this time, which would greatly reduce the speedup of the program. In the second case, outdated values of the variables are used. It has been shown (Bertsekas, 1983) that using outdated values of the variables will not prevent convergence for a fixed-point calculation that is a contraction mapping. It will, however, normally slow down the convergence rate. How much it will slow down the convergence, however, cannot easily be determined *a priori.* This will vary from problem to problem. In addition, during the first iteration, no previous values exist. It is necessary to initialize these values—most likely by calculating the sequences in the order of their precedence, as is done in the first case.

In either case, there is another factor that can seriously degrade the performance; this is a factor often referred to as "load balancing." The difficulty is that normally the module sequences that must be solved will be of different sizes. This will cause the processors working on a shorter sequence to remain idle, while other processors finish their sequences. This is because it is a synchronous method. That is, all processors begin the module-level calculation at the same time and no processors can proceed to the flowsheet level until the module level is done.

There is another potential source of parallelism in this formulation. It may be possible for some of the processors to work in an asynchronous manner on the module level, while other processors work on the flowsheet level. By asynchronous, it is meant that it is not necessary for all processors to begin or end certain parts of the code simultaneously before proceeding. In this case, processors working on one level would not wait for the value being calculated on the other level. Again, not all processors will be on the same iteration at the same time and it is unclear whether the problem can be formulated to ensure convergence. The asynchronous approach to parallelism can also be applied in the other formulations discussed below.

While this formulation has more potential for parallelism than the SeqM approach, it is not very significant improvement.

Connecting-Stream Formulation. In this case the unit modules are used on the module level to compute a moderately large and sparse set of simultaneous equations which is then solved for all connecting stream variables on the flowsheet level. To calculate the Jacobian of the flowsheet-level system, all the modules in the flowsheet must be called, but the results of one module calculation are not needed for the calculation of any other module. Thus each module can be called independently. Clearly there are strong possibilities for exploiting parallelism here. Some initial work along these lines has been reported by Chimowitz and Bielinis (1987) and La Roche et al. (1988).

The obvious strategy is to assign each module to a processor and execute them all in parallel. Again there is a load balancing problem, since the tasks performed by the individual modules will be much different in length. However, compared to the tear-stream formulation, in which the module sequence for one task might consist, for instance, of a pair of mixers and a flash, and for another task might consist of three distillation columns, tasks in the connecting-stream formulation will be smaller and closer in size. This makes the load balancing problem somewhat easier, though it remains a very significant one. Techniques for attacking the load balancing problem have recently been proposed by La Roche et al. (1988).

Another issue that arises here is that of solving the sparse matrix problem that occurs on the flowsheet level. This is generally a relatively small part of the computation, but will often be large enough to significantly lower the speedup unless it is done in parallel too. Some possible techniques for doing this are discussed in detail below.

The connecting-stream formulation of the SimM approach offers reasonable potential for the use of parallelism. If one is working with an existing sequential-modular simulator, this is likely to be the best approach for use on parallel computers.

Equation-based approach

In this case the flowsheeting problem basically becomes that of solving a very large and sparse system of nonlinear equations by the Newton-Raphson method, a quasi-Newton method or some variation thereof. There are basically three parts to the computation: the evaluation of functions, the evaluation or approximation of the Jacobian matrix, and the solution of a large, sparse, linear equation system.

Within the function evaluation phase there are obvious opportunities for parallel computation. Since all the individual functions to be evaluated are independent, they can clearly all be computed in parallel. The parallel tasks will still be unequal in size, since some equations will take longer to evaluate than others; thus, load balancing remains a concern. However, the size of the tasks and the range of sizes encountered will be much less than those for the SimM formulation; thus, load balancing will be much less a concern than in the SimM case. The way in which physical property calculations are handled will have an effect here. If physical properties are computed in subroutines called during the evaluation of a function, then some functions could take significantly longer than others to evaluate, depending on the number of physical property calculations needed. On the other hand, if the equations in the physical property models are simply treated as additional equations in the overall equation system, then the function evaluation tasks will be quite similar in size, which is attractive from the standpoint of load balancing.

In Jacobian evaluation, if the elements are computed analytically, then all the elements are independent and clearly can be computed in parallel. When estimation techniques are used, such as finite difference or quasi-Newton update, the calculation can also be readily parallelized. For instance, in finite difference, each column can be perturbed independently, and all functions within each column can be evaluated independently, so the computation of each Jacobian element is independent and can be done in parallel.

While the opportunities for parallel computation within the function and Jacobian evaluation phases are obvious, this is not the case for the sparse matrix phase. Even in serial computation this is a key phase and may represent a large fraction of the overall computing time. With function and Jacobian evaluation now parallelized, the sparse matrix solver will represent an even larger fraction of the problem and will become in effect a bottleneck, preventing significant speedup, unless it is parallelized too.

The sparse matrix problem that must be solved does not have any of the desirable numerical or structural properties, such as symmetry, bandedness, diagonal dominance, and positive definiteness, which are often found in matrices arising from the discretization of partial differential equation problems and are exploitable in developing an efficient solution method for parallel computation. Thus, in EB flowsheeting on serial machines, the sparse matrix problem is usually solved using a direct, general-purpose method employing routine Gaussian elimination or some variation thereof. The general-purpose method may be modified (Stadtherr and Wood, 1984a, b) to take advantage of the structure of a flowsheeting matrix. This structure arises from the fact that a process is composed of unit operations with few interconnections between them. This results in a matrix that is generally block-banded with several off-band blocks.

Unfortunately, for matrices of the sort arising in EB flowsheeting, there is no obvious way of parallelizing the solution. Also, while there has been a significant amount of work done on parallelizing the solution of sparse matrices arising in PDE problems, there has been much less work done on general-purpose parallel solvers such as that needed here. There has been no previous work on the use of the structure of the flowsheeting matrix in developing parallel solution techniques.

In the remainder of this article, we concentrate on the sparse matrix problem arising in EB flowsheeting and on strategies for solving it using parallel computation. Provided that the sparse matrix calculations can be parallelized, the EB formulation appears to have the strongest potential for use on parallel computers.

Sparse Matrix Strategies

In this section, we look at strategies for parallelizing the solution of large, sparse, linear equation systems that arise in EB process flowsheeting, whether in the context of steady-steady simulation and design, optimization, or dynamic simulation. We start with general-purpose methods and attempt to extend them to take advantage of the structure of the flowsheeting problem. Results for two strategies are presented here. Since special-purpose techniques for the parallel solution of highly structured sparse matrices of the sort arising from PDE problems are far more advanced than those for general matrices, another approach would be to start with such specialized solvers and try to generalize them enough to handle matrices with the flowsheeting structure. This approach has been discussed by Coon and Stadtherr (1989) and is not considered here.

It should, first of all, be noted that general *full* matrices can be solved very efficiently on parallel machines. For example, Neta and Tai (1985) report a speedup of 7.96 on a 20×20 full matrix on an eight-processor multiprocessor simulator, and Crowther et al. (1985) report a speedup of 119 in solving a $1{,}200\times 1{,}200$ full matrix on a 128-processor BBN Butterfly. Dongarra (1992) also reports high parallel efficiencies on a $1{,}000\times 1{,}000$ full matrix on various parallel machines using LINPACK. Although the use of a parallel full matrix solver may be effective on small sparse systems, the wasted operations on zeros quickly make this an inefficient approach as the problem size grows. Parallel sparse matrix techniques are then required.

Parallel processing involves identifying independent tasks to be executed simultaneously. One way of doing this is to order the matrix into a form where operations on one part of the matrix do not affect other parts of the matrix. From the standpoint of the parallel solution of general sparse matrices, this has been considered by Calahan (1973), Conrad and Wallach (1977), and Peters (1985). All essentially considered permuting the matrix to bordered diagonal form (BoDF). Bordered block diagonal form (BoBDF) has also been suggested as a desirable form for the parallel solution of sparse matrices (Ortega and Voigt, 1985). We consider both matrix forms here and describe new strategies for applying them to the process flowsheeting problem.

Bordered diagonal form (BoDF)

Figure 2 shows a matrix in BoDF. The submatrix in the upper left corner, D, is a diagonal matrix, and R, S, and T are nonzero borders. This form is of interest because each element of D can be pivoted on independently and in parallel. In performing these pivots in parallel, there is a possibility of memory conflicts when updating T, however, as will be discussed below.

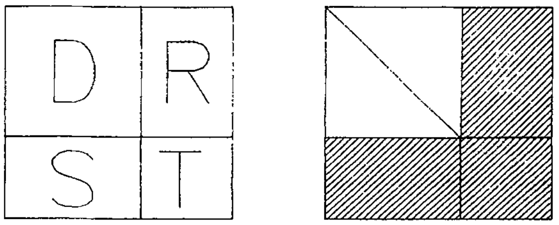

Figure 2. Matrix in bordered diagonal form (BoDF).

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 updates in T can also be computed in parallel, as the update is calculated from the value of the diagonal element and the values in the same row and column as the diagonal element. Figure 3 shows the results of one of the parallel pivoting tasks.

Although the updates of T due to each pivot 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 (Gottlieb et al., 1983) 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 results from 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.

The procedure of pivoting on the elements in the diagonal submatrix D still leaves the updated submatrix T to be pivoted upon, which could again be done by reordering to BoDF and doing parallel pivoting in the diagonal part. Thus, the overall solution proceeds recursively.

Step 1:
- Reorder to BoDF.
- Do parallel pivoting on D.

Step 2:
- Reorder remaining submatrix $T-SD^{-1}R$ into BoDF.
- Do parallel pivoting on diagonal part.

Step 3:
- Repeat with new remaining submatrix and continue steps until overall system is solved.

The question of how one should find the diagonal portion of the matrix is not trivial. It has been shown by Karp (1972) that the number of operations needed to find the maximum diagonal submatrix of a symmetric sparse matrix grows exponentially with the number of equations. The nonsymmetric case has apparently not been considered in the literature; however, finding the optimal reordering can certainly be expected to be more complex than in the symmetric case. It is not clear whether one should try to find the maximum diagonal, anyway. Peters (1985) has pointed out that in some cases pivoting on the maximum diagonal submatrix causes more overall steps in the procedure. This is because the pivots can cause fill-in in the remaining submatrix. The denser the remaining submatrix is, the more likely it is that large diagonal submatrices cannot be found in subsequent steps. For these reasons, heuristic algorithms have been considered for finding the diagonal submatrices.

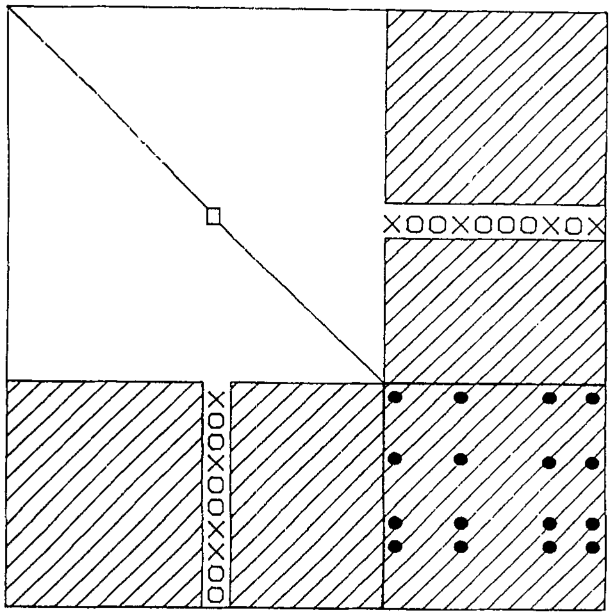

Figure 3. Result of one parallel pivot in parallel solution scheme for matrices in BoDF.

The basic heuristic (Calahan, 1973) is simple and straightforward, and proceeds conceptually as follows:

1. Consider the first row, and choose an element from it to designate as a element of D.
2. Designate other columns with nonzeros in the first row as belonging to the border R, and rows with nonzeros in the column put in D as belonging to the border S.
3. Now consider the matrix remaining when those rows and columns already assigned to D, R, and S are removed, and return to the first step.

Calahan started his procedure by choosing the leftmost element in the first row, but other choices could be made as well, based on numerical considerations perhaps. Also note that the result of this procedure depends strongly on the initial structure of the matrix, that is, the ordering of the rows (and columns if the leftmost element is always chosen in the first step). Calahan made no attempt to consider the effect of the matrix structure. He applied this method to a matrix of dimension 83×83, which arose from the simulation of a three-dimensional model of an automotive suspension system. His results were not given in a form where speedup could be determined. His results, however, did indicate that most of the speedup was achieved in the first four steps.

Conrad and Wallach (1977) also considered the parallel solution of matrices in BoDF. Conrad and Wallach, unlike Calahan, tried to take some advantage of the matrix structure in their search for diagonal submatrices. In the search, Conrad and Wallach examined the rows in the order of increasing number of nonzeros per row. This would tend to produce somewhat less fill-in in the remaining submatrix than if the rows were examined without regard to the sparsity pattern of the matrix. This method was applied to randomly generated power network matrices. The matrices ranged in size from 60×60 to 200×200. Again, a speedup could not be determined from their results. They determined, however, that after four

Table 1. Parallel Pivoting on Flowsheeting Matrices in BoDF With Natural Initial Ordering ($N = 372$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	150	0.07
2	104	0.33
3	8	0.45
4	2	0.45
5	1	0.44
6	1	0.44
7	1	0.44
8-62 etc.	1	

Table 3. Parallel Pivoting on Flowsheeting Matrices in BoDF With BLOKS Reordering ($N = 372$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	181	0.09
2	81	0.39
3	17	0.57
4	8	0.74
5	1	0.75
6	1	0.75
7	2	0.86
8-82 etc.	1	

to six parallel steps it was desirable to switch to a full matrix solver. At that point, approximately 75%-90% of the matrix had been solved. Such a switch from sparse matrix solver to dense solver has been recommended by others. For instance, Duff (1984) implemented such a switch in an experimental version of the Harwell code MA28 for the serial solution of general sparse systems.

Peters (1985) discussed the parallel solution of matrices in BoDF also. He attempted to analyze the results of Calahan. He was able to estimate that the maximum speedup ($P = \infty$) in Calahan's examples was 3.8. He also applied this approach to the solution of matrices arising from finite element calculations on a square domain. He calculated the maximum speedup in such a problem to be 1.75, independent of the size of the matrix. Peters recognized that these speedups were greatly reduced by the essentially sequential solution of the dense portion of the matrix remaining after most of the matrix had been solved. Basically a series of problems in which D consisted of only one element were being solved, and thus only one processor used at time. He, however, did not recognize the possibility of switching to a dense system solver to prevent this.

We now turn our attention to process flowsheeting matrices and to how we can use the structure of the problem to reorder the rows and columns prior to the simple search procedure for the BoDF. In considering initial reorderings, we recall that finding the largest or nearly largest diagonal part will not necessarily reduce the number of overall steps required (Peters, 1985). Instead, like Conrad and Wallach (1977), we start with orderings that will tend to reduce the amount of fill-in in T, the rationale being that in the second step a larger diagonal part will be found. Reordering algorithms for doing this have been discussed in detail by Stadtherr and Wood (1984a). We employ here two algorithms they used: SPK1, which is for general-purpose problems, and BLOKS, which is specifically for matrices with a block structure, like flowsheeting problems.

We consider the following initial reordering techniques in the order of increasing complexity:

1. *Natural Ordering.* Most EB simulators generate the equations a unit at a time and the variables a stream at a time. This leads to a natural block structure, which often has some desirable properties with regard to fill-in even without further reordering. So in this case we take the initial reordering to be that produced by the equation generation routine in the simulator without any additional reordering.
2. *Block-Only Ordering.* In this case, we take the matrix from the natural ordering, define blocks according to the method of Stadtherr and Wood (1984a), and reorder the block matrix using SPK1. No attempt is made to reorder individual equations and variables.
3. *BLOKS Ordering.* In this case, we apply the BLOKS algorithm to the original matrix. Essentially this amounts to taking the block-only ordering from above and then reordering the equations and variables within the blocks.
4. *BLOKS/SPK1 Ordering.* In the previous three cases, the matrix was reordered only prior to finding the BoDF in the first step. In this case, we reorder the matrix prior to searching for the BoDF in all steps. Prior to the first step, we apply BLOKS. Since during the first step, the block structure of the matrix will be destroyed, we apply SPK1 prior to the all subsequent steps.

Several examples from Wood (1982) have been considered. These are based on flowsheet structures taken from Sood and Reklaitis (1975) and Motard and Westerberg (1981), and involve mixers, splitter, heaters, and ideal and nonideal equilibrium-stage processes. Tables 1-4 show some typical results for each of the above initial orderings for using the BoDF structure for solving flowsheeting matrices in parallel. It can be seen that in all cases a good BoDF can be found. After two steps, 61-70% of the matrix has been pivoted through in parallel, with the BLOKS reordering getting the furthest. After five steps, both BLOKS and BLOKS/SPK1 have pivoted through 77% of the matrix.

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 denser at each succeeding step. As the lower righthand block gets smaller and fills up,

Table 2. Parallel Pivoting on Flowsheeting Matrices in BoDF With Block-Only Reordering ($N = 372$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	193	0.14
2	51	0.31
3	12	0.45
4	2	0.46
5	2	0.48
6	2	0.62
7-17	1	0.63-0.70
18	18	0.80
19	3	0.82
20-35 etc.	1	

Table 4. Parallel Pivoting on Flowsheeting Matrices in BoDF With BLOKS/SPK1 Reordering ($N = 372$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	181	0.09
2	47	0.25
3	2	0.27
4	1	0.27
5	55	0.76
6-23	1	0.77-0.86
24	5	0.88
25-68	1	
etc.		

it is necessary to switch to a full matrix solver. Just when this switch should occur will depend on the size and density of the remaining block, and will vary from problem to problem and machine to machine. It will also depend on the efficiency of the full matrix solver being used. For the four cases here, switches after anywhere from steps 2 to 5 appear to be appropriate.

In comparing initial ordering methods, simple orderings (natural and block-only) perform about the same as the slightly more expensive BLOKS and BLOKS/SPK1 orderings, though after five steps the latter have made it through about 10% more of the matrix. Therefore, as long as one takes advantage of the problem structure in the equation generator by generating unit-stream blocks, it would appear that good results can be obtained. The reason for this is that when the equations are generated in such blocks they generally contain some equations that easily form diagonal submatrices.

The speedup of the sparse portion of the procedure is shown in Table 5 for the BLOKS/SPK1 case. The results presented are cumulative over the given number of steps; for example, the results for step 3 indicate the overall speedup in the first three steps. These numbers are actually quite encouraging, especially if we use Amdahl's law to convert them to percentage parallelized (f). After four steps, f remains as high as 87% for 16 processors and as high as 82% for eight processors. At this range of f, a small change in the percent parallelization can produce a large change in speedup. Thus, the slightly better performance of the BLOKS and BLOKS/SPK1 initial ordering may be important, and even small future gains in f will be significant.

Table 5. Cumulative Speedup in the Parallel Solution of Flowsheeting Matrices in BoDF ($N = 372$)

	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
6	1.6	2.1	2.6	3.1	3.7
7	1.5	1.9	2.3	2.6	3.0
.	.	.	.	.	.
.	.	.	.	.	.
25	1.1	1.1	1.1	1.2	1.2

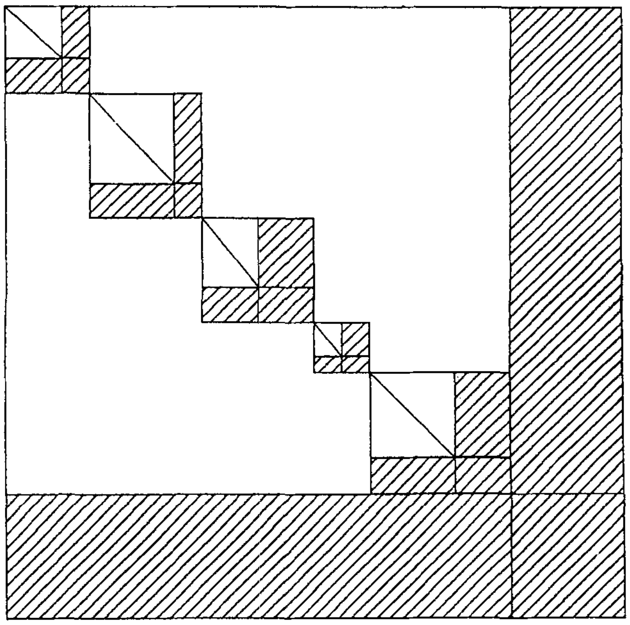
Figure 4. Matrix in bordered block diagonal form (BoBDF) with BoDF within the diagonal blocks.

The procedure using BoDF has two shortcomings, however. First, it does not take direct advantage of the block structure of the flowsheeting matrices. Also, each step includes a search for a diagonal submatrix. This search is not done in parallel. This sequential portion may greatly slow down the overall computation rate, although the search needs to be done only once at the beginning of the solution procedure. To avoid these problems, a bordered block diagonal form (BoBDF) can be used.

Bordered block diagonal form

A matrix BoBDF can be solved in a method analogous to the solution of a matrix in BoDF. Instead of each processor working on a diagonal element, each processor works on a diagonal 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 4). This allows pivots from the diagonal parts of all the blocks to be performed at the same time. So again, we are assigning processors to pivot elements rather than blocks. Also, it is possible to do the searches for the diagonal submatrices in parallel since each diagonal block can be considered independently.

To do this, we again define a block matrix according to the method of Stadtherr and Wood (1984a). It should be noted that the blocks in this matrix do not correspond to entire units in all cases, but to groupings of equations and variables from a unit (for example, equilibrium stages). A SPK1 reordering is applied to the block matrix initially, and then the diagonal part of the block matrix is searched for as above. At each step in the method then, these diagonal blocks are themselves

Table 6. Parallel Pivoting on Flowsheeting Matrices in BoBDF With Block-Only Reordering ($N = 372$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	120	0.06
2	5	0.06
3	4	0.06
4	4	0.06
5	4	0.06
6	4	0.06
7	4	0.06
8–16	4	0.05–0.06
etc.		

Table 8. Cumulative Speedup in the Parallel Solution of Flowsheeting Matrices in BoBDF ($N = 372$)

	Processors				
Step	2	4	8	16	∞
1	1.4	1.6	1.8	1.9	3.9
2	1.5	1.8	2.1	2.1	3.3
3	1.4	2.0	2.2	2.3	3.1
4	1.4	2.1	2.3	2.3	3.0
5	1.4	2.2	2.3	2.4	2.9
.	.	.	.	.	.
.	.	.	.	.	.
15	1.5	2.5	2.6	2.6	2.9
16	1.5	2.5	2.6	2.6	2.9

searched for diagonal submatrices, which as noted above can be done in parallel. The elements taken from the diagonal part of each block are then used as parallel pivots.

Tables 6 and 7 show typical results of solving flowsheeting matrices in BoBDF in parallel, for two problems taken from Wood (1982). As can be seen in the tables, the BoBDF results in much more regular results than for BoDF. The initial search for diagonals within the blocks turns up a large number of elements but less than in BoDF. This is because there are some elements in the border in BoBDF that would become diagonal elements in BoDF. What tends to happen on the first step is that some blocks, such as those resulting from mixers, splitters, and the mass balance blocks of other units, are completely eliminated, and that the remaining portions of the other diagonal blocks generally become full. This means that after the first step, there will generally be only one element in the diagonal part of each block, and thus only one pivot will be used from each block. This results in the nearly constant number of pivots that are available in subsequent steps. It also means that the number of parallel pivots is roughly proportional to the size of the problem, which implies that this method should work well on large problems. The selection of one pivot from each diagonal block generally continues until the block diagonal part has been eliminated. At that point, the remaining lower righthand submatrix of the BoBDF can be solved by a parallel full solver.

The cumulative speedup of the sparse portion of the parallel solution using BoBDF is shown in Tables 8 and 9. Unlike BoDF, the speedup with BoBDF remains fairly constant with each step. This is a result of the relatively constant number of parallel pivots in each step. For the smaller matrices, BoBDF does not provide as much speedup. Because the speedup depends on the number of pivots found in each parallel step, however, the BoBDF works very well for the larger flowsheet. In fact, for this problem the percentage parallelization is 95–97% depending on the number of processors.

Concluding Remarks

The equation-based approach to process flowsheeting appears to offer the most promise for the effective use of parallel processing, at least if the sparse matrix part of the computation can be effectively parallelized. The BoDF and BoBDF represent two matrix decompositions that can be used to provide parallelism in the solution of flowsheeting matrices. The former uses the structure of the flowsheeting problem indirectly in the initial ordering used to generate the BoDF and appears best suited to relatively small problems. The latter uses the structure of the flowsheeting problem directly in forming the blocks of the BoBDF and appears best suited to relatively large problems. For the largest problem considered here, which is not especially large by flowsheeting standards, a percentage parallelization of over 95% was achieved using BoBDF. At this level of parallelization, even small future improvements will have a significant impact on the speedup.

Acknowledgment

This work was supported in part by the National Science Foundation under Grant number DMC-8520663 and DDM-9024946. We also acknowledge the support of the National Center for Supercomputing Applications at the University of Illinois.

Table 7. Parallel Pivoting on Flowsheeting Matrices in BoBDF With Block-Only Reordering ($N = 814$)

Step	No. of Parallel Pivots	Density of Remaining Matrix
1	220	0.02
2	11	0.02
3	11	0.02
4	11	0.02
5	11	0.03
6	11	0.03
7	11	0.03
8	11	0.03
9	11	0.03
10–16	11	0.03–0.04

Table 9. Cumulative Speedup in the Parallel Solution of Flowsheeting Matrices in BoBDF ($N = 1060$)

	Processors				
Step	2	4	8	16	∞
1	1.7	3.2	5.7	9.2	62.8
2	1.8	3.4	6.4	10.4	53.7
3	1.9	3.5	6.6	10.9	49.7
4	1.9	3.6	6.8	11.1	47.2
5	1.9	3.6	6.8	11.2	45.4
6	1.9	3.6	6.8	11.2	45.2

Literature Cited

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

Bertsekas, D. P., "Distributed Asynchronous Computation of Fixed Points," *Math. Programming*, **27**, 107 (1983).

Biegler, L. T., "Improved Infeasible Path Optimization for Sequential Modular Simulators: I. The Interface," *Comput. Chem. Eng.*, **9**, 245 (1985).

Calahan, D. A., "Parallel Solution of Sparse Simultaneous Linear Equations," *Proc. of Allerton Conf. on Circuits and System Theory* (1973).

Chen, H. S., and M. A. Stadtherr, "A Simultaneous Modular Approach to Process Flowsheeting and Optimization: I. Theory and Implementation," *AIChE J.*, **31**, 1843 (1985).

Chimowitz, E. H., and R. Z. Bielinis, "Analysis of Parallelism in Modular Flowsheet Calculations," *AIChE J.*, **33**, 976 (1987).

Conrad, V., and Y. Wallach, "Parallel Optimally Ordered Factorization," *Proc. Power Industry Applications Conf.* (1977).

Coon, A. B., and M. A. Stadtherr, "Parallel Implementation of Sparse LU Decomposition for Chemical Engineering Applications," *Comput. Chem. Eng.*, **13**, 899 (1989).

Crowther, W., J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor," *Proc. of Int. Conf. on Parallel Processing*, IEEE Computer Soc. (1985).

Dennis, J. B., "Data Flow Supercomputers," *Computer*, **13**(11), 48 (1980).

Dongarra, J. J., "Performance of Various Computers Using Standard Linear Equations Software," Report CS-89-85, Computer Science Dept., Univ. of Tennessee, Knoxville, (Mar. 11, 1992). (A Postscipt copy of this report can be obtained on-line from NETLIB by sending the message *send performance from benchmark* to netlib@ornl.gov.)

Duff, I. S., "A Survey of Sparse Matrix Software," *Sources and Development of Mathematical Software*, p. 165, W. R. Cowell, ed., Prentice-Hall, Englewood Cliffs, NJ (1984).

Faber, V., O. M. Lubeck, and A. B. White, Jr., "Superlinear Speedup of an Efficient Algorithm Is Not Possible," *Parallel Computing*, **3**, 259 (1986).

Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, **C-32**(2), 175 (1983).

Gustafson, J. L., "Reevaluating Amdahl's Law," *Comm. ACM*, **31**, 532 (1988).

Haley, J. C., and P. V. L. N. Sarma, "Process Simulators on Supercomputers," *Chem. Eng. Prog.*, **85**(10), 28 (1989).

Hwang, K., *Supercomputers: Design and Applications*, IEEE Computer Soc. (1984).

Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York (1984).

Jirapongphan, S., J. F. Boston, H. I. Britt, and L. B. Evans, "A Nonlinear Simultaneous Modular Algorithm for Process Flowsheet Optimization," AIChE Meeting, Chicago (1980).

Karp, R. M., "Reducibility among Combinatorial Problems," in *Complexity of Computer Communications*, R. E. Miller and D. W. Thatcher, eds., Plenum Press (1972).

Kuhn, R. H., and D. A. Padua, *Tutorial on Parallel Processing*, IEEE Computer Soc. (1981).

La Roche, R. D., Y.-T. Chen, and D. Subramanian, "Parallel Processing Strategies for Simultaneous Modular Flowsheeting," AIChE Meeting, Washington, DC (Nov., 1988).

Levesque, J. M., "Effective Utilization of Parallel Vector Processors," AIChE Meeting, Miami Beach (1986).

McRae, G. J., "Chemical Process Modeling and Simulation Using Advanced Computer Architectures," *Foundations of Computer-Aided Process Design*, J. J. Siirola, I. E. Grossman, and G. Stephanopoulos, eds., CACHE (Elsevier) (1990).

McRae, G. J., J. B. Milford, and B. J. Slompak, "Changing Roles for Supercomputing in Chemical Engineering," *Int. J. Supercomputer Appl.*, **2**(2), 16 (1988).

Motard, R. L., and A. W. Westerberg, "Exclusive Tear Sets for Flowsheets," *AIChE J.*, **27**, 725 (1981).

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

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

Peters, F. J., "Parallelism and Sparse Linear Equations," *Sparsity & Its Applications*, Cambridge Univ. Press (1985).

Sood, M. K., and G. V. Reklaitis, *User's Manual: Material Balance Program*, Vol. II, 2nd ed., School of Chemical Engineering, Purdue Univ., Lafayette, IN (1975).

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 E. S. Wood, "Sparse Matrix Methods for Equation-based Chemical Process Flowsheeting: I. Reordering Phase," *Comput. Chem. Eng.*, **8**, 9 (1984a).

Stadtherr, M. A., and E. S. Wood, "Sparse Matrix Methods for Equation-based Chemical Process Flowsheeting: II. Numerical Phase," *Comput. Chem. Eng.*, **8**, 19 (1984b).

Trevino-Lozano, R. A., L. B. Evans, H. I. Britt, and J. F. Boston, "Simultaneous Modular Process Simulation and Optimization," *IChemE Symp. Ser.*, **92**, 25 (1985).

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 Chemical Process Flowsheeting," *AIChE J.*, **36**, 1687 (1990).

Wood, E. S., "Two-pass Strategies for Sparse Matrix Computation in Chemical Process Flowsheeting Problems," PhD Thesis, Univ. of Illinois, Urbana (1982).

Zitney, S. E., and M. A. Stadtherr, "A Frontal Algorithm for Equation-Based Chemical Process Flowsheeting on Vector and Parallel Computers," AIChE Meeting, Washington, DC (1988).

Manuscript received Feb. 3, 1992, and revision received May 1, 1992.