Vector Processing Strategies for Chemical Process Flowsheeting

The potential for effective vectorization of the sequential-modular, simultaneous-modular, and equation-based approaches to process flowsheeting is assessed. The last appears to be the most promising, but the required sparse matrix techniques currently used do not vectorize well. Different approaches to vectorizing the sparse matrix calculation are considered. With a good matrix reordering scheme, the most promising sparse matrix strategy for the vectorization of flowsheeting problems is the frontal method, a technique currently used primarily on finite-element problems.

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

Introduction

Over the past several years a new generation of computer technology has emerged. New computer architectures, especially those involving vector processing and multiprocessing (parallel processing), offer tremendous increases in computational power relative to conventional machines. An introduction to such advanced computer architectures has been given by Vegeais et al. (1986). Stadtherr and Vegeais (1985), and more recently Elkin (1988) and McRae et al. (1988), have discussed some of the current and future applications of these architectures in chemical engineering.

The application of interest here is chemical process flowsheeting. For the design engineer using a flowsheeting program, the ideal situation would be to obtain an essentially instantaneous response from the program. That is, the engineer could make a design change and see the implications almost immediately, not several minutes or hours later, as is still too often the case today. A true interaction between man and machine will not only mean increased design productivity, implying that since more designs can be considered a better one is more likely to be found, but also an enhancement of the design thought process itself, since it is no longer interrupted by long waits for computational answers.

Today, flowsheeting programs can be run on machines ranging from supercomputers to workstations or PCs. The latter are quite capable of handling the initial phases of a new process design, but as the level of process complexity and integration grows we need to rely on increasingly more powerful machines. What we envision is that each design engineer will have a workstation connected by a network to a larger and much faster advanced-architecture machine. The workstation would handle as much computation as possible locally and, transparently to the user, send larger jobs to the faster machine. Such configurations of workstations and supercomputers are possible now. It is also worth noting that the workstations themselves may today be based on advanced computer architectures.

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

The use of advanced computer architectures, namely vector processing and multiprocessing architectures, has the potential to provide very significant advances in the area of process flowsheeting. However, to realize such benefits we must be able to use problem formulations, solution algorithms, and computer codes that effectively exploit these architectures. In some cases, computer codes can be changed to take better advantage of vectorization or parallelization, and tools for helping the programmer do this are becoming increasingly better. But in other cases, and we believe process flowsheeting is one, code modifications will not help significantly, since the basic problem formulation and solution algorithms used are not generally amenable to vectorization or parallelization. This is the problem that needs to be addressed.

Correspondence concerning this paper should be addressed to M. A. Stadtherr.
The current address of J. A. Vegeais is: Shell Development Co., Westhollow Research Center, Houston, TX 77082-3101.

In this paper we consider the use of vector processing architectures in process flowsheeting. We assess the potential for vectorization of the three standard formulations for the flowsheeting problem: sequential modular (SeqM), simultaneous modular (SimM), and equation based (EB). We conclude that the last is most amenable to vector processing, but that to realize the potential of the EB approach in this context we need sparse matrix algorithms that exploit the vector architecture well. We discuss this problem here in some detail and present a promising approach to its solution.

Background

Much basic information about advanced computer architectures is available elsewhere (Kuhn and Padua, 1981; Hwang, 1984; Hwang and Briggs, 1984; Vegeais et al., 1986). Thus, there is no discussion along these lines here. We do briefly discuss measures of algorithm performance on vector machines, since this plays a part in the interpretation of the results.

The performance of an algorithm or code on a vector processor is often quantified in terms of its percentage vectorization or its speedup. Speedup is defined here as the ratio of the time it takes for a job to execute without vectorization to the time it takes for that job to execute with vectorization on the same computer.

An important relationship between percentage vectorization 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 vectorization. Amdahl's law has the form $S = V/[V(1-f)+f]$, where S represents an upper bound on the speedup, f is the fraction of utilized code vectorized, and V is the ratio of the peak vector processor speed to the peak scalar processor speed on a given machine. Ideally then, if a code were completely vectorized ($f = 1$), we would get a speedup of V. In practice the speedup would be less, due to factors such as the time required to start a vector operation.

Figure 1 shows a plot of speedup vs. the fraction of utilized code that is vectorizable, for some values of V. It can be seen that significant speedup requires that a very large fraction of the code be vectorized. Even a very small amount of nonvectorized code can result in a significant loss of performance. Another way to look at this, however, is that once a relatively high level of vectorization has been achieved, even small improvements in the amount vectorized will yield large improvements in performance. In practice, the size of such improvements is limited by the fact, mentioned above, that the peak speedup V can rarely be obtained. This is especially true for short vector lengths, in which case vector start-up time becomes proportionately more important.

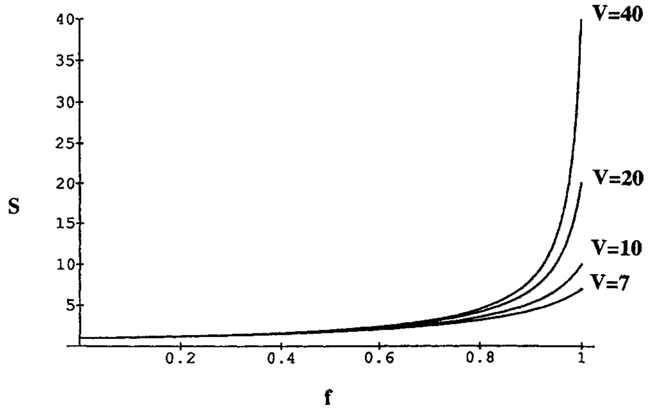

Figure 1. Plot of Amdahl's law.

In addition, another effect has been noted by Gustafson (1988) with regards to Amdahl's law. Gustafson points out that the law assumes a fixed problem size, but that for most applications f actually increases with problem size. That is, as larger and larger problems are attacked, the speedup tends to increase.

Effect of Problem Formulation

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

Sequential-modular approach

Several years ago, Duerre and Bumb (1981) reported installing and running the public domain version of ASPEN, a SeqM simulator, on a Cray-1 computer. They found that the program ran only two to three times faster than on an IBM 370. Since one would expect performance improvements on this order simply due to the faster scalar speed of the Cray-1, this indicates that little advantage was taken of the vector architecture. More recently, Haley and Sarma (1989) studied the performance of two SeqM programs, PROCESS, a commercial package, and CPES, DuPont's in-house simulator, on a Cray X-MP and a VAX 8800 computer, using thirty-four industrial problems. They found that the Cray X-MP ran roughly an order of magnitude faster than the VAX 8800. Again the increase in computational speed is not greatly beyond that expected on the basis of scalar speed. Some vectorization occurs, but not enough to make a truly significant impact. It should be noted, however, that the performance improvement due simply to scalar speed is, in this case, quite impressive. In fact, Haley and Sarma report that, at least under DuPont's computing cost structure, runs on the Cray X-MP cost one-half to one-third of those on the VAX 8800, both on complex simulations and routine day-to-day problems. Better results have been reported by Harrison (1989) and Zitney (1990). In the last case, the frontal techniques described in this paper were used to improve vectorization in some modules.

In the first two studies referred to above no significant effort was made to optimize the code for vector processing; in the second two some such efforts were made. However, it is unclear to what extent such an effort would improve performance of SeqM codes in general. In fact, it seems likely that any attempts to vectorize a SeqM package will be at most only partially successful. A major reason is that the run time of a SeqM flowsheeting program tends to be divided among many modules, some for unit operations, some for physical properties, and most of which use different and special-purpose equation solving methods. Keeping Amdahl's law in mind, it would be necessary to significantly vectorize these many different subroutines to obtain any really meaningful speedup. In some cases this might be done by optimizing the existing code, in other cases it may be necessary to change the code entirely, perhaps employing a different algorithm, and in some cases there may be little or no vectorization possible. If significant vectorization is not possible in even just a few important modules, there could be rather little overall speedup. In some simulations, however, the run time is concentrated in a rather small number of modules. Typically these would be physical property modules and rigorous separation modules. By focusing vectorization efforts on such modules it is possible that, for a good number of problems, a fair speedup might be attained. This has been confirmed by Zitney (1990). Unfortunately, such an approach will not help speed up the simulation of flowsheets in general—only those requiring that a large percentage of time be spent in the modules that can be vectorized.

Simultaneous-modular approach

The SimM approach to process flowsheeting uses the same library of modules as in the SeqM case. Again the large majority of the run time is divided among these many modules, and the comments made above about the SeqM approach still apply. While there are some additional types of calculations involved, many of which involve linear algebra and may be amenable to vectorization, these do not account for a large fraction of the run time. From the standpoint of vector processing, there is no significant difference between the potential of the SimM and SeqM approaches; neither is especially attractive.

Equation-based approach

In the EB approach to process flowsheeting, all of the equations describing the system are gathered together and solved as one very large set of nonlinear equations, normally by the Newton-Raphson method, a quasi-Newton method, or some related technique. Physical property models may be included directly in this equation set, or may be handled indirectly as subroutines called when function and derivative evaluations are made. The latter approach appears to be the most common today.

In the EB approach, unlike the previous two approaches, most of the computation time is spent in just a few key sections. These major parts are the evaluation of functions, the evaluation or approximation of the Jacobian matrix, and the solution of a large, sparse, linear equation system. Effective vectorization of the equation-based approach involves the vectorization of each of these subproblems.

In function evaluation, the number of different equation types involved is fairly limited. Thus, one essentially has an equation library (Stadtherr and Hilton, 1982b), instead of the module library found in a SeqM or SimM code. While vectorizing a module may be difficult and time consuming, perhaps requiring the development and use of a new algorithm, if the evaluation of an equation can be vectorized, it should be readily apparent and not especially difficult to do. When physical properties are handled indirectly using subroutines, rather than directly as equations, the situation is somewhat different. In this case, the physical property routines are nested within the function evaluations, and will typically dominate the function evaluation time. Again, as in the SeqM and SimM cases, vectorization of the physical property routines is extremely important. Fortunately, some physical property models do appear to be amenable to vectorization (Seader, 1985; Zitney, 1990).

Jacobian evaluation may be done explicitly or by finite difference. In either case the considerations are the same as for function evaluation. If done explicitly, there will be a limited number of derivative equation types, facilitating vectorization when possible. If done by finite difference, the work primarily involves doing more function evaluations. Again physical property calculations may dominate. If the Newton-Raphson method is used to converge the problem, requiring a Jacobian evaluation every iteration, the Jacobian evaluation time will be comparable to or exceed the function evaluation time. On the other hand, if a quasi-Newton method is used, Jacobian evaluations are not required every iteration, and the Jacobian approximations will require relatively little time.

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, or positive definiteness, that are often found in matrices arising from the discretization of partial differential equation problems, and that are exploitable in developing an efficient solution method. Thus, in EB flowsheeting the sparse matrix problem is usually solved using a direct, general-purpose method employing routine Gaussian elimination or some variation thereof. Typically, the Harwell subroutine MA28 (Duff and Reid, 1979) or something similar is used. Some alternative routines (Stadtherr and Wood, 1984a, b; Chen and Stadtherr, 1984) have also been proposed, one of which, LU1SOL, is now used in an industrial EB simulator, and significantly outperforms MA28. It is also worth noting that LU1SOL was recently rated in an independent study (Kaijaluoto et al., 1989) as the best available for EB flowsheeting on conventional machines. On some flowsheeting problems, the time required to solve the sparse matrix problem is the largest component of the overall solution time. On others, especially if the Newton-Raphson method is used, the function and Jacobian evaluation time, including physical property calculations, may be larger. In any case, the sparse matrix problem represents a key step in EB process flowsheeting.

Unfortunately, the direct solution of large, sparse sets of linear equations with no regular structure is a problem that has not proved particularly amenable to vector processing. In order to eliminate the storage of the zeros, codes such as MA28 or LU1SOL store the matrix not as a full two-dimensional array, but as three one-dimensional arrays containing the nonzero coefficient values and information about their row and column indices. This storage scheme requires the use of indirect addressing in the sparse matrix solver. Experience has been that general-purpose sparse matrix codes do not vectorize well due to this use of indirect addressing. For example, Duff and Reid (1982) found that a general-purpose full matrix routine executed on a Cray-1 was roughly twenty to thirty times faster than the same routine executed on an IBM 3033. On the other hand, when the same comparison was made using the MA28 general-purpose sparse matrix routine, it was found to be only about two times as fast on the Cray-1. Since the scalar speed of the Cray-1 is roughly twice that of the IBM 3033, it is apparent that little vectorization was possible. As Duff and Reid conclude, we need to "rethink our sparse matrix algorithms."

In the remainder of the paper, we concentrate on this problem, and attempt to develop strategies for effectively using vector processing computer architectures in solving the sparse matrix problems that arise in equation-based flowsheeting. It should be emphasized, however, that, although the solution of the sparse matrix problem is a key computational step in equation-based flowsheeting, there are other computationally intensive steps as well, as discussed above. Although we do not consider the function and Jacobian evaluation steps further here, their vectorization is also very important.

Flowsheeting Matrix Structure

Although flowsheeting matrices do not have a highly regular structure, they do have some natural structure arising from the unit-stream nature of a process. Since we intend to try to exploit this structure in using vector processing, this structure is discussed briefly here.

In an equation-based flowsheeting package, the equations are normally generated unit by unit, and the variables stream by stream (Perkins and Sargent, 1982; Stadtherr and Hilton, 1982b). That is, first, all the equations describing the first unit and its interconnecting streams are generated, then the equations describing the second unit and its streams are generated, and so on. When generated this way, it can easily be seen that the nonzeros in the matrix will fall into blocks corresponding to units (or subunits) and streams.

Stadtherr and Wood (1984a) took advantage of this block structure in their reordering scheme, BLOKS. In their matrix formulation, one block "variable" was created for each interconnecting stream and one block "equation" was created for each output stream. Note that although there is one block created for each interconnecting stream, the variables within those blocks are not limited to variables describing the interconnecting streams. Variables internal to the unit are also in these blocks. For this reason, not all blocks are of the same size and not all blocks are square. Also, it should be noted that the flowsheet structure results in a block matrix that has more columns than rows. The rows that are required to make the block matrix square correspond to the design specifications that must be made to fully specify the problem.

As an example, consider the flowsheet of the Cavett problem (1963), a network of flashes (units 2, 3, 5, and 6) and mixers (units 1 and 4) shown in Figure 2. If the units and streams are numbered as shown, the block matrix structure given in Figure 3 results. Note that there are eleven block variables and only ten block equations. The eleventh block equation would comprise all of the design specifications. Note also that this block representation does not depend on the exact equation formulation. That is, this matrix does not depend on the variables and equations used to describe the units in the process, only on the structure of the flowsheet itself.

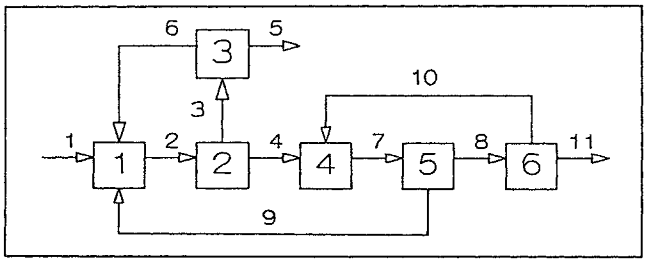

Figure 2. Cavett problem flowsheet.

$$
\begin{array}{c|ccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & T & E \\
1 & \times & \times & & & & \times & & & \times & & \\
2 & & \times & \times & \times & & & & & & & \\
 & & & \times & \times & & & & & & & \\
3 & & & \times & & \times & \times & & & & & \\
 & & & & & \times & \times & & & & & \\
4 & & & & \times & & & \times & & & \times & \\
5 & & & & & & & \times & \times & \times & & \\
 & & & & & & & & \times & \times & & \\
6 & & & & & & & & \times & & \times & \times \\
 & & & & & & & & & & \times & \times
\end{array}
$$

Figure 3. Block matrix for Cavett problem.

This example matrix is fairly typical of the block structure of flowsheeting matrices. If adjacent units and streams are numbered more or less consecutively whenever possible, the resulting matrices generally are nearly block-banded with some off-band blocks scattered throughout the matrix. These off-band blocks can be thought of as representing recycle or feedforward streams. A process where the output of one unit becomes the input of the next, such as a series of reactors, will have a block bidiagonal form. If a process consists of only equilibrium stages where the exit streams of a unit are connected to the units immediately preceding and following that unit, such as in distillation columns, the resulting matrix will have a block tridiagonal form. In general, the bandwidth of the matrix varies, however. Blocks of zeros can occur on the diagonal or elsewhere in the band.

Each of the blocks described above is, in fact, generally another sparse matrix. The sparsity of these blocks, however, is much less than that of the matrix in general. The structure within the blocks depends on the type of unit being described and the model used to describe it.

In the discussion above, we used the method suggested by Stadtherr and Wood (1984a) to define the matrix blocks. It should be noted that there are other useful ways to define the blocks. For instance, Stadtherr and Hilton (1982a) define the blocks so that a bordered-block-triangular form results, and Westerberg and Berna (1978) so that a bordered-block-diagonal form results. Both have computationally attractive features.

Vector Processing Strategies

On vector computers, it is desirable to operate on data in contiguously or regularly indexed vectors. General-purpose sparse matrix algorithms make use of indirect addressing in order to reduce storage requirements and the number of operations performed. This means that the data are not stored in contiguously or regularly indexed vectors. Two ways to make a sparse matrix algorithm operate on contiguously or regularly indexed vectors are the use of gather and scatter and the treatment of relatively dense parts of the sparse matrix as if they were full submatrices.

Gather/scatter

In the gather/scatter method, the needed indirectly addressed data are gathered into contiguous locations. Vector operations are then performed on these data. Finally, the results are scattered back into their original indirectly addressed locations. Most vector computers have special gather/scatter software or (usually) hardware.

Early experience was that gather/scatter software took little advantage of the possible speedup of vector processors (Dembart and Neves, 1977; Petersen, 1983). However, today most vector machines have hardware gather/scatter, which allows the gather and scatter to be performed as vector memory operations, and significantly improves performance. Lewis and Simon (1988) have compared the relative execution times for the factorization of seven sparse matrix test problems on a Cray X-MP/24 with and without hardware gather/scatter. The test matrices chosen were all symmetric, positive definite. Six of the seven matrices were from finite-element problems while the seventh arose from an electric power network problem. For the six finite-element problems, the use of hardware gather/scatter resulted in consistently good speedups of from roughly 5.5 to 7.8, compared to the runs without hardware gather/scatter. On the power network problem, however, only a small speedup (1.34) was observed for hardware gather/scatter. At least part of the reason for this is that the power network problem is more sparse than the other problems. This results in shorter vectors and, therefore, a smaller speedup. While these results look generally encouraging, the structure of flowsheeting matrices is more similar to the power network matrices than to the others. In fact, recent experiments by Zitney and Stadtherr (1988b), have shown that on flowsheeting problems, hardware gather/scatter provides virtually no improvement in performance.

Block-oriented approach

One way to improve the performance of a vector processor in solving a flowsheeting matrix is to treat parts of the matrix as if they were full, since operations done on the full submatrices will vectorize. In order to do this, some matrix structure must be identified. We have already seen one type of structure in flowsheeting matrices, namely their natural block-oriented structure, that can be used for this purpose.

Calahan (1979, 1980) first proposed the block-oriented approach as a means of vectorizing a sparse matrix code. The general idea is to keep the blocks intact and store them as full matrices. The system is then solved using block Gaussian elimination. Since the blocks are now stored in a regularly indexed fashion, the operations performed on the blocks will be vector operations. In Calahan's implementation, the number of rows in a block above the diagonal and the number of columns in a block below the diagonal are fixed by the size of the diagonal blocks. The location and size of each block are described by arrays that hold pointers to the start of each block, the number of columns in each block above the diagonal, the number of rows in each block below the diagonal, and by an array that links each block to the diagonal block in the same group of rows or columns. The actual generation of blocks has to be done by the user and the user has to anticipate the location of all possible fill-in and create blocks to contain this fill-in. Although Calahan's code can be applied to any block matrix, its implementation is described on only banded and block tridiagonal matrices. Flowsheeting matrices would have to be first reordered to ensure the presence of a diagonal block. The code of Calahan makes no provision for pivoting to maintain numerical stability.

One difficulty with this approach is that the blocks that occur in flowsheeting matrices are themselves usually quite sparse. Treating them as full can add very substantially to the storage requirement and, more importantly perhaps, will result in performing many wasted operations on zero elements. We essentially have a situation where there is a tradeoff between computational rate and number of operations required.

A second and more serious difficulty with the block-oriented approach is that flowsheeting matrices are generally not diagonally dominant. Thus, when applying this approach to flowsheeting matrices it is often necessary to perform row or column interchanges in order to preserve numerical stability. These interchanges require the redefinition of the blocks, which involves searching through the blocks and transferring columns from block to block. These interchanges may also result in the creation of extra blocks of fill-in. Since there appears to be no efficient way to maintain numerical stability, the block-oriented approach does not appear to be particularly useful for solving process flowsheeting matrices on vector computers.

Frontal approach

Another type of matrix structure that can be used to limit calculations to a set of full submatrices is the banded structure. A technique used with this matrix structure is known as the frontal approach. As seen previously, flowsheeting matrices are roughly block-banded, though with some off-band blocks.

The frontal approach originated as a band or profile solver for finite-element problems. Jennings (1966) first developed the frontal approach to solve symmetric systems of linear equations. Hood (1976) extended the method to unsymmetric matrices. The motivation of Jennings and Hood was to solve large matrices in limited memory and did not involve vectorization. The potential of the frontal method for vectorization was apparently first recognized by Duff (1979). In this method, operations are confined to a relatively small submatrix, called a frontal matrix, that can be stored as full. This method takes advantage of the fact that each variable appears in only a few equations and that pivoting on a variable will affect only a small number of equations and variables. Its two main advantages are that the inner loop of the code is vectorizable and that the solution of the matrix may be done in a small amount of memory.

The basic frontal algorithm, as applied to individual equations and variables rather than finite elements, is very simple. Equations (rows) enter the frontal matrix one at a time in numerical order. Variables (columns) enter the frontal matrix if they occur in the entering row. After each row enters, it is determined whether there are any variables whose occurrences are completely within the frontal matrix. If there are any such variables, they can be pivoted on and eliminated. The eliminated variables and the corresponding pivot rows can then be removed from the frontal matrix. Then the next row enters the frontal matrix and the procedure is repeated.

Perhaps the best known code for implementing the frontal method is the Harwell code MA32 (Duff and Reid, 1982; Duff, 1984a). This code is generally structured for input by finite element and is less efficient when used in an equation-by-equation manner, as is needed for solving flowsheeting matrices. Duff and Reid (1984) have also developed a multifrontal code, MA37. The multifrontal code, as its name implies, divides the matrix so that more than one frontal matrix is employed. It also is designed with finite-element problems in mind.

The major advantage of this method, at least for banded systems, is that the frontal matrix stays fairly small and dense throughout the solution process, and thus can readily be stored and operated on as a full matrix. This allows the use of vector operations during elimination. Another advantage of this method is that the amount of storage necessary for the frontal matrix and other needed arrays is small. Thus an out-of-core solution procedure is possible in which the only part of the matrix that needs to be in core is the frontal matrix. This was the original motivation for the development of the frontal method. Of course, if sufficient memory is available, it will be more efficient to keep the entire matrix in core.

Flowsheeting matrices have a roughly banded structure, but the band is relatively sparse, and there are generally off-band blocks. This suggests that the frontal matrix will be neither small nor dense. We can still store and operate on the frontal matrix as if it were full, thus allowing vectorization. However, as the size of the frontal matrix grows the operation count will grow proportionately; and, as the frontal matrix becomes sparser, there will be more and more wasted operations on zeros. So again we face the tradeoff between computation rate and operation count. A key question to be considered here is whether the frontal matrix can be kept small enough on flowsheeting problems to provide a significant performance enhancement on a vector machine. The size of the frontal matrix depends largely on the ordering of rows in the original matrix.

As emphasized above, the frontal method has been used almost exclusively in connection with the solution of partial differential equation problems by discretization, primarily by the finite-element method. Prior to this work, the only attempt to apply the frontal method in process flowsheeting was by Cameron (1977), and was not in the context of vectorization.

Frontal matrix size

In solving sparse matrices on sequential computers, it is very important to keep the number of operations performed small. On vector computers, however, it has been shown (Dembart and Neves, 1977; Pottle, 1979) that at times it is advantageous to perform unnecessary operations in order to increase the length of vectors being operated on and, therefore, the rate of operations. This is because the execution rate of vector computers generally increases with the size of the vectors being operated on. This is strictly true for memory-to-memory machines, such as the Cyber 205. For register-to-register machines, such as all Cray machines, it is true in a piecewise sense. Thus, we need to reconsider the assertion made above that the frontal matrix must be kept small.

In the frontal method, longer vectors would occur if pivoting were delayed and the number of variables and equations in the frontal matrix were allowed to become greater than necessary. The delaying of one pivot would result in one extra equation and one extra variable remaining in the frontal matrix. The following shows that such a procedure would not reduce the execution time. In the following, the subscript N refers to elimination in a frontal matrix with N columns and M rows, and $N + 1$ refers to the elimination in a frontal matrix with $N + 1$ columns and $M + 1$ rows, the matrix with one extra row and column because a pivot has been delayed.

Let T be the time taken to perform elimination using one pivot, O be the number of operations performed in updating the elements in the remaining frontal matrix during this elimination, and R be the rate of operations during the elimination. Clearly,

$$T_N = \frac{O_N}{R_N}$$

$$T_{N+1} = \frac{O_{N+1}}{R_{N+1}}$$

and

$$O_N = 2(N-1)(M-1)$$

$$O_{N+1} = 2NM$$

This is because there is a multiplication and an addition operation involved in the updating of each element in the remaining frontal matrix. For vectors of length N, corresponding to the row length of the frontal matrix, the execution rate of a vector computer can be given by (Calahan and Ames, 1979; Hockney, 1982):

$$R_N = \frac{AN}{(N+B)}$$

$$R_{N+1} = \frac{A(N+1)}{(N+1+B)}$$

where A and B are positive constants whose values depend on the particular computer. These equations arise from the fact that vector operations require a start-up time. That is:

$$T_N = N/R_\infty + (1+s)T_0$$

where T_0 is the start-up time, R_∞ is the maximum execution rate of the operation, and s is the number of vector "strips" that must be operated on. For a register-to-register machine, each strip has a length equal to the vector register length L (64 for Cray machines). Thus s is the integer part of N/L. For a memory-to-memory machine, $s = 0$. Start-up time is required for each strip that is operated on. From this, and using

$$R_N = \frac{N}{T_N}$$

it can be seen that:

$$R_N = \frac{N}{(N/R_\infty + (1+s)T_0)}$$

which is of the form above with $A = R_\infty$ and $B = R_\infty T_0(1 + s)$. Note that B is only piecewise constant, since as N increases, s will increase whenever N exceeds a higher multiple of L, thus causing a drop in computation rate. Using these relationships,

$$T_N = \frac{2(N-1)(M-1)(N+B)}{AN}$$

$$T_{N+1} = \frac{2NM(N+1+B)}{A(N+1)}$$

For a speedup to occur by keeping the extra row and column, T_{N+1} must be less than T_N, or:

$$N^2M(N+1+B) < (N^2-1)(M-1)(N+B)$$

But M and N are positive integers and B is positive, so this is clearly false. From this it can be seen that it is important to keep the frontal matrix as small as possible, and that frontal matrix size is a good measure of the efficiency of the method.

Since the row ordering determines the size of the frontal matrix, which in turn determines the efficiency of the method, we now look in detail at several strategies for ordering the rows of a flowsheeting matrix.

Reordering Strategies and Results

We consider now the problem of reordering the flowsheeting matrix into a form that will tend to keep the frontal matrix small.

Desirable matrix forms

Since the frontal method was developed with banded matrices in mind, this is obviously one form of matrix that is desirable. If the half-bandwidth B of a matrix is the distance from the diagonal to the nonzero element farthest from the diagonal, then the frontal matrix need only contain $B + 1$ rows and $2B + 1$ columns. As long as the bandwidth can be kept small this is clearly an attractive ordering.

As discussed previously, although flowsheeting matrices have a basically banded structure, the presence of off-band blocks means they are not strictly banded. If the band is enlarged to encompass the off-band blocks, the bandwidth, and thus the size of the frontal matrix could become quite large. Techniques for reordering to a band form with minimal bandwidth are discussed below.

Although the frontal approach was developed for band and profile matrices, there are other matrix forms that are desirable for this approach as well. Matrices in upper-triangular form are perhaps the most desirable, because as each row enters the frontal matrix there is one pivot variable available. Because of this the largest frontal matrix need only contain one row and N columns, where N is the number of equations in the system being solved.

Flowsheeting matrices are not strictly triangular of course. Some nonzeros will always be found below the diagonal, in the so-called spike columns. (For a complete discussion of such a spiked matrix form see Stadtherr and Wood, 1984a). For a triangular matrix with spikes running the length of the matrix, the size of the largest frontal matrix is $NSPK + 1$ rows and N columns, where $NSPK$ is the number of spikes. This is because the first pivot cannot be eliminated until $NSPK + 1$ rows have entered the frontal matrix. After the first $NSPK$ rows, each time an equation is added to the frontal matrix one can be eliminated. It is important to note that the maximum front size given here is actually an upper bound that applies to the case in which the triangular part of the matrix is full. In practice for flowsheeting matrices, the triangular part is usually quite sparse, thus reducing the size of the frontal matrix required.

We now look at different reordering methods and present results for each. First we consider the unsymmetric reorderings that attempt to put matrices in a nearly triangular form; then we consider bandwidth reduction methods that produce a banded matrix and attempt to minimize the bandwidth.

Unsymmetric reorderings

As discussed above, triangular form can be a desirable form for the frontal approach. Reordering algorithms that attempt to put the matrix in triangular form have been widely used in the general-purpose solution of sparse matrices on sequential machines, since they reduce the amount of fill-in that occurs, thus reducing the storage requirement and the operation count. Fill-in is not a concern here, but obtaining a nearly triangular form is.

The desired row–column ordering for the frontal approach is actually the opposite of the order used for a general-purpose sparse matrix solver on a sequential machine. For that case, it is desirable to have small rows first and small columns last, that is, a lower triangular form. For the frontal method, it is desirable to process small columns first and small rows last, that is, an upper triangular form. Because of this, matrices to be solved by the frontal approach can be ordered by taking an *a priori* reordering method intended to reduce fill-in in general sparse matrix solvers and then reversing the order of the rows. This can be clearly seen in Table 1, which shows the size of the frontal matrix required using several different sparse matrix reordering algorithms. A description of the reordering methods P^4, SPK1, SPK2, BLOKS, and the various HP algorithms can be found in Stadtherr and Wood (1984a). The P^4 algorithm is from Hellerman and Rarick (1972) and the HP algorithms are from Lin and Mah (1977). The other algorithms were developed by Stadtherr and Wood (1984a). The prefix "r-" in the tables indicates the use of a reverse ordering. The flowsheeting test problem used here, and those used in subsequent tests, have been used previously by Stadtherr and Wood (1984a,b) and are taken from Wood (1982). It can be seen that the reverse ordering results in a much more compact frontal matrix than the original ordering.

Table 2 shows the frontal matrix size that is required for reverse-SPK1, reverse-SPK2, reverse-BLOKS, and reverse-HP30 on six flowsheeting test problems. It can be seen that in all

Table 1. Frontal Matrix Size Resulting from Orderings and Reverse Orderings (N = 372)

Reordering Method	Max. Frontal Matrix Size	
	Eqs.	Variables
SPK1	160	160
r-SPK1	55	160
SPK2	160	160
r-SPK2	44	160
BLOKS	160	163
r-BLOKS	31	160
HP	160	161
r-HP	31	161
HP10	160	161
r-HP10	37	161
HP20	160	166
r-HP20	40	166
P4	131	131
r-P4	44	131

Table 2. Frontal Matrix Size Required for Several Unsymmetric Reorderings

Example	Eqs. (Nonzeros)	Reordering Method	Max. Frontal Matrix Size	
			Eqs.	Variables
1	372 (3,253)	r-SPK1	55	160
		r-SPK2	44	160
		r-BLOKS	31	160
		r-HP30	40	166
		r-P4	44	131
2	814 (7,448)	r-SPK1	87	244
		r-SPK2	74	278
		r-BLOKS	47	263
		r-HP30	51	267
		r-P4	79	202
3	1,060 (6,254)	r-SPK1	203	490
		r-SPK2	25	155
		r-BLOKS	20	161
		r-HP30	30	182
		r-P4	107	270
4	1,564 (9,369)	r-SPK1	168	428
		r-SPK2	39	183
		r-BLOKS	17	140
		r-HP30	30	198
		r-P4	87	220
5	2,224 (13,577)	r-SPK1	174	466
		r-SPK2	49	245
		r-BLOKS	17	180
		r-P4	87	220
6	2,878 (17,772)	r-SPK1	142	413
		r-SPK2	43	299
		r-BLOKS	17	215
		r-P4	87	235

cases the r-BLOKS reordering performs the best, and often by a substantial margin. Moreover, this reordering is also by far the fastest of these reordering algorithms. The BLOKS algorithm was developed specifically for flowsheeting problems, and performs so well because it takes into account the block-oriented structure of the flowsheeting matrix. The frontal matrix size obtained using r-BLOKS is quite reasonable. The performance of the frontal method using this reordering is considered below.

Bandwidth reduction

Bandwidth reduction methods attempt to reorder a sparse matrix so that the half-bandwidth,

$$\max \operatorname{abs}(i - j) \quad a_{ij} \neq 0$$

is a minimum. Profile reduction, a technique similar to bandwidth reduction, involves attempting to minimize the profile of a matrix:

$$\sum_{i=1}^{N} (i - \min \{j | a_{ij} \neq 0\})$$

The profile of a matrix can be thought of as the sum of a local bandwidth, that is, the sum of the bandwidth of all rows. These techniques, to date, have been developed only for matrices with a symmetric nonzero structure (Duff, 1984b). Such techniques perform symmetric permutations. That is, the same permutations are used to order both rows and columns in the matrix. These methods can be used for unsymmetric matrices by reordering a matrix with the structure of $A + A^T$, rather than just A. A reordering that reduces the bandwidth of $A + A^T$ also reduces the bandwidth of A. While the reordering algorithm must consider more matrix elements while finding a reordering that reduces the bandwidth of $A + A^T$, the reordering that is found can be applied to the matrix A, so the extra storage required to store $A + A^T$ rather than A is not needed for the solution of the matrix.

Lewis (1982) coded efficient versions of the symmetric reordering algorithms of Gibbs, Poole, and Stockmeyer (GPS) and Gibbs and King (GK). These symmetric reordering algorithms were applied to the flowsheeting test matrices used in this study. These orderings are improvements of the algorithms of Cuthill and McKee (1969) and King (1970). The GPS method attempts to reduce the matrix bandwidth and the GK method attempts to reduce the matrix profile. Table 3 shows that while the flowsheeting matrices do have an overall banded structure, the original off-band blocks still cause the matrices to have a large bandwidth and profile even when these reorderings are used. This indicates that the bands that result with these reorderings are actually very sparse.

Since, in the frontal approach, a variable enters the frontal matrix only when it is first encountered, it is possible that a large bandwidth might not necessarily result in a large frontal matrix if the band is sparse, as is the case in flowsheeting matrices.

Table 3. Bandwidth and Profile Resulting from Flowsheeting Matrices Reordered with Gibbs-Poole-Stockmeyer and Gibbs-King Algorithms

Method*	Eqs.	Bandwidth	Profile
GPS	372	93	18,440
GK	372	138	17,220
GPS	814	225	87,125
GK	814	318	73,891
GPS	1,060	269	138,798
GK	1,060	945	82,862

*GPS, Gibbs-Poole-Stockmeyer.
GK, Gibbs-King.

Table 4 compares the size of the frontal matrix needed for the symmetric reorderings and the r-BLOKS reordering. As can be seen, except for one case the size of the frontal matrix required for the symmetric reorderings is very large. A large frontal matrix indicates many unnecessary operations and an inefficient algorithm.

It should also be noted that in addition to the symmetric and unsymmetric reorderings reported on here, we also considered some simpler reordering techniques. In all cases the r-BLOKS ordering provided the best results.

Performance of the frontal method

We now take the best reordering (r-BLOKS) and analyze the performance of the frontal method on the six flowsheeting test matrices. To do this we consider the percentage of operations done as vector operations and the percentage of operations wasted because they were done on zeros. These figures were determined by inserting counters into the frontal code. Operations were considered vectorizable if the loops were inner loops with operations on vectors and had no recursion, subroutine calls, or IF statements. Wasted operations were those that were added because of the storage and operation on zero elements in the frontal matrix.

Table 4. Size of Frontal Matrix Required for Symmetric Reorderings

		Max. Frontal Matrix Size	
Eqs.	Reordering	Eqs.	Variables
372	GPS	166	215
	r-GPS	128	216
	GK	170	221
	r-GK	108	221
	r-BLOKS	31	160
814	GPS	303	545
	r-GPS	321	545
	GK	299	536
	r-GK	326	536
	r-BLOKS	47	263
1,060	GPS	297	493
	r-GPS	216	493
	GK	36	140
	r-GK	119	140
	r-BLOKS	20	161

Table 5. Vector and Wasted Operations for Frontal Approach with Reverse-BLOKS Reordering

Example	Eqs.	Vector Opns. %	Wasted Opns. %
1	372	93.8	30.9
2	814	95.8	38.9
3	1,060	91.6	32.5
4	1,564	89.4	27.7
5	2,224	89.9	27.6
6	2,878	88.8	31.2

The results are shown in Table 5. Over a wide range of problem sizes, we consistently obtained about 90% vector operations. Thus using the frontal method we can obtain a high computational rate. The tradeoff is that about 30% of the operations done will be wasted. This tradeoff appears to be clearly in our favor, however. For instance, if we estimate the speedup of a vectorized frontal code over a nonvectorized code (in which no operations would be wasted) using Amdahl's law, with a maximum possible speedup of 20, and 90% vectorization, we obtain 11.8. If we take into account the fact that approximately 30% of the operations are unnecessary, we get an effective speedup of 8.3. In general, according to Amdahl's law, the percentage speedup provided by 90% vectorization is considerably more than enough to offset the need to do unnecessary operations on zeros. In practice, we would expect the actual speedup to be less, since the figures above are based on the maximum possible speedup, which is not likely to be attainable, and since in computing actual speedup it should be relative to the best scalar code, in this case LU1SOL.

These results indicate that as long as a good reordering scheme, such as r-BLOKS, is used to keep the frontal matrix relatively small, the frontal method is a very effective way to vectorize the solution of flowsheeting matrices.

Conclusions

The equation-based approach to process flowsheeting appears to offer the most promise for the effective use of vector processing, at least if the sparse matrix part of the computation can be vectorized effectively. To do this we need to abandon the traditionally used sparse matrix methods. With a good matrix reordering strategy, the frontal method provides an effective algorithm for vectorizing the sparse matrix computations. However, it should be emphasized that while the sparse matrix computations are a major part of the overall computation, another key part is often the calculation of physical properties. Attention needs to be given to the vectorization of this part of the calculation as well.

Acknowledgment

This work was supported in part by the National Science Foundation under Grant No. DMC-8520663. We also acknowledge the support of the National Center for Supercomputing Applications at the University of Illinois.

Literature Cited

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

Calahan, D. A. "Block-Oriented Sparse Equation Solver for the Cray-1," *Proc. 1979 Conf. on Parallel Processing*, IEEE Computer Soc. Press, Silver Spring, MD (1979).

Calahan, D. A., "A Block-Oriented Equation Solver for the Cray-1," SEL Rep. No. 136, Systems Eng. Lab., Univ. Michigan (1980).

Calahan, D. A., and W. G. Ames, "Vector Processors: Models and Applications," *IEEE Trans. Circuits and Systems*, **CAS-26**(9) (1979).

Cameron, I. T., "Mass Balances for Process Design by the Frontal Solution Method," M.S. Thesis, Univ. Washington (1977).

Cavett, R. H., "Application of Numerical Methods to the Convergence of Simulated Processes Involving Recycle Loops," *Proc. Am. Petrol. Inst.*, **43**, 57 (1963).

Chen, H. S., and M. A. Stadtherr, "On Solving Large Sparse Nonlinear Equation Systems," *Comput. Chem. Eng.*, **8**, 1 (1984).

Cuthill, E. H., and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proc. 24th ACM Nat. Conf.*, 157 (1969).

Dembart, B., and K. W. Neves, "Sparse Triangular Factorization on Vector Computers," *Exploring Applications of Parallel Processing to Power System Analysis Problems*, EPRI Rep. EL-566-SR, Palo Alto (1977).

Duerre, K. H., and A. C. Bumb, "Implementing ASPEN on the Cray Computer," Los Alamos Rep. LA-UR-81-3528, Los Alamos Nat. Lab., Los Alamos, NM (1981).

Duff, I. S., "Recent Developments in the Solution of Large Sparse Linear Equations," IRIA 4th Int. Symp. on Computing Methods in Applied Sciences and Engineering, December 10–14, 1979, Versailles (1979).

———, "Design Features of a Frontal Code for Solving Sparse Unsymmetric Linear Systems Out-of-Core," *SIAM J. Sci. Stat. Comput.*, **5**, 270 (1984a).

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

Duff, I. S., and J. K. Reid, "Some Design Features of a Sparse Matrix Code," *ACM Trans. Math. Software*, **5**, 18 (1979).

———, "Experience of Sparse Matrix Codes on the Cray-1," *Comput. Phys. Commun.*, **26**, 293 (1982).

———, "The Multifrontal Solution of Unsymmetric Sets of Linear Equations," *SIAM J. Sci. Stat. Comput.*, **5**, 633 (1984).

Elkin, B., "A Survey of Chemical Engineering Applications in Supercomputing," AIChE Ann. Meet., Washington, DC (1988).

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

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

Harrison, B. K., "Performance of a Process Flowsheeting System on a Supercomputer," *Comput. Chem. Eng.*, **13**, 855 (1989).

Hellerman, E., and D. C. Rarick, "The Partitioned Preassigned Pivot Procedure (P^4)," *Sparse Matrices and Their Applications*, D. J. Rose, R. A. Willoughby, eds., Plenum, New York (1972).

Hockney, R. W., "Characterization of Parallel Computers and Algorithms," *Comput. Phys. Commun.*, **26**, 285 (1982).

Hood, P., "Frontal Solution Program for Unsymmetric Matrices," *Int. J. Numer. Methods Eng.*, **10**, 379 (1976).

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).

Jennings, A., "A Compact Storage Scheme for the Solution of Symmetric Linear Simultaneous Equations," *Comput. J.*, **9**, 281 (1966).

Kaijaluoto, S., P. Neittaanmäki, and J. Ruhtila, "Comparison of Different Solution Algorithms for Sparse Linear Equations Arising from Flowsheeting Problems," *Comput. Chem. Eng.*, **13**, 433 (1989).

King, I. P., "An Automatic Reordering Scheme for Simultaneous Equations Derived from Network Analysis," *Int. J. Numer. Meth. Eng.*, **2**, 523 (1970).

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

Lewis, J. G., "Implementation of the Gibbs–Poole–Stockmeyer and Gibbs–King Algorithms," *ACM Trans. Math. Software*, **8**, 180 (1982).

Lewis, J. G., and H. D. Simon, "The Impact of Hardware Gather/Scatter on Sparse Gaussian Elimination," *SIAM J. Sci. Stat. Comput.*, **9**, 304 (1988).

Lin, T. D., and R. S. Mah, "Hierarchical Partition—A New Optimal Pivoting Algorithm," *Math. Program.*, **12**, 260 (1977).

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).

Perkins, J. D., and R. W. H. Sargent, "Speedup: A Computer Program for Steady-State and Dynamic Simulation and Design of Chemical Processes," *Selected Topics on Computer-Aided Process Design and Analysis*, R. S. H. Mah, G. V. Reklaitis, eds., *AIChE Symp. Ser.* (1982).

Petersen, W. P., "Vector FORTRAN for Numerical Problems on Cray-1," *Commun. ACM*, **26**, 1008 (1983).

Pottle, C., "Solution of Sparse Linear Equations Arising from Power System Simulation on Vector and Parallel Processors," *ISA Trans.*, **18**, 81 (1979).

Seader, J. D., "Process Design," Conf. on Application of Computational Mathematics and Large-Scale Scientific Computing in Process and Chemical Engineering, Minneapolis, December 18–21 (1985).

Stadtherr, M. A., and C. M. Hilton, "On Efficient Solution of Large-Scale Newton-Raphson Based Flowsheeting Problems in Limited Core," *Comput. Chem. Eng.*, **6**, 115 (1982a).

———, "Development of a New Equation-Based Process Flowsheeting System: Numerical Studies," *Selected Topics on Computer-Aided Process Design and Analysis*, R. S. H. Mah, G. V. Reklaitis, eds., *AIChE Symp. Ser.* (1982b).

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

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).

———, "Sparse Matrix Methods for Equation-Based Chemical Process Flowsheeting. II: Numerical Phase," *Comput. Chem. Eng.*, **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).

Westerberg, A. W., and T. J. Berna, "Decomposition of Very Large Scale Newton-Raphson Based Flowsheeting Problems," *Comput. Chem. Eng.*, **2**, 61 (1978).

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

Zitney, S. E., "A Frontal Code for ASPEN PLUS on Advanced-Architecture Computers," AIChE Ann. Meet., Chicago, November 11-16 (1990).

Zitney, S. E., and M. A. Stadtherr, "Computational Experiments in Equation-Based Chemical Process Flowsheeting," *Comput. Chem. Eng.*, **12**, 1171 (1988a).

———, "A Frontal Algorithm for Equation-Based Chemical Process Flowsheeting on Vector Computers," *Proc. 3rd Int. Symp. on Process Systems Engineering*, Institution of Engineers—Australia, 288 (1988b).

Manuscript received Oct. 16, 1989, and revision received Aug. 22, 1990.