Parallel Implementations of Direct Solvers For Sparse Systems of Linear Equations on PVM System and nCUBE Machine[1]

Xuyang Li
Civil Engineering Department / 4190 Bell Engineering Center
Azhar Maqsood, James M. Conrad[2]
Computer Systems Engineering / 313 Engineering Hall
University of Arkansas, Fayetteville, AR 72701
E-mail: {xl0, am02, jmc3}@engr.engr.uark.edu

Abstract - The basic problems in developing parallel direct solvers of sparse systems of linear equations are discussed in this report. These problems, including the storage schemes of sparse matrices, the running environment of the programs, and the parallelization of the sequential algorithms, are handled while keeping current parallel computer architectures in mind. The behavior of the parallel machines used for the underlying problem is also discussed in this report. The problem is applied over two different parallel environments: the Parallel Virtual Machine (PVM) and the nCUBE machine (Hypercube). Test results for both versions are analyzed in terms of the machine structure and algorithm design.

1. Introduction

Solving large systems of equations in which a majority of the coefficients are zero is very important in scientific research and engineering computing. Systems of equations like these, called sparse systems of equations, are often encountered in numerical deductions of problems for which analytical solutions are very hard to obtain. Examples of such problems include solving partial differential equations by numerical methods, multi-dimensional spline interpolations and finite element method calculations in weather forecasting, computer aided design and computer assisted manufacturing, fluid dynamics calculations, and simulation of natural behaviors. Some problems result in systems of linear equations with coefficient matrices of special structures, others result in systems of linear equations with coefficient matrices of random structures. It is often inefficient and sometimes impossible to solve sparse systems of equations by using dense matrix system solvers because the memory occupied by the zero elements of the matrix is too large to handle. Solving these systems of equations involve more complex algorithms and data structures than their dense counterparts.

A system of n linear equations has the following form:

$$\begin{cases} a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0, \\ a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_1, \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}. \end{cases}$$

In matrix notation, this system can be represented by

$$Ax = b,$$

where A is the $n \times n$ matrix of coefficients such that

$$A[i, j] = a_{i,j},$$

b is an $n \times 1$ vector

$$[b_0, b_1, \ldots, b_{n-1}]^T$$

and x is the desired $n \times 1$ solution vector

$$[x_0, x_1, \ldots, x_{n-1}]^T.$$

The matrix A is considered sparse if a computation involving it can utilize the number and location of its nonzero elements to reduce the run time over the same computation on a dense matrix of the same size.

Although there are many good algorithms and programs on sequential computers that can be used to solve sparse linear systems, they have limited success in solving sparse systems of linear equations on parallel computers [1, p.454]. The reasons for this are twofold. The iterative methods for sparse linear systems are fast if they converge. The problem is they are sometimes not convergent. The direct methods, on the other hand, are very stable, but they involve a large amount of communication among processors on distributed-memory parallel computers. In this report, different aspects of implementing parallel algorithms of sparse linear systems are discussed. The storage scheme, algorithm, and implementation details of a direct method are given. Finally, the results comparing with its dense counterpart and its sequential implementation are also given.

[1] Proceedings of the 1996 Arkansas Computer Conference, Sercy, AR, pp. 52-63, March 1996. This version has been reformatted.
[2] Currrently at UNC Charlotte, 9201 University City Blvd, Charlotte, NC 28223, jmconrad@uncc.edu

2. Direct Methods Versus Indirect Methods

There are two totally different kinds of methods for solving systems of linear equations. They are direct methods and indirect methods. The indirect methods or iterative methods are techniques to solve systems of equations of the form $Ax = b$ that generate a sequence of approximations to the solution vector x. In each iteration, the coefficient matrix A is used to perform a matrix-vector multiplication. The number of iterations required to solve a system of equations with a desired precision is usually data dependent, hence, the number of iterations is not known prior to executing the algorithm. Iterative methods do not guarantee a solution for all systems of equations, even though the systems are not singular. This means the iterative methods may be divergent when applied to certain data. This was the main reason that direct methods were chosen to be used in the implementation presented in this report.

There is another reason why the direct methods were used in the implementation here: the iterative methods are sometimes for special purposes. This means that one iterative method may be only suitable for solving a special kind of systems of equations, such as those resulting from finite element method calculations. For example, the conjugate gradient method and the preconditioned conjugate gradient algorithm are only suitable for solving large sparse systems of linear equations with symmetric positive definite matrices [1, pp.433 - 436]. Direct methods are useful for solving sparse linear systems because they are general and robust. Although there is substantial parallelism inherent in sparse direct methods, only limited success has been achieved to date in developing efficient general-purpose parallel formulations for them. Developing efficient general-purpose parallel formations of direct methods for unstructured or random sparse matrices is currently an active area of research. Although all of these methods are based on Gaussian elimination (for general matrices) and Cholesky factorization (for symmetric positive definite matrices), their parallel formulations can be quite complicated.

Here, the Gaussian elimination with partial pivoting method is implemented on parallel computers. The implementations can be used for solving general sparse linear systems.

3. Parallel Computers Used For Implementation

Parallel computers have different structures and different software environments. The Parallel Virtual Machine (PVM) was chosen as the first kind of parallel computing environment for the implementation. The main reason for choosing the PVM is that it is a system with multi-architecture compatibility. The PVM is not a specific machine. Rather, it is a software environment. PVM permits a network of heterogeneous UNIX computers to be used as a single large parallel computer. Thus large computational problems can be solved by using the aggregate power of many computers. These machines are often the most popular computers now in use, such as the SUN SPARCstations, the CRAY supercomputers, the 80386/80486 UNIX box, the Thinking Machines, the DEC Alpha, and the micro VAX. PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. Applications, written in C or FORTRAN, can be parallelized by using message-passing constructs common to most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. PVM supports heterogeneity at the application, machine, and network level. So PVM allows application tasks to exploit the architecture best suited to their solution. All the data conversion that may be required if two computers use different integer or floating point representations are handled by PVM. Even machines that are interconnected by a variety of different networks can be used by PVM. Programs running on PVM do not need to know the details of communication. The library functions used on PVM system for different architectures have the same syntax. Because of these, programs written for PVM system can be easily ported from one architecture to another without modifications. They can be run simultaneously on different machines. This flexibility makes PVM one of the most powerful parallel computing environments. However, the PVM system is not perfect. Because the PVM system mainly uses networks to transmit data from machine to machine, the performance of the system is largely dependent on the performance of the networks. This is really a consequence of its flexibility. This architecture, as the results shown later, limits the performance of the parallel implementation of Gaussian elimination method [2, p.1 - 5].

Another kind of parallel computer used for the implementation of algorithms was the nCUBE (a hypercube machine). The nCUBE is a high-performance parallel computer. The big advantage of the nCUBE over PVM is that the communication among processors on the nCUBE is highly efficient. nCUBE machines are also distributed-memory machines.

4. Storage Scheme For Sparse Matrices

It is customary to store an $n \times n$ dense matrix in an $n \times n$ array. However, if the matrix is sparse, storage is wasted because a majority of the elements of the matrix are zero and need not be stored explicitly. If the positions and values of all the nonzero elements of the matrix are known, then the whole matrix is known. It is a common practice to store only the nonzero elements and to keep track of their locations in the matrix. Currently there are many storage schemes that can be used to store and manipulate sparse matrices [1, pp.409 - 412]. These specialized schemes not only save storage but also yield computational savings. Since the locations of the nonzero elements in the matrix are known explicitly, unnecessary multiplications and additions with zero can be avoided. Each of these schemes is developed for specific purposes. They are suitable for different implementations on machines with different architectures. Some data structures are more suitable for a parallel implementation than others. There is no single best data structure for storing sparse matrices. In the implementation presented here, a special storage scheme is used. This storage scheme is characterized by the employment of a group of single direction linked lists and a row pointer vector.

In this scheme, each row of the matrix is represented by a single direction linked list. One element of the matrix in a row is represented by a node in the linked list. The node is represented by the following data structure using C notation:

```
struct row {
  float elem;
  int col;
  struct row * next;
}
```

The field `elem` of `struct row` is the value of a nonzero matrix entry in the current row, while the field `col` of `struct row` is the column number of this entry. The field `next` of `struct row` is a pointer that points to the node representing the next nonzero element in the same row. The field `next` of the last node of this row is set to `NULL`.

During initialization and computation, the order of the nodes in the linked list is kept so that the `col` is always in ascending order along the direction of the linked list. This order keeps the searching of an element with a certain `col` number fast.

All the rows in the sparse matrix are bound together by a vector of structure pointers. Each element of this vector is a pointer that points to the first element of the linked list that represents a row in the matrix with its row number equal to the index of this element in the vector. So the type of this vector is as follows:

```
struct row **
```

in which `struct row` is defined above. For the example matrix shown in Figure 1, the schematic representation of it is shown in Figure 2.

In the implementation presented later, the memory occupied by the vector of pointers is allocated at the time the number of equations n (that is, the number of rows in the coefficient matrix) is known. The memory is allocated so that the number of elements in the vector is exactly n. Memory occupied by the vector will not change after that.

Note that each row of the sparse matrix of a non-singular system of linear equations must have at least one nonzero element (otherwise, it would be singular), so there should be no element of the above vector that has the value `NULL` during the whole computational process. This means that the above storage scheme wastes no memory in the vector for non-singular systems of equations. Also, in the linked list representation of rows, sequential search for an element is needed due to the sequential property of the linked list, and because the storage order of the elements is maintained as described above, the time needed for accessing an element in a row is **O**(s/2), where s is the number of nonzero elements in the row. Memory needed for storing a sparse matrix using the above scheme is calculated as:

N*sizeof (struct row) + n*sizeof (struct row *), where N is the number of nonzero elements in the matrix, and n is the number of rows in the matrix. This number is always changing during processing, because dynamic memory allocation is used to keep memory usage the most economical. In the implementation, an element in the matrix is considered to be zero if the absolute value of it is less than or equal to a preset small positive number (user defined). So in the process of elimination, newly produced nonzero elements are added to the matrix, while all the newly produced zero elements are removed from the matrix.

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.0 & 0.0 & 3.0 \\ 4.0 & 5.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 6.0 & 0.0 & 0.0 & 0.0 & 8.0 \\ 9.0 & 0.0 & 0.0 & 10.0 & 11.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 14.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Figure 1 - A Sparse Matrix

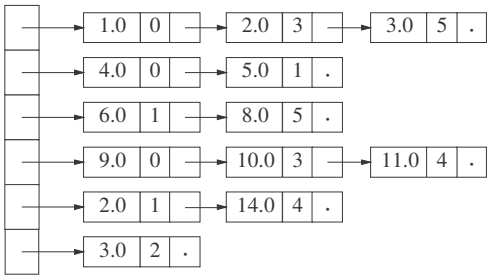

Figure 2 - Sparse Matrix Storage Scheme

5. Basic Algorithms

A system of equations $Ax=b$ is usually solved in two stages. First, through a series of algebraic manipulations, the original system of equations is reduced to an upper-triangular system of the form

$$\begin{cases} u_{0,0}x_0 + u_{0,1}x_1 + \cdots + u_{0,n-1}x_{n-1} = y_0, \\ u_{1,1}x_1 + \cdots + u_{1,n-1}x_{n-1} = y_1 \\ \vdots \\ u_{n-1,n-1}x_{n-1} = y_{n-1}. \end{cases}$$

This can be written as $Ux=y$, where U is a matrix in which all subdiagonal entries are zero. That is $U[i,j] = 0$ if $i > j$, otherwise $U[i, j] = u_{i,j}$. U is called an upper-triangular matrix. This stage is called factorization. In the second stage of solving a system of linear equations, the upper-triangular system is solved for the variables in reverse order from x_{n-1} to x_0 by a procedure known as back-substitution. The basic algorithm used in this implementation is Gaussian elimination with partial pivoting. The sequential version of it has several nested loops. Figure 3 shows the Gaussian elimination with partial pivoting algorithm used as the basis of parallelization.

```
Procedure
GAUSSIAN_ELIMINATION_W_PARTIAL_PIVOTING(A, b, n)

var
  marked : array [0 .. n - 1] of boolean;
  pivot : array [0 .. n - 1] of 0 .. n - 1;
  i, j, k, picked : integer;
  tmp, tmp1 : real;

begin
  for i: = 0 to n - 1 do
  begin

    { pivoting operations }
    tmp1 := 0;
    for j := 0 to n - 1 do
    begin
      if ((not marked[j]) and (ABS(A[j, i]) >
        tmp1)) then
      begin
        tmp1 := ABS(A[j, i]);
        picked := j;
      endif;
    endfor;
    tmp1 := A[picked, i];
    marked[picked] := true;
    pivot[picked] := i;

    { elimination operations }
    for j := 0 to n - 1 do
    begin
      if (not marked[j]) then
        begin
        tmp := A[j, i] / tmp1;
        b[j] := b[j] - b[picked] * tmp;
        for k := i + 1 to n - 1 do
        begin
          A[j, k] := A[j, k] - A[picked, k] * tmp;
        endfor;
      endif;
    endfor;
  endfor;
  for i := 0 to n - 1 do
  begin
    if (not marked[i]) then
    begin
      pivot[i] = n - 1;
    endif;
  endfor;
end GAUSSIAN_ELIMINATION_W_PARTIAL_PIVOTING
```

Figure 3 - Sequential Algorithm of Gaussian Elimination with Partial Pivoting

After the full matrix A has been reduced to an upper-triangular matrix U, the back-substitution operation is conducted to determine the vector x. The sequential back-substitution algorithm for solving the upper-triangular system of equations $Ux = y$ is shown in Figure 4.

```
procedure BACK_SUBSTITUTION (U, y, pivot, n)

var
  i, j, row : integer;

begin
  for i := n - 1 downto 1 do
  begin
    for row := 0 to n - 1 do
    begin
      if (pivot[row] = i) then
      begin
        exit for;
      endif;
    endfor;

    { solution for i'th variable }
    y[row] := y[row] / U[row, i];

    { back-substitute }
    for j := 0 to n - 1 do
    begin
      if (pivot[j] < i) then
      begin
        y[j] := y[j] - y[row] * U[j, i];
      endif;
    endfor;
  endfor;
  for row := 0 to n - 1 do
  begin
    if (pivot[row] = 0) then
    begin
      exit for;
    endif;
  endfor;
  y[row] := y[row] / U[row, 0];
end BACK_SUBSTITUTION.
```

Figure 4 - Back-substitution Algorithm

The Gaussian elimination and back-substitution algorithms were originally designed for solving dense matrix systems of equations. In order to save execution time, unnecessary memory movements are avoided by using the array `pivot` and `marked` in the algorithm. Rather than assigning zero to the eliminated elements in matrix *A*, the algorithm simply leaves them unchanged because they are not used in the following steps. All the unnecessary assignments to elements of matrix *A* are avoided in this way. For sparse matrix systems of equations, the algorithm should be modified so that the storage occupied by newly produced zero elements of matrix *A* be released to save memory. Sparse systems of equations often have very large sparse matrices, so the above modification is necessary.

6. Parallelization of The Algorithms

6.1 General Criteria And Data Partitioning

6.1.1 Ordering of The System of Equations

The characteristics of the machines used in the implementation should be considered when parallelizing the above algorithms. Four steps are considered in the parallization of the above algorithms. They are ordering, symbolic factorization, numerical factorization, and solving a triangular system. Some parts of these steps may be omitted when considering the implementation of the algorithms on specific machines. Ordering is an important phase of solving a sparse linear system because it determines the overall efficiency of the remaining steps. The aim of it is to rearrange the rows of the original coefficient matrix so that the permuted matrix leads to a faster and more stable solution. The numerical stability of the solution is increased by ensuring that the diagonal elements or pivots are large compared to the remaining elements of their respective rows. This is already included in the sequential algorithm and needs to be parallelized. The ordering criteria for obtaining a faster parallel solution are very complex. This process needs large amount of changing positions of rows in the matrix, so for distributed-memory machines such as PVM, this could use a large proportion of the whole computation time. The benefits resulting from ordering will be fully surpassed by the waste of time in communication, because data transmission in PVM system is crucial to the overall performance of the implementation. Considering this factor, the implementations presented here only emphasize the enhancement of the numerical stability of the solution and contain a partial pivoting process. A large amount of communication is avoided by eliminating a full ordering process.

6.1.2 Data Partitioning And Factorization of The System of Equations

Due to the availability of very fast serial algorithms and the high data-distribution cost involved in parallelizing them, implementations of parallel symbolic factorization on message-passing computer (distributed-memory machines) tend to be inefficient. Moreover, symbolic factorization is often performed once and then several systems with the same sparsity pattern are solved, amortizing the cost of symbolic factorization over all the systems [1, p.458]. On the other hand, the programs presented here are used to solve systems of equations that usually have no relation with each other. So the benefits of using symbolic factorization in these programs are not obvious. Because of this, the symbolic factorization to the system was not conducted in the programs.

In order to conduct numerical factorization in parallel, the coefficient matrix needs to be mapped onto all the processors. This involves partitioning the matrix into small blocks so that each block can be assigned to a specific processor. It is important to choose an appropriate data-mapping scheme for distributed-memory machines. For the PVM and nCUBE machines, it is best to use the block-striped partitioning of the matrix because this can reduce the communication costs. In this partitioning scheme, the matrix is divided into groups of complete rows, and each processor is assigned one such group. Each group contains contiguous rows. For example, a matrix with 16 rows can be divided into four groups and each group is assigned to one of four processors. This is shown in Figure 5.

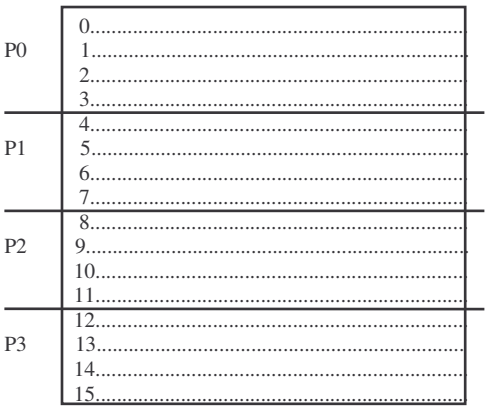

Figure 5 - Block Striping Partitioning of a 16 x 16 Matrix on 4 processors

The partition is made so that the difference between the number of rows contained in any two groups is at most one. This distributes the data evenly to all the processors.

Apart from the coefficient matrix, the right-hand side vector of the system of equations is also partitioned by the same partitioning scheme. Each processor has its own array `pivot` and array `marked`. In addition to the processors that contain the data, a separate processor is used as the master node that controls the whole computational process.

6.2 Parallelizing The Algorithms

6.2.1 Parallelizing The Pivoting Process

The pivoting process is somewhat complicated when trying to run in parallel. First, each processor or node searches for the row that has the maximum absolute element value in the current column. This element is called the local pivot. The local pivot together with the ID of the node that contains it is then sent to the master node. The master node collects all the local pivots and the corresponding node IDs and compares the local pivots. The pivot with the largest absolute value (which is the TRUE pivot) is then found, and the ID of the node that has this pivot is broadcast to all the slave nodes. Each node then compares the received ID with its own, and the one that has the pivot row broadcasts the entire pivot row to other slave nodes.

In both the PVM implementation and the nCUBE implementation, each slave node searches their local pivot in parallel. In the PVM implementation, the master node collects the local pivots sequentially. On the nCUBE, however, the special structure of the machine is considered so that the pivoting is conducted in a faster way. This will be discussed later in this report. The sequential part limits the speedup of the PVM implementation.

6.2.2 Parallelizing The Elimination Process

The parallel elimination process is relatively simple compared to the pivoting process. It is also more efficient. All the nodes eliminate their own part of data simultaneously using the pivot row received. Because there is no swapping of rows in the pivoting and elimination process, the order of the rows has been random from the beginning. The pivoting and elimination loads are evenly distributed to all processors.

Combining the above two parts, the parallel pivoting and elimination processes are shown in Figure 6.

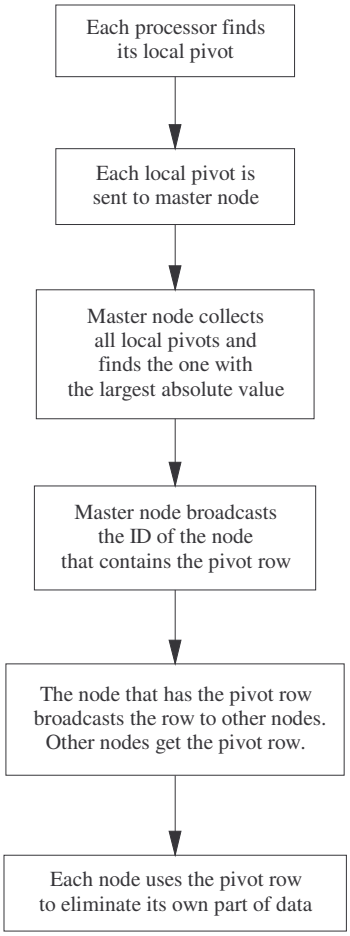

Figure 6 - Parallel Pivoting and Elimination Processes

6.2.3 Parallelizing The Back-substitution Process

Back-substitution is done in parallel, too. The back-substitution process begins from the last variable. The order is controlled by the master node. Because the order of the rows in the matrix is random, each processor needs to search for the current variable simultaneously. The node that has found the current variable broadcasts it to all the other nodes. The next step is to back-substitute the variable simultaneously by all the processors. This process repeats until all the variables are found. The parallel back-substitution process is shown in Figure 7.

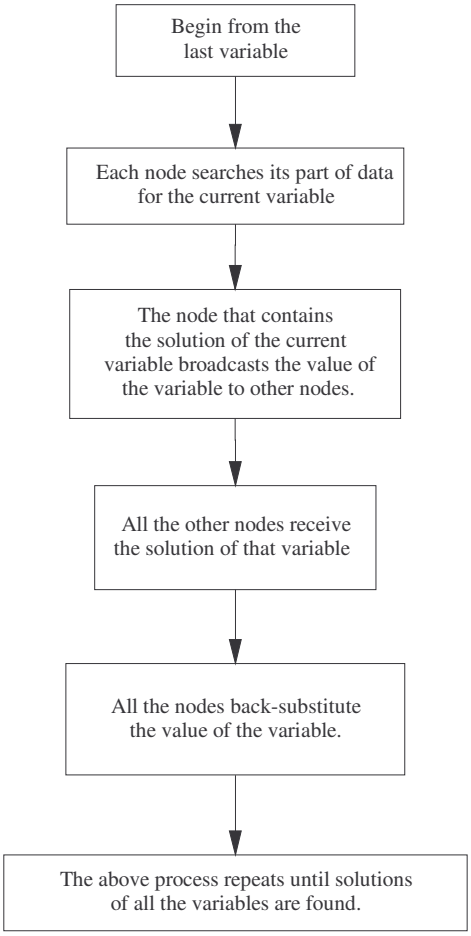

Figure 7 - Parallel Back-substitution Process

6.2.4 Parallelizing The Input And Output Processes

It is natural to consider parallelizing the input and output processes because they are usually time consuming. However, from an analysis of the time used in each part, it was found that the input and output processes only used a very small fraction of time of the whole process. Considering this and the sequential characteristics of the input and output files, the input process was only parallelized to a minor degree, and the output process was done sequentially.

7. Implementation of Algorithms on PVM and nCUBE

7.1 Implementation on The PVM System

The PVM version of the direct solver of systems of linear equations has two separate modules, called the master module and the slave module. The master module is run on the master processor and the slave module is run on the slave processors. The master module is responsible for reading command line parameters that include the number of processors, the input file name, and the output file name. It is also responsible for launching the slave module on each slave processor. This is done by calling the PVM routine `pvm_spawn()`. The master module distributes the work load evenly to the slave processors. The most important work the master module does is to help finding the pivot rows in the elimination process. Functions `pvm_initsend()`, `pvm_pkint()`, `pvm_pkfloat()`, `pvm_mcast()`, and `pvm_send()` are used to send messages to other processors. Functions `pvm_recv()`, `pvm_upkint()`, and `pvm_upkfloat()` are used to receive messages from other processors. The master module is also responsible for collecting the final results and writing them to the output file.

The slave module first reads the corresponding part of data from the input file according to their processor ID and then conducts the elimination and back-substitution. Finally, the solutions are sent to the master processor at the request of the master processor.

7.2 Implementation on The nCUBE System

Implementation on the nCUBE system is quite similar to the implementation on the PVM system. However, there is no separate master processor in this implementation. The first node is used as the master processor. In the pivoting process, the master processor collects the local pivot by using a binary message passing scheme. This scheme greatly reduces the steps needed for pivoting, especially when large numbers of processors are used. In each step of this scheme, message passing is conducted simultaneously between several pairs of processors which are direct neighbors to each other. That is, the Hamming distance between the two processors is one. Due to the characteristics of the hypercube architecture, passing messages from one node to its direct neighbor is faster than passing messages between nodes that are not direct neighbors. This scheme is best described by an example. Suppose eight processors are used in the computation. They are numbered 0 through 7. Processor 0 is the master processor. The message passing process contains three steps. They are shown in Figure 8.

```
Step 1:  7->6, 5->4, 3->2, 1->0
Step 2:     6---->4,    2---->0
Step 3:           4---------->0
```

Figure 8 - A Binary Message Passing Scheme on nCUBE

7.3 Program Interface

7.3.1 Command Line Parameters

The PVM version of the program is launched in the following way (suppose the master program has the name "gpmaster"):

```
gpmaster <# of processors> <input file name>
<output file name>
```

The nCUBE version of the program is launched by the `xnc` utility as follows (suppose the name of the program is "gs"):

```
xnc -d <dimension of subcube> gs <input file
name> <output file name>
```

7.3.2 Input And Output File Format

The formats of the input and output file for both the PVM version and the nCUBE version are the same. The input file format is as follows. The first number in the file is the number of equations. Following that are the nonzero elements of the coefficient matrix and the right-hand side of the equations. Each nonzero element of the matrix is represented by a triple of numbers: the row number of this element, the column number of this element, and the element itself. The right-hand side of each equation is represented by the following triple of numbers: the row number, -1, and the value itself. These triples can be in any order in the input file. For example, a system with coefficient matrix A shown in Figure 1 and right-hand side $b = [3.0, -2.0, 4.0, 1.0, 2.5, 1.4]^T$ can be represented by the following file:

```
6
0 0 1.0
0 3 2.0
0 5 3.0
0 -1 3.0
1 0 4.0
1 1 5.0
1 -1 -2.0
2 1 6.0
2 5 8.0
2 -1 4.0
3 0 9.0
3 4 10.0
3 5 11.0
3 -1 1.0
4 1 2.0
4 4 14.0
4 -1 2.5
5 2 3.0
5 -1 1.4
```

The output file format is quite simple. It is shown below (the ellipses and n are replaced by proper numbers in real files):

```
x[0] = ...
x[1] = ...
...
x[n-1] = ...
```

8. Results

The programs were run to solve systems of equations resulted from practical problems. The systems of equations generated in the process of solving partial differential equations by using bivariate cubic spline functions [3, p.9 and 5, p.213] were used in the testing. These systems of equations are typical sparse systems. In addition to these equations, other systems of equations generated by a special program were also tested.

The PVM version was tested on SUN SPARCStations. For a system of 100 equations, the result is shown in Table 1. The number of processors and the corresponding execution time is listed in the table. In this case no speedup was achieved in the testing. A system of 600 equations was also used for testing. The result is shown in Table 2. There is little speedup in this case. In both cases, the execution time increases rapidly as the number of processors increases.

Table 1 - Result of PVM Version Running For a System of 100 Equations

Number of Processors	*Execution Time (Seconds)*
1	6
2	9
3	11
4	13
5	17
6	16
7	16
8	20
9	25
10	28

The nCUBE version was tested on the SUNCUBE machine at Texas A&M University. In this case, the corresponding program for dense matrix systems was also tested. The results of both sparse solver and dense solver are shown in Table 3.

Table 2 - Result of the PVM Version Running For a System of 600 Equations

Number of Processors	*Execution Time (Seconds)*
1	71
2	54
3	63
4	74
5	96

Table 3 - Result of the nCUBE Version Running For a System of 100 Equations

Dimension of Subcube	*Execution Time (Sparse) (nano-seconds)*	*Execution Time (Dense) (nano-seconds)*
0	1233520	3721190
1	1415780	3218430
2	1562920	3269990
3	2090550	3722900
4	6680630	8838780
5	10048100	17154800

9. Discussion of the Results

The results showed that for small systems of equations, it is hard to get any speedup when running these programs. This is because the communication cost for small systems is very high. That is why there is no speedup in most of the cases tested. From Table 3 it is seen that the sparse system solver is more efficient than the dense solver for sparse systems of equations. This is an expected result. From the results shown in Table 1 and Table 2, it is obvious that the performance of the program is improved for larger sparse systems. It can be inferred that the programs will show apparent speedup for larger systems, such as systems of thousands of equations. Due to the limit of disk quota in the machines used for testing, larger systems of equations could not be tested because the input file would be very large to be saved on the disk.

10. Conclusion

Direct solvers for sparse systems of equations are very useful in scientific research and engineering. The results of the research showed that the algorithms for solving sparse systems of equations on distributed-memory machines are sometimes inefficient because of the high communication cost involved. To improve the performance of the algorithms and the programs, more efficient ordering and elimination algorithm suitable for distributed-memory machines needs to be developed. Other kinds of machines, such as multi-processors, can also be considered in the implementation. Of course the characteristics of this kind of machines should be examined when making new algorithms.

References

[1] Kumar, Vipin, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms,* Benjamin/Cummings: Redwood City, CA, 1994.

[2] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, *PVM 3 User's Guide and Reference Manual,* Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1993

[3] nCUBE Corp., *nCUBE 2 Programmer's Guide,* 1994

[4] Texas A&M University Super-computer Center, *Parallel Computing Orientation Guide,* College Station, Texas, 1993

[5] Xiong, Z. X. and X. Y. Li, The Application of Bivariate Cubic Spline, *Approximation, Optimization and Computing: Theory and Applications,* Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990

[6] Matarese, Joe, *nCUBE Manual Page Form*, Earth Resources Lab, MIT, Cambridge, MA, 1994