Parallel Matrix Inversion Techniques

K.K.Lau, M.J. Kumar, S. Venkatesh
Department of Computer Science, Curtin University of Technology
Bentley, Perth, Australia
email: (laukk,kumar,svetha)@cs.curtin.edu.au

Abstract

In this paper, we present techniques for inverting sparse, symmetric and positive definite matrices on parallel and distributed computers. We propose two algorithms, one for SIMD implementation and the other for MIMD implementation. These algorithms are modified versions of Gaussian elimination and they take into account the sparseness of the matrix. Our algorithms perform better than the general parallel Gaussian elimination algorithm. In order to demonstrate the usefulness of our technique, we implemented the *snake* problem using our sparse matrix algorithm. Our studies reveal that the proposed sparse matrix inversion algorithm significantly reduces the time taken for obtaining the solution of the *snake* problem. In this paper, we present the results of our experimental work.

Keywords : Sparse matrices, matrix inversion, SIMD, MIMD, PVM, computer vision and snakes.

1 Introduction

Solving systems of linear equations is a problem of importance in many applications. Typically, a set of linear equations is represented by

$$Ax = b$$

where A is a $n \times n$ matrix, x and b are $n \times 1$ vectors. This problem can be solved by matrix inversion. The inverse of a matrix A, denoted by A^{-1}, has the property that a system of equations $Ax = b$ can be solved by performing the matrix vector multiplication $x = A^{-1}b$. However, $O(n^3)$ operations are required to compute x on a serial computer where b is given [Sed88]. Even with some other factorization techniques like LU-decomposition, Choleski factorization and normalized factorization, we need $O(n^2)$ operations to compute x given b [DD91]. Besides this, there are several iterative methods for inverting matrices such as Jacobi iterative method and the conjugate gradient method [BT89]. Iterative methods do not obtain an exact solution of $Ax = b$ in finite time, but converge to a solution asymptotically. Unfortunately, no single iterative method is robust enough to solve all sparse linear systems accurately and efficiently [DD91].

However, to solve sparse linear equations, a normal direct matrix inversion or factorization is inefficient in terms of computer time and memory. Normal matrix operations do not take into account the sparseness of the matrices. A great deal of computing power is spent on multiplication of zeros. Moreover, significant memory is wasted in storing zeros. There is a need to optimize the computation time and memory by exploiting the sparseness of the matrices.

Even though, matrix inversion demands high computational requirements, its inherent parallelism has led to much interest in parallel implementation. The idea of exploiting parallelism in matrix inversion can be traced back to [Pea67]. Since then, several parallel matrix inversion algorithms have been developed. Gaussian elimination can be easily implemented in parallel and its time complexity is $O(n^2)$ with n processors [BT89]. Csanky [Csa76] discusses a method for inversion of a square matrix in $O(log^2 n)$ time, where he has shown that inverting matrices of a certain class can be reduced to the problem of matrix multiplication. However, Csanky's algorithm is prone to numerical problems and it uses an excessive number $O(n^4)$ of processors, Bojanczyk [Boj84] also transforms the inversion problem into matrix multiplication and achieved the same complexity, $O(log^2 n)$ but using only $O(n^3)$ processors. However, if fewer, say n processors are available, a very accurate approximation of the inverse algorithm of [Boj84] is obtained in $O(n^2 \log n)$ time steps. This is somewhat slower than Gaussian elimination. Furthermore, all the parallel algorithms described in the literature above do not take into account the sparseness of a matrix.

To the best of our knowledge, not much research has been reported on parallel sparse matrix inversion. In this paper, we propose two algorithms, one for SIMD computers and the other for MIMD computers, for inverting a sparse matrix A which is symmetric, positive definite. An example of such a matrix is shown in Figure 1. Each

$$A = \begin{bmatrix} a & b & c & & & & c & b \\ b & a & b & c & & & & c \\ c & b & a & b & c & & & \\ & c & b & a & b & c & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & c & b & a & b & c \\ c & & & & c & b & a & b \\ b & c & & & & c & b & a \end{bmatrix}$$

Figure 1: an example of a sparse matrix

row (or column) has the same number of non-zero (k) elements and we assume that $k << n$. Such systems occur in many applications [BE77]. Our implementations are modified versions of the basic Gaussian elimination algorithm. We take advantage of the fact that the matrix we used is sparse, symmetric and positive definite. For matrix size of $n \times n$, our SIMD algorithm works best when the PE array size is greater than $((k/2)+k+n)\times n)$. For larger matrices, we map several matrix elements onto each PE. In section 4, we shall prove that our parallel SIMD sparse matrix inversion technique gives good results even when the PE array size does not match with the matrix size. We shall also prove that our parallel MIMD sparse inversion technique has higher computational speed and lower communication cost compared to the general MIMD Gaussian elimination algorithm. We have implemented our MIMD algorithm on a network of workstations (NOWs) using PVM.

The remainder of this paper is organized as follows : we will examine the characteristics of sparse, symmetric and positive definite matrix inversion in section 2. And based on those characteristics, we will present our algorithms and implementation details for SIMD and MIMD computers in section 3. In section 4, we will discuss the performance of the implementation technique.

2 Characteristics of sparse symmetric matrix inversion

A matrix is symmetric if $A = A^T$, where A^T is the transpose of A. In our experiments, we make use of two important characteristics of sparse symmetric matrix inversion. Firstly, if A is symmetric and positive definite, then A^{-1} is also symmetric and positive definite. This identity matches the concept of Cholesky factorization. Therefore, the factorization steps can be continued without pivoting until the inversion is obtained.

Secondly, if A is sparse, its inverse will not be sparse in general. This identity supports work done by [DD91]. Consequently, sparsity of a matrix does not reduce the storage requirements of its inverse. Based on the characteristics above, we identified that Gaussian elimination as the most suitable method for our implementation of sparse symmetric matrix inversion. Since matrix A is positive definite, it is unnecessary to do any parallel pivoting when we are eliminating the matrix. This can save a lot of computation cost. Parallel pivoting strategies are discussed in [Ala95] and [Ala89]. Furthermore, from our experiments, we observed the relations shown in Figure 4. 1) each row in matrix A^{-1} is a rearrangement of the vector $[d, e, f, ...]$, 2) The ith row vector can be obtained by shifting the (i+1)th row vector one to the left with wrap around. Therefore, by using Gaussian elimination, if we reduce matrix A to the upper triangular matrix by row operations and simultaneously apply these operations to identity matrix I, we can have the last row vector of matrix A^{-1}. Therefore, we can use the last row to generate the whole matrix A^{-1} in parallel instead of further elimination or forward/backward substitution.

$$A = \begin{bmatrix} a & b & c & & & & c & b \\ b & a & b & c & & & & c \\ c & b & a & b & c & & & \\ & c & b & a & b & c & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & c & b & a & b & c \\ c & & & & c & b & a & b \\ b & c & & & & c & b & a \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} d & e & f & g & h & \ldots & g & f & e \\ e & d & e & f & g & \ldots & h & g & f \\ f & e & d & e & f & \ldots & g & h & g \\ g & f & e & d & e & \ldots & f & g & h \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & & \\ g & h & g & f & e & \ldots & d & e & f \\ f & g & h & g & f & \ldots & e & d & e \\ e & f & g & h & g & \ldots & f & e & d \end{bmatrix}$$

Figure 2: The relations of matrix A and A^{-1}

3 Modified Gaussian elimination algorithms

3.1 Storage scheme for sparse matrix

Our implementation is a modified version of the basic Gaussian elimination algorithm. Following the Gaussian elimination, we reduce A to the upper triangular matrix by row operation and simultaneously apply these operations to I to produce A^{-1}. Since each row has k non-zero elements, matrix A is stored using two $(k/2+k)\times n$ arrays called VAL and X. Each row of VAL contains the non-zero elements of the corresponding row of the sparse matrix and the array X stores the row numbers of the corresponding entries in VAL. In Figure 3, we show a sparse 64×64 matrix with $k = 5$ stored in our format. The purpose of the condensation is to convert the representation of the sparse matrix to a format better suited for Gaussian elimination. In specific, the goals are :

- To assign mainly the non-zero elements of the matrix to processors. Thus, the converted matrix is denser and the utilization is enhanced.
- To preserve the connectivity of the sparse matrix. As we shall see in the later part of this section, in each column elimination, there is no conflict in the position of the X coordinates.

We observe, first that any row operation on A can be described as a sequence of :

1. Division Step : let $\lambda = a_(i,i)$ and we divide row i by λ
2. Elimination Step : let $\lambda = a_(i,j)$ and we add $-\lambda$ times row i to row j.

In the following sections, we will present the SIMD and MIMD implementation of the division step and elimination step for matrix A and matrix A^{-1}.

$$VAL = \begin{bmatrix} a & b & c & - & - & c & b \\ b & a & b & c & - & - & c \\ c & b & a & b & c & - & - \\ c & b & a & b & c & - & - \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ c & b & a & b & c & - & - \\ c & - & - & c & b & a & b \\ b & c & - & - & c & b & a \end{bmatrix}$$

$$X = \begin{bmatrix} 0 & 1 & 2 & - & - & 62 & 63 \\ 0 & 1 & 2 & 3 & - & - & 63 \\ 0 & 1 & 2 & 3 & 4 & - & - \\ 1 & 2 & 3 & 4 & 5 & - & - \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ 59 & 60 & 61 & 62 & 63 & - & - \\ 0 & - & - & 60 & 61 & 62 & 63 \\ 0 & 1 & - & - & 61 & 62 & 63 \end{bmatrix}$$

Figure 3: The structure of VAL and X

3.2 SIMD algorithm

Given the PE array of size $m \times m$, we transform the $n \times n$ matrix A, into two $((k/2)+k) \times n$ matrices VAL and X and we map the matrices onto the first $(k/2)+k$ columns of the PE array; k is the number of non-zero elements on each matrix row. If $m \geq (k/2+k+n)$ we map the identity matrix I' on the following n columns. Therefore, any operations applied to VAL matrix can be simultaneously applied to I' in parallel. In other words, this algorithm works best when the PE array size is $((k/2+k)+n) \times n$. However, in the case where the matrix size is bigger than PE array size, we map the I' onto the whole PE array evenly and apply those row operations to VAL and I' in serial time.

In cases where the matrix size is bigger than the PE array, if we can ensure that we map row i and $i+1$ to different PE rows, then each PE will create at most one matrix non-zero element in each column elimination. This results in higher computational speed than general parallel SIMD Gaussian elimination algorithm.

In the rest of this section, we assume $m = n$. Figure 4 shows our modified version of Gaussian elimination. To do the division step, we broadcast the $val(i,0)$ to the rest of the row. Then since every PE in the row i has a copy of $val(i,0)$, the first $(k/2+k)$ PEs can do the division step for matrix A by performing $val(i,j)/val(i,0)$ in parallel. Similarly, every PE on the i^{th} row can do the division step for matrix I' by performing $I'(i,j)/val(i,0)$ in parallel.

The elimination step is carried out in a similar way as illustrated in the Figure 4. A typical computation of our SIMD Gaussian elimination procedure in the i^{th} iteration of the outer loop is shown in Figure 5. Since each column of matrix A has only k non-zero elements, only $(k \times ((k/2)+k))$ of VAL and $k \times n$ of matrix I' are computationally active.

Once the elimination step is completed, we shift the entries at VAL and X for next column elimination. This

```
procedure_SIMD(VAL, Inv)
begin
  for i = 0 to n-1 do
  begin
    for PE(i,0)
      Broadcast VAL(i,0) along its row
    for PE(i,j), j = 0 to (k/2+k)-1 do in parallel
      VAL(i,j) /= VAL(i,0)
    for PE(i,j), j = 0 to n-1 do in parallel
      I'(i,j) /= VAL(i,0)

    for PE(i,j)
      Broadcast VAL(i,j) or I'(i,j) along its column
    for PE(l,i), i < l < i+(k/2) or l > n - (k/2)
      Broadcast VAL(l,0) along its row

    for PE(l,j), j = 0 to (k/2+k)-1 do in parallel
      VAL(l,j) -= VAL(i,j) * VAL(l,0)
    for PE(l,j), j = 0 to n-1 do in parallel
      I'(l,j) -= I'(i,j) * VAL(l,0)
  end
end
```

Figure 4: Our SIMD Gaussian elimination algorithm

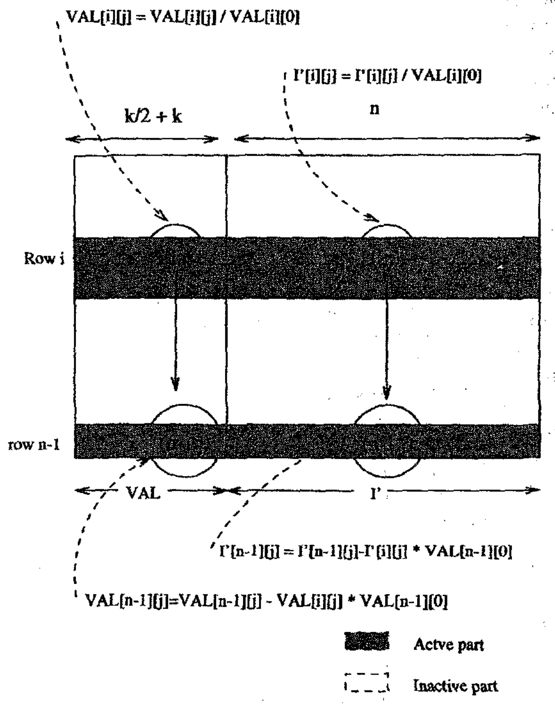
VAL[i][j] = VAL[i][j] / VAL[i][0]
I'[i][j] = I'[i][j] / VAL[i][0]
k/2 + k
n
Row i
row n-1
VAL
I'
I'[n-1][j] = I'[n-1][j]-I'[i][j] * VAL[n-1][0]
VAL[n-1][j]=VAL[n-1][j] - VAL[i][j] * VAL[n-1][0]
Actve part
Inactive part

Figure 5: Computation in our modified Gaussian elimination

movement is shown in Figure 6.

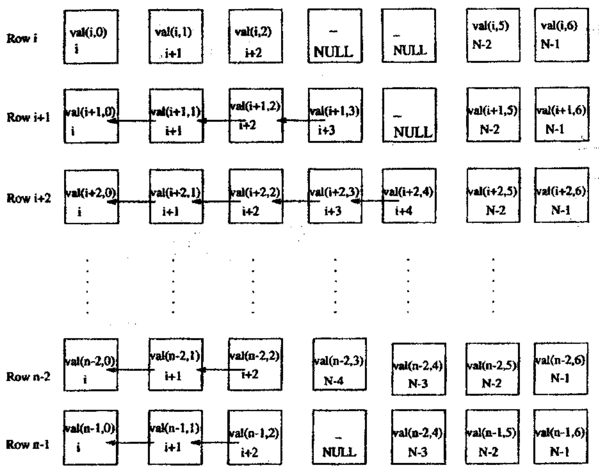

Figure 6: Shifting VAL and X

3.3 MIMD algorithm

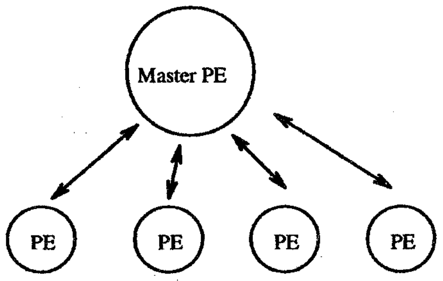

Figure 7: master/slave model

Our algorithm is implemented in a master/slave program. All initial matrices and the result matrix will be stored in the master workstation. The slaves are sent partitions by the master, which are then stored in local memory and operated on, with the results being sent back to the master to be collated. For implementation of the MIMD algorithm, we employ a network of workstations (NOWs) and the Parallel Virtual Machine (PVM) library. In the rest of this section we refer to each workstation as a PE. The configuration is shown in Figure 7.

Similar to the SIMD algorithm, we transform the $n \times n$ matrix A into two $((k/2)+k) \times n$ matrices VAL and X. The whole of VAL and X will be stored in the master PE. The master PE also maps the matrix I' onto the the slave PEs evenly. For instance, if the matrix size is 256, the number of slaves is 4 and k is equal to 5, the master PE will have the transformed matrices VAL and X (each 7×256) and each slave will have a submatrix of I' of size 64×256.

Our MIMD Gaussian elimination algorithm is shown in Figure 8 and Figure 9. In each column elimination, the master PE needs to broadcast all the non-zero elements of column i in matrix VAL to all the slaves (see Figure 8). Since each column has k non-zero elements at the most, this results in lower communication cost than general parallel MIMD Gaussian elimination algorithm.

```
procedure MIMD_master(VAL, Inv)
begin
  For i = 0 to n-1 do
  begin
    For MASTER do
      Broadcast VAL(i,0) to the slaves
        For j = 0 to (k/2+k)-1 do
            VAL(i,j) /= VAL(i,0)

       For l = i < l < i+(k/2)
         Broadcast VAL(l,0) to the slaves
         For j = 0 to (k/2+k)-1 do
             VAL(l,j) -= VAL(i,j) * VAL(l,0)

       For l = l > n - (k/2)
         Broadcast VAL(l,0) to the slaves
         For j = 0 to (k/2+k)-1 do
             VAL(l,j) -= VAL(i,j) * VAL(l,0)
    end
end
```

Figure 8: Our MIMD Gaussian elimination algorithm (for master)

```
procedure MIMD_slave(VAL,Inv)
begin
  For i = 0 to n-1 do
  begin
    For SLAVE do
        receive VAL(i,0)
        For j = 0 to n-1 do
          I'(i,j) /= VAL(i,0)

        For l = i < l < i+(k/2)
          receive VAL(l,0)
          For j = 0 to n-1 do
              I'(i,j) -= I'(i,j) * VAL(l,0)

        For l = l > n - (k/2) do
          receive VAL(l,0)
          For j = 0 to n-1 do
              I'(i,j) -= I'(i,j) * VAL(l,0)
    end
end
```

Figure 9: Our MIMD Gaussian elimination algorithm (for slave)

Once the communication operation is completed, the master and slaves can perform their computation independently. The master can compute the division step and elimination step for matrix VAL and X (Figure 8) and concurrently, each slave can compute the division step and elimination step for matrix I' (Figure 9). Since each column has only k non-zero elements, this results in lower computational cost than general parallel Gaussian elimination algorithm.

4 Performance analysis and result

In this section, we determine the number of computational steps for our implementation of sparse matrix inversion.

Time complexity for the SIMD algorithm

Assuming the matrix size is $n \times n$ and PE size is $m \times m$ and if $m > ((k+2)+k+n)$, the total parallel time to compute A^{-1} is $O(n)$. Each column elimination takes constant time. However, as matrix VAL and matrix I' are mapped in different set of PEs, any operations applied on matrix VAL will be simultaneously applied to I' in parallel. Moreover, if $n >> m$, we perform operations on matrix VAL and matrix I' in separate steps. Suppose $n = pm$ $(p \geq 1)$, we map $(p \times p)$ subarrays of matrix I' to the PE array. Even though the time complexity is the same as that of the general Gaussian elimination, there is a significant reduction in the execution time and the space in our algorithm. This is manifested in our experimental results.

Time complexity for the MIMD algorithm

If we use $m << n$ workstations, map $\frac{n}{m} = p$ rows of the given matrix to each of the slave PEs for computation. Each PE performs computation for $\frac{n}{m}$ rows in a serial fashion. Hence the complexity of the algorithm is $O(\frac{n}{m})$.

Further, as we are using the condensed form of the given matrix, we are using less memory space. If there are k valid elements in the row of A, we store A in an array of size $(\frac{k}{2}+k) \times n$ rather than $n \times n$. Hence, saving in space is $\frac{n \times n}{(\frac{k}{2}+k) \times n} \times 100\% = \frac{n}{\frac{k}{2}+k} \times 100\%$.

Result

The implementation of our algorithms were tested for various matrix size and various non-zero element (k). The results of the SIMD implementation on the DEC MP-1 Maspar system are shown in Table 1. Our Maspar MP-1 is a general purpose SIMD machine with 2048 (32×64) processing elements. Our result is compared to the general SIMD Gaussian elimination algorithm.

size	k	SIMD_sparse	SIMD_general
64×64	5	0.1833	0.1833
128×128	5	0.5833	1.0166
256×256	5	1.549	5.6164
512×512	5	4.416	24.4657

Table 1: Time performance of SIMD matrix inversion (in seconds)

The results of the MIMD implementation on the network of workstations (NOWs) are shown in Table 2. The number of slaves is eight. Comparing to the general MIMD Gaussian elimination algorithm (see Table 3), it is shown that our MIMD algorithm is faster.

k	64×64	128×128	256×256	512×512
5	0.21	0.39	0.75	1.48
7	0.21	0.4	0.78	1.49
9	0.21	0.41	0.86	1.61

Table 2: Time performance of MIMD sparse matrix inversion algorithm (in seconds)

k	64×64	128×128	256×256	512×512
5	0.144	0.304	1.121	4.409
7	0.121	0.312	1.238	4.796
9	0.113	0.316	1.444	4.843

Table 3: Time performance of MIMD general matrix inversion algorithm (in seconds)

Furthermore, the sequential sparse matrix Gaussian elimination algorithm and the sequential general Gaussian elimination algorithm have also been implemented. The algorithms were executed on a DEC/500 workstation. The timing for various matrix size and non-zero elements are shown in Table 4 and Table 5.

k	64×64	128×128	256×256	512×512
5	0.02	0.13	1.17	7.11
7	0.03	0.16	1.65	7.38
9	0.03	0.24	1.94	7.65

Table 4: Time performance of sequential sparse matrix inversion algorithm (in seconds)

5 Application - *snake*

Our technique has proven to be useful when the matrix is sparse, symmetric and positive definite. We have used this technique in an application called *snake*. The problem *snake* is from the domain of motion tracking in computer vision. It detects the edges of an object by finite element method. A *snake* is an iterative energy minimization procedure using sparse matrix method. It involves convolution, sparse matrix inversion and matrix-vector multiplication. The coordinates of the snake points at $time_t$ are given by

$$x_t = (A + \gamma I)^{-1}(x_{t-1} - f_x(x_{t-1}, y_{t-1}))$$
$$y_t = (A + \gamma I)^{-1}(y_{t-1} - f_y(x_{t-1}, y_{t-1}))$$

Assuming n is the number of points in a *snake*, x and y are $n \times 1$ vectors which store the x coordinates and y coordinates of the points. A is a $n \times n$ sparse, symmetric and

k	64 × 64	128 × 128	256 × 256	512 × 512
5	0.28	1.85	15.59	121.61
7	0.28	1.22	13.51	126.42
9	0.28	1.1	12.34	114.94

Table 5: Time performance of sequential general matrix inversion algorithm (in seconds)

positive definite matrix. From the equations above, we can observe that matrix A occurs in a problem where the solution $Ax = b$ is incorporated into an iterative procedure. Readers not familiar with the working of the *snake* are advised to refer to [KW87].

We use our matrix inversion algorithm to compute A^{-1}. Since the matrix A^{-1} is a dense matrix, we use the general parallel matrix-vector multiplication algorithm to perform all the iteration. We have constructed a *snake* of a square representing its boundary using 64 points. Figure 10 shows the original image and the initial position of *snake*. We set $\alpha = 0.3$ and $\beta = 0.04$ (α and β are the parameters of *snake*) Figures 11 - 11 show the locations of *snake* in every 30 iterations.

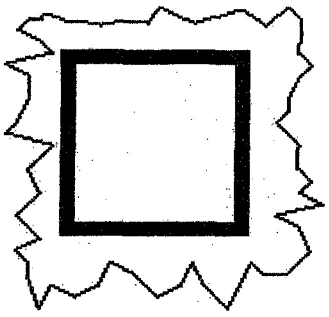

Figure 10: The Original image and initial position

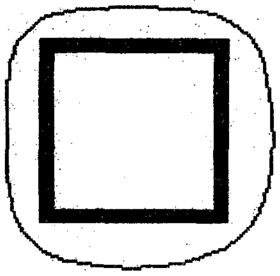

Figure 11: 30 iterations

In the figures, we can see that the *snake* was attracted to the square boundary from a fairly large distance. Table 6 shows the speedup of various implementations of the *snake*. The SIMD *snakes* algorithm was executed on the DEC MP-1 Maspar system, the MIMD algorithm was

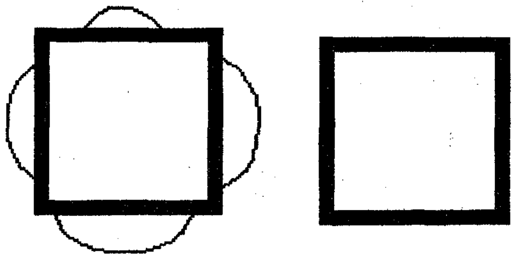

Figure 12: a) 60 iterations and b) 90 iterations

executed on the network of workstations (1 master and 8 slaves) and the sequential algorithm was executed on the DEC/500 workstation. The timing is measured from convolution, matrix inversion to 90 energy minimization iterations.

No of points	Sequential	SIMD	MIMD
64	34.2	1.9999	2.2
12	43.89	3.6665	4.2
256	116.763	7.5496	10.8
512	404.78	16.2326	35.3

Table 6: Timing difference between the SIMD *snake*, MIMD *snake* and sequential *snake* for the example (in seconds)

6 Conclusion

In this paper, we have presented efficient algorithms for inverting a sparse, symmetric and positive definite matrix problems. These algorithms are modified version of Gaussian elimination and take into account the sparseness of the matrix. The results obtained by us are very encouraging as they indicate a substantial improvement in execution time over the general parallel Gaussian elimination algorithm. We have presented results for SIMD as well as MIMD computations.

References

[Ala89] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing*, 11:201–221, 1989.

[Ala95] G. Alaghband. Parallel sparse matrix solution and performance. *Parallel Computing*, 21:1407–1430, 1995.

[BE77] A. Benson and D. J. Evans. A normalized algorithm for the solution of positive definite symmetric quindiagonal systems of linear equations. *ACM Transactions on Mathematical Software*, 3:96–103, 1977.

[Boj84] A. Bojanczyk. Complexity of solving linear systems in different models of computation. *SiAM Journal of Numerical Analysis*, 21:792–603, 1984.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation. Numerical Methods.* Prentice Hall, 1989.

[Csa76] L. Csanky. Fast parallel matrix inversion algorithm. *SIAM Journal on Computing*, 5(4):618–623, 1976.

[DD91] J. J. Dongarra and I. S. Duff. *Solving linear systems on vector and shared memory computers.* Philadelphia : Society for Industrial and Applied Mathematics, 1991.

[KW87] M. Kass and A. Witkin. Snakes: Active contour models. In *Proceedings of 1st International Conference on Computer Vision*, pages 259–268, London, 1987.

[Pea67] M. C. Pease. Matrix inversion using parallel processing. *Journal of the ACM*, 14:757–764, October 1967.

[Sed88] R. Sedgewick. *Algorithms.* Addison Wesley, 1988.