Parallel Triangular Decompositions of an Oil Refining Simulation

Xiaodong Zhang
Computer Science Program
University of Texas at San Antonio
San Antonio, Texas 78249
zhang@ringer.cs.utsa.edu

Abstract

One important process in oil refining is to separate the crude oil into various oil products. This process is called *distillation*. In designing a complex distillation column, a large computer simulation is conducted. This paper presents our experience with parallelizing an oil refining simulation application that computes the composition of the various oil products in designed refining columns operated under a given set of conditions. Mathematical models for the simulation form large sparse nonlinear systems of equations. Triangular decompositions of sparse nonlinear systems are fundamental numerical methods in this simulation computing. Different approaches have been applied to carry out this simulation in parallel. Parallelisms of the simulation are exploited at three levels — direct parallelization, structured parallelization and asynchronous parallelization of the problem. The approaches of this practical research provide insight into some important issues of parallel computing for real-world applications. The parallel programs were implemented and run on the Intel iPSC/860. Parallel computing results are presented with comparisons and discussions.

1 Introduction

One important process in oil refining is to separate the crude oil into various oil products. This process is called *distillation*, which is a thermal separation method that separates a mixture of liquids according to the differences in their vapor pressures or their boiling point ranges. It is done by letting the liquid mixture flow through a distillation column. Researchers usually carry out computer aided design and simulation of complex multicomponent separation processes of distillation using the well-known equilibrium stage model [12]. Briefly, this model includes the assumption that the streams leaving a particular stage are in equilibrium with one another. The computation involves solving very large sparse nonlinear systems of equations which are generated by equilibrium relationships, component-material balances, total-material balances and energy balance. The numerical problem in the simulation is to solve a large-scale nonlinear system of equations of the form

$$F(x) = 0, \tag{1}$$

where $F : R^n \rightarrow R^n$ and the Jacobian $J(x)$ is sparse. Sizes of the sparse nonlinear systems for a normal distillation column model range from 1,000 to 5,000 variables. A modern refinery has a number of distillation columns connected in series, and their simulation generate a multi-level large sparse nonlinear system. The computation is very time-consuming, and some important real-time instances can not be solved on time with current sequential methods and machines. In addition to making the algorithms parallel, the methods for handling the sparsity of the system must be very efficient. Therefore this application, with its sparse nonlinear systems makes a good candidate for research into algorithms and implementations on parallel computers.

The approach of this practical research is to start with a complete and real-world sequential application program, and parallelize it in different ways in order to gain experience in developing efficient parallel programs on a distributed memory multicomputer. We exploited parallelism in this application at three levels:

1. *Direct parallelization*: Parallelizing the sequential program without changing the basic algorithm structure. This is done by directly applying existing parallel algorithms to the sequential program.

2. *Structured parallelization*: Restructuring the program and transforming the application problem into special structures so that the computation can be effectively decomposed for parallel computing.

3. *Asynchronous parallelization*: Not waiting for predetermined data to become available, but trying to solve the problem with whatever data happen to be available at the time under certain conditions. The purpose of this approach is to overlap communications with computations.

The next section describes oil refining simulation problems and introduces a nonlinear system that comes from such a

ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0271...$1.50

real-world application. Section 3 briefly discusses the problems of the direct parallelization approach, namely the direct application of Newton's method to the problem, but the emphasis is on the structured parallelization approach. Specifically, we apply the triangular decomposition methods proposed by Dennis, Martínez and Zhang [5] [6] to this refining simulation. The asynchronous parallelization approach is described in section 4. Section 5 contains computation results for the simulation using both structured and asynchronous approaches on the Intel iPSC/860, a distributed memory multicomputer. The results show the advantage of the both approaches. Section 6 gives a discussion and summary of the work.

2 Distillation column modeling

2.1 Overview of the oil distillation process

A *phase* is defined as that part of a system that is chemically and physically uniform throughout. In practice, there are three phases — vapor, liquid and solid. At a certain pressure and a certain temperature, two phases can exist in equilibrium, which means that the tendency to escape from one phase to another is equal to the tendency to escape in the opposite direction. There are three equilibrium lines: solid-vapor equilibrium, liquid-vapor equilibrium, and solid equilibrium. Moreover, three phases can exist together, and this occurs at the point called the triple point or the three-phase equilibrium point. The phase equilibrium principle is the fundamentals of oil refining.

One important oil refining process conducted in an oil distillation column uses a cylindrical column fitted out interally with trays. Each tray is a horizontal circular plate above which a layer of condensed liquid can be collected. In the process of distillation, a certain pressure and temperature are provided to separate the crude oil into various oil products. Vapor flows from the bottom to the top and gives up energy to the liquid phase, which in turn flows downward. This movement results in vaporization of the lower boiling components of the liquid phase and condensation of the higher boiling components of the vapor phase. After the column has been running for some time, the temperature through the entire column stabilizes. The highest temperatures are found at the inlet of the column. Under ideal conditions, an equilibrium between liquid and vapor is present on each tray, depending on the temperature of the tray. A conventional distillation column is defined as one that has one feed and two product streams, the distillate D and the bottom product B, as shown in Figure 1.

There are two types of distillation: *atmospheric distillation*, which is conducted in a normal atmospheric environment, and *vacuum distillation*, which is conducted in a special vacuum environment. On the other hand, there are two kinds of simulation for distillation column design: *multicomponent simulation*, which computes multiple compositions of a distillation, and *binary component simulation* which computes only two compositions, for liquid products and for vapor products respectively in each tray of a distillation column.

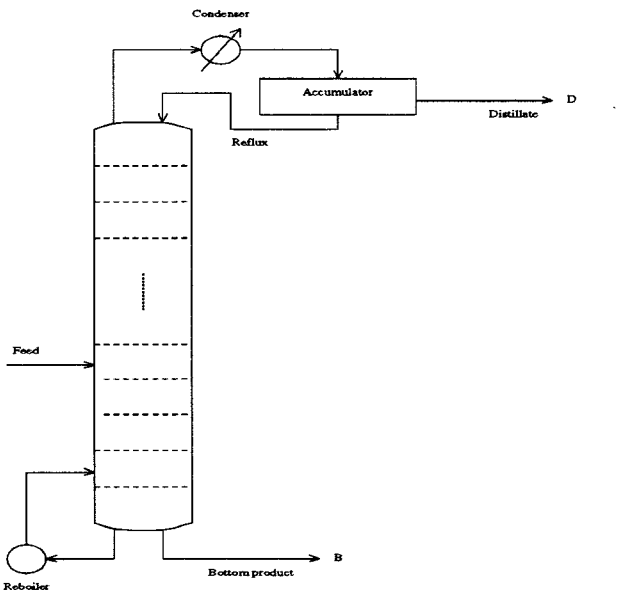

Figure 1: Sketch of a conventional column

2.2 Computer simulation for a distillation column

The simulation we conducted came from a real-world design of a distillation column in a major oil company in Houston, Texas. The simulation consists of two major computations: a single column simulation and a multiple distillation column series simulation. The single column model carries out a simulation of a binary component atmospheric distillation column using 155 trays. The multiple distillation column series simulation is to compute oil distillation components under a refining model which has a number of identical distillation columns connected in series, where the liquid flow goes in one direction. Besides the initial feed, top product and bottom product, each column links to the next column in one direction by certain liquid outputs in the middle part of the column, which serve as the connection feed to the next column. The multiple column series model is simplified in Figure 2.

The mathematical model for simulating the single/multiple distillation column simulation operated at a given set of conditions is determined by the following phase-equilibrium equations:

1. Equilibrium relationships,
2. Component-mass balance,
3. Total mass balance, and
4. Energy balance (Enthalpy balance).

For detailed information about these equations and their physical meanings, the readers may refer to [12].

In a distillation column designed this way, the *ith* tray ($i =$

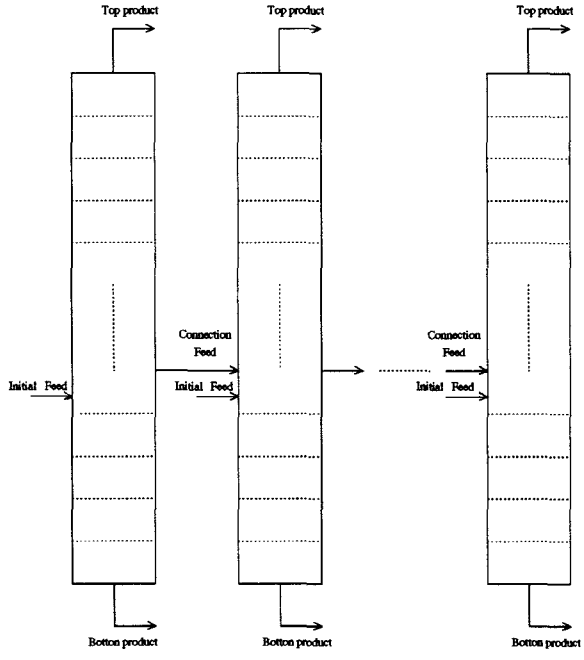

Figure 2: Multiple distillation column series.

1, ...155) has 9 variables:
$LC_{i,1}$ — Liquid Component flow data for component 1,
$LC_{i,2}$ — Liquid Component flow data for component 2,
$VC_{i,1}$ — Vapor Component flow data for component 1,
$VC_{i,2}$ — Vapor Component flow data for component 2,
TL_i — Total Liquid flow data,
TV — Total Vapor flow data,
T_i — Temperature,
LE_i — Liquid Enthalpy, and
VE_i — Vapor Enthalpy.

Besides the $155 \times 9 = 1395$ tray variables, there are another 19 variables for feed, reflux, reboil etc:
$SLCP_1$ and $SLCP_2$ — Two Side Liquid Component Products,
$SLTF$ — Side Liquid Total Product Flow,
$SLPT$ — Side Liquid Product Temperature,
$SLPP$ — Side Liquid Product Presure,
$SLPE$ — Side Liquid Product Enthalpy,
$SLPV$ — Side Liquid Product Vapor Fraction,
FC_1 and FC_2 — Feed Components,
TFF — Total Feed Flow,
FT — Feed Temperature,
FP — Feed Presure,
FE — Feed Enthalpy,
FVF — Feed Vapor Fraction,
DD_1, DD_2 and DD_3 — Duty Data,
$RF4$ — Reflux Ratio, and
BU — Boil Up Ratio.

The system has a total of 1414 variables and 1414 equations and forms a sparse nonlinear system of equations in the form of (1). The system structure of the single column simulation is briefly described in Table 1.

Number of variables	1414
Number of parameters	52
Percentage of non-zeros in Jacobian	0.35%

Table 1: The system structure of the single column simulation.

Number of columns	8
Number of variables in each connection	4
Total number of variables	11,376
Number of parameters	416
Percentage of non-zeros in Jacobian	0.04%

Table 2: The system structure of the multiple column simulation.

The multiple distillation column series simulation connects a number of identical single columns together in one direction. The simulation naturally generates a block triangular system. The block triangular structure is well balanced because each diagonal block is the sparse system of the single column simulation. The connections of these columns are additional equations that set some liquid outputs of a column to the inputs of its forward neighbor column. These equations are placed in the bottom of the triangular system. The system structure of the multiple column simulation is briefly described in Table 2.

3 Triangular decompositions for the sparse nonlinear system

3.1 Basic idea

For $k = 0, 1, ...,$ the standard Newton's method for (1) generates successive estimates $x^k \in R^n$ of a solution $x^* \in R^n$ of (1) as:

$$\text{Solve} \quad J(x^k)s^N = -F(x^k) \quad \text{and set} \quad x^{k+1} = x^k + s^N. \quad (1)$$

The simplest approach to solve the simulation problem is to directly apply this standard Newton's method. The major part of the computation is to solve the linear Jacobian system at each iteration. A standard version of a linear system solver (see e.g. [11] and [13]) may be applied to the implementation of Newton's method to solve the simulation problem. There are two major problems with applying this straightforward approach. First, the parallel operations are performed by row and column distributions. However, a distributed memory multicomputer favors block distributions, or large grain size. The major part of the performance degradation would be in the LU decomposition of the Jacobian matrix at each iteration because non-trivial communications are involved. Second, direct parallelization treats the sparse system as a dense system, which wastes a large amount of computing cycles and memory bandwidth to deal with 0's. In the case of large sparse nonlinear problems, a single standard method, such as Newton's method, may not handle all

the instances of (1) efficiently, but rather the algorithm must take into account the sparsity structure and other special characteristics of the problem. In addition, the randomly sparse and irregular Jacobian matrix structure makes the computation difficult to vectorize and parallelize.

Dennis, Martinez and Zhang [5] [6] suggest one way to use existing technology automatically to tailor methods for particular sparsity patterns. This approach is motivated by using a graph coloring method to compute $J(x^k)$ efficiently (see [3] [4], and [7]). After that, one might apply the Harwell code MA28 [8] [9] to solve the linear system (1). (The MA28 program decomposes a sparse linear system into a block triangular form). If so, then permutation matrices P and Q would be found such that

$$PJ(x^k)Q(Q^T s^N) = -PF(x^k) \tag{2}$$

is a block lower triangular system. If this block lower triangular system exists, the original system is called a reducible system.

In summary, this process of nonlinear triangular decomposition reorders the equations and variables *before* starting the nonlinear solution process, rather than after starting it. If Newton's method is implemented as outlined above, then reordering information is computed and used in the inner linear loop, but it is not used by the nonlinear outer loop.

A straightforward approach without any complex graph transformation programs would apply the Harwell code MA28 to the Jacobian matrix to generate a structure matrix filled with 0's and 1's, where 1's represent the non-zero elements of the Jacobian matrix. Using the information of the structure matrix, we can manually reorder the nonlinear system into a lower block triangular form. This simple method may only apply to a system of moderate size for experimental studies.

3.2 Newton-Gauss-Seidel versus Gauss-Seidel-Newton

If we apply Newton's method to solve the block triangular transformation of (1) given by

$$F(x) = \begin{cases} F_1(x_1) \\ F_2(x_1,\ x_2) \\ . \\ . \\ . \\ F_m(x_1,\ x_2,\ ...,\ x_m) \end{cases} = 0, \tag{3}$$

where

$$x = (x_1,\ x_2,\ ...,\ x_m)^T \in R^{n_1} \times R^{n_2} \times \ ... \ \times R^{n_m} = R^n,$$

$$F_i : R^{n_1} \times R^{n_2} \times \ ... \ \times R^{n_i} \rightarrow R^{n_i}, \quad i = 1,\ 2,\ ,...,\ m$$

and

$$\sum_{i=1}^{m} n_i = n\ ,$$

then we have to solve

$$\begin{cases} \frac{\partial F_1(x_1^k)}{\partial x_1} s_1^N = -F_1(x_1^k) \\ \frac{\partial F_2(x_1^k, x_2^k)}{\partial x_1} s_1^N + \frac{\partial F_2(x_1^k,\ x_2^k)}{\partial x_2} s_2^N = -F_2(x_1^k, x_2^k) \\ . \\ . \\ . \\ \frac{\partial F_m(x_1^k,...,x_m^k)}{\partial x_1} s_1^N + ... + \frac{\partial F_m(x_1^k,...,x_m^k)}{\partial x_m} s_m^N = -F_m() \end{cases} \tag{4}$$

and set $x_i^{k+1} = x_i^k + s_i^N$ for $i = 1, ..., m$ and $k = 0, 1, 2, ...$.

We should use forward block substitution, which is the same as the block Gauss-Seidel linear iteration, to compute the Newton step by:

$$\frac{\partial F_1(x_1^k)}{\partial x_1} s_1^N = -F_1(x_1^k), \tag{5}$$

and for $i = 2, ..., m$, s_i^N comes from the $n_i \times n_i$ linear system

$$\frac{\partial F_i(x_1^k, ..., x_i^k)}{\partial x_i} s_i^N = -(F_i(x_1^k, ..., x_i^k) + \sum_{j=1}^{i-1} \frac{\partial F_i(x_1^k, ..., x_i^k)}{\partial x_j} s_j^N). \tag{6}$$

But now the key observation is that the function on the right side of (6) are just the first order Taylor approximations to $-F_i(x_1^k + s_1^N, x_2^k + s_2^N, ..., x_{i-1}^k + s_{i-1}^N, x_i^k)$. Thus, if we can compute values of F_i independently, it must surely be better to do so than to build a Taylor approximation to it at the expense of computing the strict lower triangular part of the Jacobian. This suggests using the block version of the Gauss-Seidel-Newton method given by

$$\frac{\partial F_i(x_1^k, ..., x_i^k)}{\partial x_i} s_i = -F_i(x_1^k + s_1, ..., x_{i-1}^k + s_{i-1}, x_i^k) \tag{7}$$

where $x_i^{k+1} = x_i^k + s_i$, $i = 1, ..., m$ and $k = 0, 1, 2, ...$.

3.3 The Gauss-Seidel-Newton Method and its parallel version

The Gauss-Seidel principle uses new information as soon as it is available in order to achieve fast convergence, and this should be especially advantageous for block lower triangular systems. For $i = 1, ..., m$,

$$x_i^{k+1} = x_i^k - [J_i(x_i^{k,i})]^{-1} F_i(x_i^{k,i}) \ , \tag{8}$$

where $x_i^{k,i} = (x_1^{k+1}, ..., x_{i-1}^{k+1}, x_i^k)$, $J_i(\cdot) = \frac{\partial F_i(\cdot)}{\partial x_i}$. Below, we will use $x_i^{k,i}$ and J_i without further definition.

This method should converge in less time than Newton's method because only diagonal Jacobian blocks are evaluated rather than the whole matrix. Experiments demonstrating this advantage are described in Section 5.

In practice, applying more than one inner iteration to each block can reduce the total number of outer sweeps and the

total computing time. Of course, if there are too many inner iterations, one is doing nonlinear Gauss-Seidel, which is slower.

We found it best to implement the inner iterations in the cheapest way: we use a stationary Newton method with a new block function evaluation at each inner iteration, but only one Jacobian evaluation and factorization per block and per outer sweep.

Obviously little parallelism can be exploited from the Gauss-Seidel-Newton because most operations are sequential. A variation well suited to parallel computation evaluates and factor the diagonal blocks concurrently at x^k. Dennis, Martínez and Zhang [6] show that this modified method and its variations are quadratically convergent. The algorithm for q inner stationary Newton steps on each block for $l = 1, ...q$, is as follows:

$$x_i^{k,l} = x_i^{k,l-1} - [\hat{J}_i(x_i^{k,0})]^{-1} F_i(x_1^{k+1,l-1}, ..., x_{i-1}^{k+1,l-1}, x_i^{k,l-1}), \quad (9)$$

and

$$x_i^{k+1} = x_i^{k,l}, \quad (10)$$

where

$$\hat{J}_i(x_i^{k,0}) = \left. \frac{\partial F_i(x_1^{k,0}, ..., x_{i-1}^{k,0}, x_i)}{\partial x_i} \right|_{x_i = x_i^{k,0}}$$

3.4 Block Jacobi-Newton.

Finally, we include the well-known Jacobi-Newton method. Each iteration of this method independently solves m blocks of nonlinear equations by Newton's method:

$$x_i^{k+1} = x_i^k - [J_i(x_i^k)]^{-1} F_i(x_1^k, ..., x_i^k), \quad i = 1, ..., m, \quad (11)$$

where

$$J_i(x_i^k) = \left. \frac{\partial F_i(x_1^k, ..., x_{i-1}^k, x_i)}{\partial x_i} \right|_{x_i = x_i^k}$$

are the diagonal blocks of the Jacobian matrix, for $i = 1, ..., m$. Besides the simple structure permitting easy implementation, the Jacobi method also exhibits a high degree of parallelism. Its main drawback is its slow rate of convergence.

3.5 Advantages of the Triangular Decomposition Methods

In comparison with Newton's method, there are four advantages to applying the nonlinear triangular decomposition methods to solve a reducible sparse nonlinear system: 1) The evaluations and factorizations of the Jacobian are performed only on diagonal blocks; 2) Each sub-nonlinear system can be solved by different methods; 3) The condition numbers for diagonal blocks often are considerably smaller than the condition number for the whole Jacobian matrix; and finally, 4) The operations can be decomposed for parallel processing at the nonlinear level.

4 Asynchronous parallelization of the simulation

4.1 Communication issues on a multicomputer

On a modern multicomputer system such as the Intel iPSC/860, the computational subsystems of the nodes have been shown to have high speed. Our experimental studies also indicate that the iPSC/860 displays a basic disparity between computational speed on each node and internode communication speed. This disparity tends to favor a very large grain-size for computation, with minimal internode communication. Such an arrangement may be possible for problems which can be scaled in relation to the number of nodes available; however, in the case of fixed problem sizes the internode communication can become a major bottleneck as the number of processors is increased, resulting in poor parallel efficiency.

Improving the efficiency of solving various numerical problems in applications requires a careful study of communication overhead. Although low channel bandwidth is the largest factor, other factors also contribute to the problem, such as non-neighbor communication, the effects of channel contention, the proxy message overhead associated with long messages and others. All of these factors contribute to slow message transfer rates. Although they can be controlled to a limited extent, better results can be obtained by attempting *to overlap useful computation with ongoing communication.* This requires a design for asynchronous algorithms with concern for effectiveness and the correctness, and also requires efficient data structures implementing the asynchronous concepts on a distributed memory multicomputer. The basic concepts and several numerical applications of asynchronous parallel computing are well addressed in [1]. Other asynchronous methods for nonlinear optimization are in [2], [10], and [14].

The term *asynchronous computing* can be used for a scheme in which individual processors do useful work during the time normally spent waiting for a synchronized operation to complete. This concept has been implemented on a shared memory multiprocessor for the solution of several large and artificial block triangular nonlinear systems [5]. The algorithm refers to the case in which a processor is given a locally useful computation to perform while waiting for other processors to arrive at a barrier. In this section, we present a way that this concept can be "simulated" and extended on a distributed memory multicomputer. In such a case, instead of blocking for receipt of a message, the processor should recognize that no message has yet become available, and thus can turn its attention to another task which is not dependent upon the contents of the incoming message. In this way, the global computation can be advanced and cycles are not wasted.

4.2 Asynchronous methods for the simulation

4.2.1 Asynchronous Gauss-Seidel-Newton method

The objective of asynchronous approaches is to further exploit parallelism by overlapping communications with local

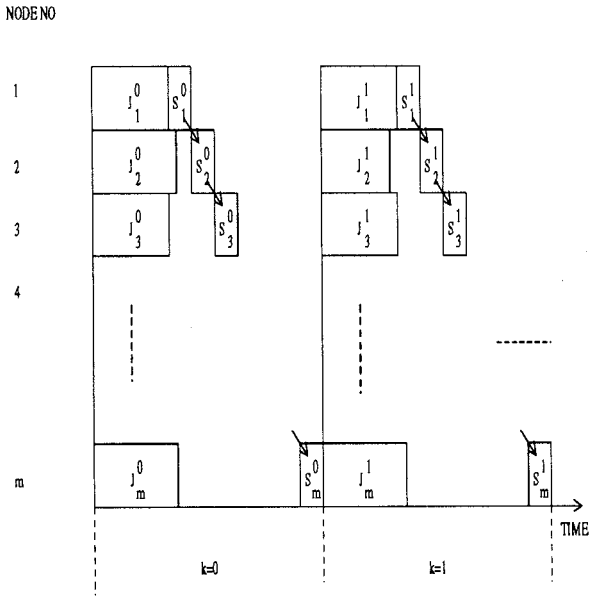

Figure 3: Data flow of the synchronous parallel Gauss-Seidel-Newton method.

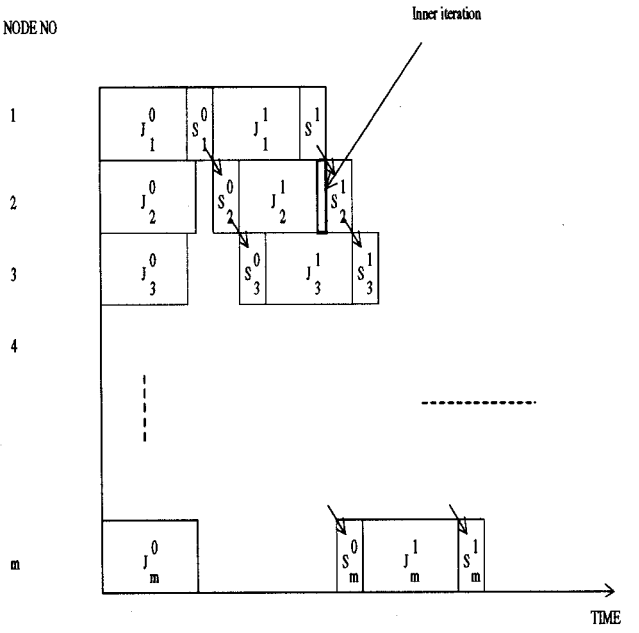

Figure 4: Data flow of the controlled asynchronous parallel Gauss-Seidel-Newton method.

computations. Of course certain algorithms may fail to converge when implemented asynchronously. However, a large number of positive and effective results are available for nonlinear block triangular systems. Recall that the synchronous parallel Gauss-Seidel-Newton method may be simplified in the data flow in Figure 1.

To simplify the description, we assume each processor is assigned a diagonal block. At each iteration, the Jacobian evaluations and the LU decompositions, represented by J_i^k $(i = 1, ..., m, k = 0, 1, 2...)$ are performed in parallel. Then the function evaluations and solutions of the x_i variables, represented by $s_i^k (i = 1, ..., m, k = 0, 1, 2, ...)$, are performed sequentially because of the dependent relationship. Idle gaps in each iteration degrade the overall parallel performance. Minimizing these gaps will significantly improve the performance. The sequential processes are important, because there is a danger that iterations performed on the basis of outdated information will not be effective and may even be counterproductive. A good asynchronous method must not have the problem of using outdated information.

The basic idea of our asynchronous Gauss-Seidel-Newton method is that each processor "steals" free computing cycles for extra inner iterations while waiting for the updated variables. Dennis, Martínez and Zhang [6] show analytically and experimentally that the inner iterations may achieve a better convergence rate. Therefore using inner iterations to asynchronously control the computing order of the Gauss-Seidel-Newton method should improve convergence and parallel performance. We call this method a *controlled asynchronous method.* Although local computation is overlapped with communication, the order of computing processes in each processor is controlled by the inner iterations. Data flow for the controlled asynchronous Gauss-Seidel-Newton method is shown in Figure 2.

In the controlled asynchronous Gauss-Seidel-Newton method, sequential operations for the solutions of block variables are performed only in the first iteration ($k = 0$). No more waiting cycles are spent in each processor during the remaining iterations. The iterations in each processor are performed using updated variable values from the related blocks. This is guaranteed by dynamically inserting inner iterations to fill the idle gaps. Figure 2 shows an example of such an inner iteration(s) in block 2 on processor 2 of the second iteration.

4.2.2 Asynchronous Jacobi-Newton method

Although each Jacobi-Newton iteration solves m blocks of nonlinear equations by Newton's method independently, each process has to wait at the barrier until all processes arrive. When the block sizes are reasonably close, the barrier overhead is small. However, when the block sizes widely differ, the barrier overhead may be considerable. Data flow for a standard Jacobi-Newton method is shown in Figure 3.

One variation of the parallel Jacobi-Newton implementation would eliminate the synchronization barrier at the end of each iteration, and let the Newton steps be solved asynchronously. The best choice is to implement this version of Jacobi-Newton on a shared memory multiprocessor, since there will be no extra overhead involved. The convergence rate of this asynchronous Jacobi-Newton method is dependent on the size and computation time of each block. The best case occurs when the size of block i is always smaller than the size of block $i + 1$, because the updated values from the block of ith and $< ith$ are always used for solving

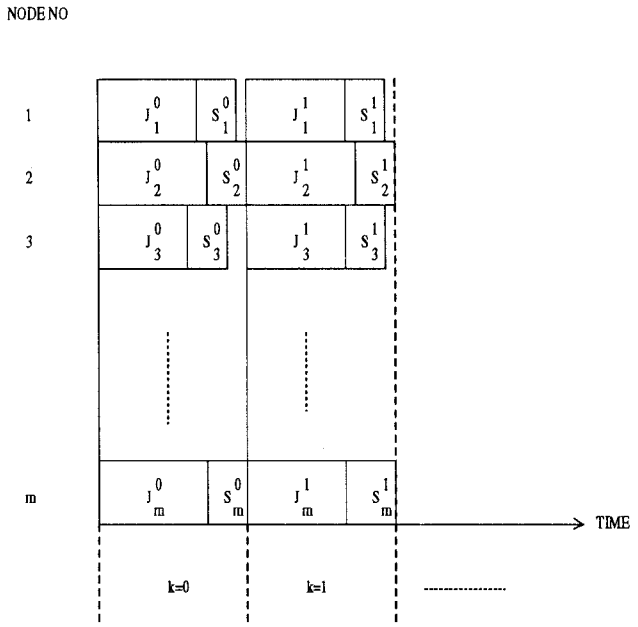

Figure 5: Data flow of a standard Jacobi-Newton method.

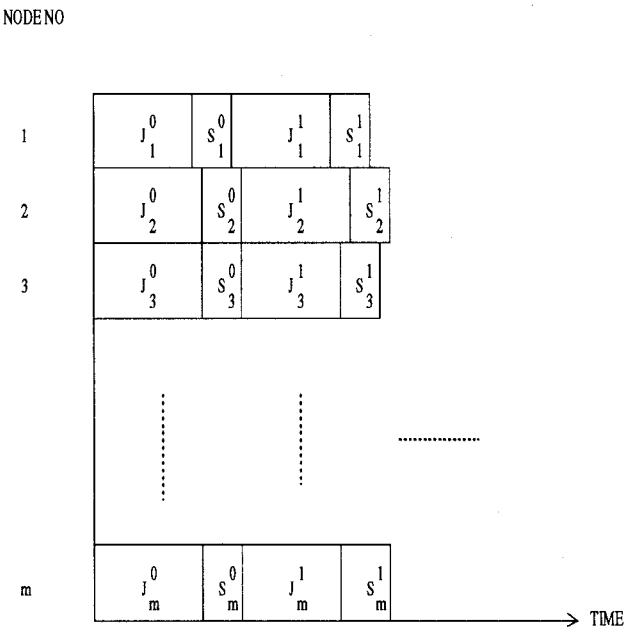

Figure 6: Data flow of the asynchronous Jacobi-Newton method.

the $i+1th$ block. The slow case occurs when the method is equivalent to the standard Jacobi method where no updated values from the blocks of ith or $< ith$ are used for solving block $i+1$. The worst case occurs when the size of block i is always larger than the size of block $i+1$, so the updated values from the block of ith and $< ith$ may never be used for solving block $i+1$; for all the cases, $i = 1, ..., m$. We expect, on the average, that the performance would be slightly improved. One example of the data flow of such a asynchronous Jacobi-Newton method is shown in Figure 4.

4.3 Nonblocking implementation of the asynchronous methods

A modern distributed memory multicomputer, such as the Intel iPSC/860 and the Intel Paragon, provides at least two basic types of message-passing primitives: blocking and nonblocking protocols. A blocking protocol will cause the sender to block until the message is received. A nonblocking protocol will continue other work after sending a message. In addition, the nonblocking protocol on the Intel machines allows for a special message type to be used that delivers messages directly to a user's buffer, rather than to a system buffer. The time for copying the content from the system buffer to the user buffer is eliminated. This effective nonblocking message delivery can be guaranteed only when the user buffer and the nonblocking receive instruction are available prior to the arriving message.

The nonblocking protocol (isend()/irecv()) provided by the Intel iPSC/860 multicomputer is used to implementing these asynchronous methods. To simplify the explanation of the implementation, we assume that each processor computes one block, i.e. the number of processors is equal to the number of blocks. Therefore, the user's buffer in processor i contains the variables of blocks 1 to $i-1$. The blocks in a user's buffer for a processor must be updated at the time when the processor does the function evaluation for the solution of the local block variables. For example, processor 1 does not have a user buffer because it does not receive any new updated values. Processor 2's user buffer contains variable blocks 1 which will be updated by processor 1. Processor 3's user buffer contains variable block 1 and 2 which will be updated by processor 2. (Processor 2 will pass the updated block 1 and its own updated block to the user's buffer in processor 3. Message-passing of large size in a single packet is more effective than one of small size in multiple packets.) Each processor will not stop inner iterations until the user's buffer is filled. The number of inner iterations in each processor is dynamically changed.

5 Computing results on the Intel iPSC/860

The single column simulation problem, written as 88 FORTRAN subroutines (6600 lines of code), generates a sparse nonlinear system of 1414 variables, including computation of initial values and of many parameters. In order to discover the available parallelism, the Unix *gprof* utility on a Sun SPARC workstation was used to obtain an execution profile of the sequential program. Figure 7 gives the execution distribution structure of the program. If the 21.4% of data initialization and input time is not counted, 96% of the total computing time will contribute on solving the sparse nonlinear system.

The problem was transformed into a lower block triangular system with the aid of the MA28 routines. However, the

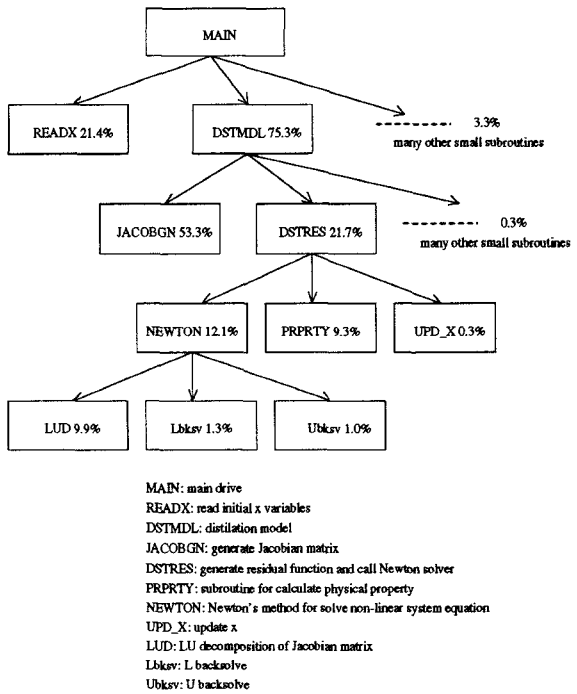

Figure 7: Profile of the sequential simulation

block structure is not well balanced for a parallel distribution, where many on the diagonal are small. Based on the original output of the MA28, we defined 31 square blocks on the diagonal with roughly equal sizes. Some of them are still very sparse, and some of them have only diagonal variables/blocks.

The multiple distillation column series simulation connects 8 identical single columns together in one direction. The block triangular structure is well balanced because each diagonal block represents the sparse system of each single column in the series simulation. The connections of these columns are additional equations that set the outputs of a column to the inputs of its forward neighbor column. These equations are placed in the bottom of the triangular system.

The distributed memory multicomputer we used for solving the problem is the Intel iPSC/860 hypercube. The system has up to 128 nodes, comprised of the i860 40 MHz processor, 8 MB of memory, an 8 KB data cache and a 4 KB instruction cache. The processor speed is extremely fast compared with the bandwidth of the network.

In the synchronous Gauss-Seidel-Newton method, each processor first evaluates one or more Jacobian blocks and factors these Jacobian blocks using LU decomposition. This process is perfectly parallel. After the parallel computing part, the function evaluation and one, or more than one solution blocks are computed in sequential. Inter-processor communications are involved for processor $i+1$ to receive the updated block variable values from processor i $(i = 1, ..., m)$.

Methods	Number of iterations
Syn. Jacobi-Newton	25
Asyn. Jacobi-Newton	23
Syn. Gauss-Seidel-Newton	13
Asyn. Gauss-Seidel-Newton	10

Table 3: Number of iterations of the different synchronous and asynchronous methods for the single column simulation.

The controlled asynchronous Gauss-Seidel-Newton method described in the last section was implemented on the Intel iPSC/860 machine. Dynamic inner iterations were applied in each processor to control the correct order of the asynchronous computing.

The Jacobi-Newton method for solving the problem was also implemented. The asynchronous Jacobi-Newton method was implemented by simply deleting the barrier at the end of each iteration of the Jacobi-Newton method.

The original simulation routines provide a set of reasonably good initial values. In all cases, the stopping criterion was to reduce the total function l_2 norm below 10^{-5}. Table 3 gives the number of iterations needed by each method to satisfy the stopping criterion. Figure 8 reports the two groups of performance results and comparisons between the synchronous and asynchronous approaches. The top two curves are the timing results for the Jacobi-Newton method. The bottom two curves are the timing results for the Gauss-Seidel-Newton method. The asynchronous approach applied to each method is effective and improves parallel performance. Figure 3 also indicates that the single column simulation problem is not big enough for effective executions on the iPSC/860 with more than 16 processors where little speedup could be gained.

The multiple distillation column series simulation was mapped to 8 processors. Each diagonal block represents an individual distillation column. We compare the parallel computing results with the sequential simulation computed by Gauss-Seidal-Newton's method on a Sun SPARC 10 workstation. Table 4 lists the numbers of iterations of 4 different methods for computing the simulation. The comparative performance results are plotted in Figure 9. Significant computing time is reduced on the parallel machine. The asynchronous approach is again more effective than the synchronous approach for this simulation.

In both single and multiple column simulations, the controlled asynchronous Gauss-Seidel-Newton method spent 3 times less iterations than the synchronous Gauss-Seidel-Newton method for the same stopping criterion. The results confirm that the inner iterations in Gauss-Seidel-Newton method for solving nonlinear block triangular systems are faster in convergence, which has been theoretically proved in [6].

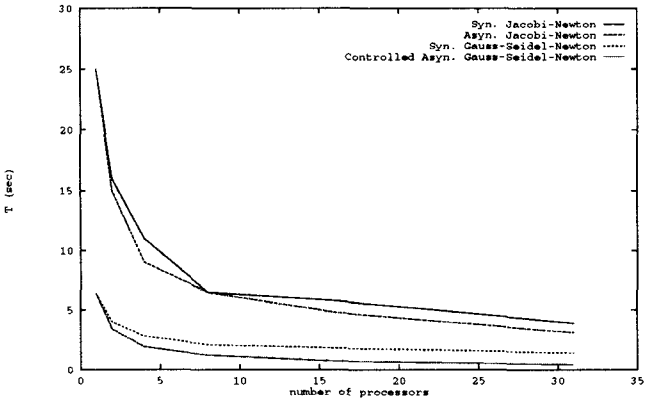

Figure 8: Performance results and comparisons between the synchronous and asynchronous Gauss-Seidel-Newton methods for computing the single column simulation.

Methods	Number of iterations
Syn. Jacobi-Newton	26
Asyn. Jacobi-Newton	25
Syn. Gauss-Seidel-Newton	18
Asyn. Gauss-Seidel-Newton	15

Table 4: Number of iterations of the different synchronous and asynchronous methods for the multiple column simulation.

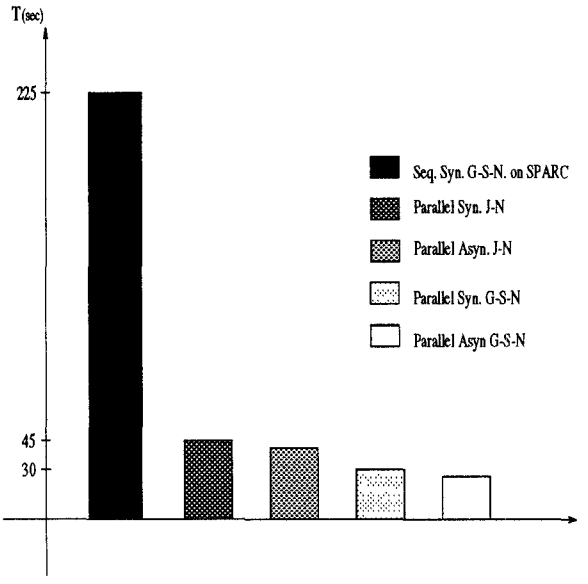

Figure 9: Performance results and comparisons between the synchronous and asynchronous Gauss-Seidel-Newton methods for computing the multiple column simulation.

6 Summary

Efficiently parallelizing large application programs for execution on multiprocessor architectures is a key issue for the development of high-performance computing. We believe that a structured approach that decomposes an application system into certain forms is an effective tool for developing parallel methods in various scientific computing applications. The triangular decomposition method is an example for transforming a large sparse nonlinear system into efficient forms for parallel processing. Our experience with the oil refining simulation indicates that algorithm designers should have a better understanding of how a computation can be carried out in a cost-effective manner on a multiprocessor architectures. For example they may need to overlap communication and computation using a nonblocking message-passing scheme. On the other hand, the architecture designer should also have a better understanding of the true nature of scientific computations so as to build more efficient multiprocessor architectures.

We have shown experimentally that the parallel iterative methods presented here are efficient. There appear to be effective ways to parallelize the solution of sparse nonlinear systems of this type. Our experimental results also demonstrate that asynchronous computing on the Intel iPSC/860 can improve the performance by overlapping communication and computations. The current work involves applying the triangular decomposition method to solve large scale sparse nonlinear systems from different applications on different multiprocessor architectures.

Acknowledgement

The author is grateful to J. E. Dennis Jr. and J. M. Martínez for collaboration on the numerical analysis work regarding triangular decomposition methods, and for their encouragement and support for this application. H. Wang did partial programming implementations on the Intel iPSC/860. The author also thanks J. G. Renfro for providing the simulation problem and instruction in using the programs. Finally, the author wish to thank N. Wagner for carefully reading the paper and contributing many valuable suggestions and comments.

This work has been supported in part by the National Science Foundation under research grants CCR-9008991 and CCR-9102854, and instrumentation grant DUE-9250265, by the U.S. Air Force under Agreement FD-204092-64157, by a grant from the Cray Research and by a grant from the Shell Oil Foundation. This work was partially conducted while the author visited the Center for Research on Parallel Computation at Rice University. The parallel experiments were performed on the Intel iPSC/860 machines in the Center at Rice University and in the Intel Supercomputer Systems Division.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation, Numerical Methods*, Prentice Hall, Englewood Cliffs, N.J., 1989.

[2] E. D. Chajakis and S. A. Zenios, "Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization", *Parallel Computing*, Vol. 17, pp. 873-894, 1991.

[3] T. F. Coleman and J. J. Moré, "Estimation of sparse Jacobian matrices and graph coloring problems," *SIAM J. Numer. Anal.* **20**, pp. 187-209, 1983.

[4] A. R. Curtis, M. J. D. Powell and J. K. Reid, "On the estimation of sparse Jacobian matrices," *J. Inst. Maths. Applics*, **13**, pp. 117-120, 1974.

[5] J. E. Dennis, Jr., J. M. Martínez and X. Zhang, "Parallel block triangular decompositions for solving sparse nonlinear systems of equations," *Proceedings of the 5th SIAM Conference on Parallel processing for scientific Computing*, SIAM Press, pp. 168-173, 1992.

[6] J. E. Dennis, Jr., J. M. Martínez and X. Zhang, "Triangular decomposition methods for solving reducible nonlinear systems of equations," to appear in *SIAM J. Optimization.*

[7] J. E. Dennis, Jr. and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations,* Prentice-Hall, Englewood Cliffs, 1983.

[8] I. S. Duff, "MA28 – a set of Fortran subroutines for sparse unsymmetric linear equations," *Report*, AERE R8730, HMSO, London, 1977.

[9] I. S. Duff, A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, 1989.

[10] H. Fischer and K. Ritter, "An asynchronous parallel Newton method", *Mathematical Programming*, **42**, pp. 363-374, 1988.

[11] G. A. Geist and C. H. Romine, "LU factorization algorithms on distributed memory multiprocessor architectures," *SIAM J. Sci. Stat. Comput.*, **9**, pp. 639-649, 1988.

[12] C. D. Holland, *Fundamentals and Modeling of Separation Processes*, Prentice-Hall, 1975.

[13] G. Li and T. Coleman, "A new method for solving triangular systems on distributed-memory message-passing multiprocessors," *SIAM J. Sci. Stat. Comput.*, **10**, pp. 382-396, 1989.

[14] S. A. Zenios and M. C. Pinar, "Parallel block-partitioning of truncated Newton for nonlinear network optimization", *SIAM J. Sci. Stat. Comput.*, **13**, pp. 1173-1193, 1992.