Direct Methods for Linear Systems on Distributed Memory Parallel Computers

S. Nishimura[1], T. Shigehara[1], H. Mizoguchi[1], T. Mishima[1] and H. Kobayashi[2]

[1]Department of Information and Computer Sciences, Saitama University
Shimo-Okubo 255, Urawa, Saitama 338-8570, JAPAN
Phone: +81-48-858-9035, Fax: +81-48-858-3716
E-mail: seiji@me.ics.saitama-u.ac.jp, sigehara@ics.saitama-u.ac.jp
[2]Department of Mathematics, College of Science and Technology, Nihon University
Surugadai 1-8-14, Kanda, Chiyoda, Tokyo 101-8308, JAPAN

Abstract: We discuss the direct methods (Gauss-Jordan and Gaussian eliminations) to solve linear systems on distributed memory parallel computers. It will be shown that the so-called row-cyclic storage gives rise to the best performance among the standard three (row-cyclic, column-cyclic and cyclic-cyclic) data storages. We also show that Gauss-Jordan elimination, rather than Gaussian elimination, is highly efficient for the direct solution of linear systems in parallel processing, though Gauss-Jordan elimination requires a larger number of arithmetic operations than Gaussian elimination. Numerical experiment is performed on HITACHI SR2201 with the standard libraries MPI and BLAS.

1. Introduction

Gauss-Jordan and Gaussian eliminations are the standard direct methods to solve dense linear systems. In Gauss-Jordan elimination, the reduction of the coefficient matrix A to the identity matrix consists of n major steps which correspond to n matrix row operations. These are similar to the n steps in Gaussian elimination (LU decomposition) which reduce A to an upper triangular matrix U. The difference between the two eliminations lies in the fact that while the elements only below the diagonal are eliminated in LU decomposition, Jordan's variation eliminates the elements not only below, but also above the diagonal. As a result, the number of the arithmetic operations (additions and multiplications) is of $O(2n^3/3)$ for Gaussian elimination, whereas Gauss-Jordan elimination requires the operations of $O(n^3)$. (We use the notation $O(cn^k)$ also for the highest order term in the expression.) This is indeed the reason why Gaussian elimination rather than Gauss-Jordan elimination is often used in sequential processing. However, the aforementioned algorithms indicate that the parallelism in Gauss-Jordan elimination is much higher than that for Gaussian elimination, especially in the last steps for reduction [1]. The first purpose of this paper is to show that parallel Gauss-Jordan elimination is even faster than parallel Gaussian elimination over a wide range of problem size n. It is obvious that the parallelism is highly desired especially in large-scale problems which can never be solved in a realistic duration with the standard sequential machines. Keeping in mind the recent development of numerical simulations in various fields including structure analysis, fluid dynamics and quantum chemistry, it is crucial to complete the reduction of matrix larger than $n \sim 10000$ in a realistic duration. We show in this paper that Gauss-Jordan elimination is indeed highly efficient even for such a huge size of linear systems.

The second purpose is to examine the dependence on data storage in the parallel eliminations. On distributed memory parallel computers, it is important to choose suitable data storage which ensures the good performance of arithmetic operations on each processor as well as the balance between communication and computation. We examine the standard three types; row-cyclic [2], column-cyclic [3], cyclic-cyclic [4] data storages. We will show that the row-cyclic storage is most suitable both for the eliminations.

The paper is organized as follows. Experimental environment is explained in Sect.2. After examining the standard sequential algorithm of Gauss-Jordan and Gaussian eliminations in Sect.3, we discuss the parallel algorithms in Sect.4. The current work is summarized in Sect.5.

2. Experimental Environment

We summarize experimental environment in this section. Numerical experiments are performed on HITACHI SR2201, at the Computer Centre (current Information Technology Center), the University of Tokyo. The SR2201 is one of up-to-date parallel supercomputers with distributed memory architecture. The theoretical peak performance of a single processing element is 300 MFLOPS. The source programs are written by C programming language. The compile options are `+O4 -Wc,'-h'`. We use the DAXPY subroutine of level 1 BLAS library [5] for the most inner core loop in both Gauss-Jordan and Gaussian eliminations. The operation of DAXPY is

$$\mathbf{y} := \mathbf{y} + m\mathbf{z}, \quad (1)$$

where $\mathbf{y}, \mathbf{z}$ are vectors with the same dimension and m is a scalar. In parallel processing, we use MPI (Message Passing Interface) [6] as communication library. The MPI is a *de facto* standard library which describes the data communication among multi processors. As the coefficient matrix of linear systems, we take the Frank

matrix $A = (a_{i,j})$ $a_{i,j} = n + 1 - \min(i,j)$; $i,j = 1,2,\cdots,n$ throughout the paper.

3. Standard Algorithm

The standard sequential algorithm of Gaussian and Gauss-Jordan eliminations is shown in Fig.1. Here the partial pivoting is taken into account. In the actual implementation, we use the DAXPY routine for the most inner loop in both. Table 1 shows the execution time for the problem size n = 1024 ,2048 and 4096. A remarkable feature is that Gauss-Jordan elimination shows fairly good performance even on a single processor. The ratio of the execution time between Gauss-Jordan and Gaussian eliminations is close to 1.0, contrary to the value 1.5 expected from the operation count. This is because the performance of the DAXPY routine depends on the length of the loop which calls DAXPY inside. Fig.2 shows the average execution time for a single call of DAXPY. Horizontal axis is the length of the loop which calls DAXPY inside. The broken and solid lines are the results for vector length 1024 and 2048, respectively. As the loop length increases, the performance of DAXPY becomes higher. In Gauss-Jordan elimination, all rows except the pivot row are eliminated. Hence, DAXPY is always called $n - 1$ times at each step of the elimination. On the other hand, Gaussian version eliminates the rows only below the pivot row. As a result, the number of calls of DAXPY becomes smaller and smaller, as the elimination proceeds. This is the reason why Gauss-Jordan elimination is comparable with Gaussian elimination even in a single processing.

Table 1: Execution time [sec] of Gauss-Jordan and Gaussian eliminations on a single processor.

n	Gauss-Jordan	Gaussian
1024	14.36	13.53
2048	113.50	108.29
4096	905.55	877.26

4. Parallel Algorithm

We proceed to the parallel algorithms of Gauss-Jordan and Gaussian eliminations. In the following, we denote the number of processors by p and label p processors by $0, 1, 2, \cdots, p-1$, respectively. In parallel processing, the performance depends on the data storage, in general. We examine the standard three types; row-cyclic, column-cyclic, cyclic-cyclic data storages. Fig.3 schematically shows each data storage in case of $p = 4$ and $n = 8$. The data partition for a general case is determined in a similar manner. (We assume that n/p is an integer for row-cyclic and column-cyclic storages, while $\sqrt{p}$ and $n/\sqrt{p}$ are integers for cyclic-cyclic storage.)

For row-cyclic storage, the parallel algorithm at each step of the elimination is as follows;

Row-cyclic storage:

Gauss-Jordan elimination

```
/* initialize of index vector :  pv */
for k=1 to n
  /* partial pivoting */
  pv_k   := pv(k);
  inv_pv := 1/a(pv_k,k);
  for i=1 to n
    if(i ≠ pv_k)
      m := a(i,k)*inv_pv;
      for j=k+1 to n
        a(i,j) := a(i,j)-m*a(pv_k,j);
      end
      b(i) := b(i)-m*b(pv_k);
    end
  end
end
```

Gaussian elimination (LU factorization)

```
/* initialize of index vector :  pv */
for k=1 to n
  /* partial pivoting */
  pv_k   := pv(k);
  inv_pv := a(pv_k,k) := 1/a(pv_k,k);
  for i=k+1 to n
    pv_i := pv(i);
    m    := a(pv_i,k) := a(pv_i,k)*inv_pv;
    for j=k+1 to n
      a(pv_i,j) := a(pv_i,j)-m*a(pv_k,j);
    end
    b(pv_i) := b(pv_i)-m*b(pv_k);
  end
end
```

Figure 1: Standard sequential algorithm for Gauss-Jordan and Gaussian eliminations with partial pivoting.

1. Each processor performs local partial pivoting and sends the local pivot element to processor 0 by using MPI_Reduce function with MPI_MAXLOC (maximum and location of maximum) operand.
2. Processor 0 determines the global pivot row and renews the index vector.
3. Processor 0 sends the index vector to all other processors by using MPI_Bcast function.
4. Processor which has the global pivot row sends it to all other processors by using MPI_Bcast function.
5. Each processor performs the elimination procedure by using DAXPY.

Note that one call of MPI_Reduce and two calls of MPI_Bcast are required at each step of the elimination for row-cyclic storage.

For column-cyclic storage, the parallel algorithm at k-th ($k = 1, 2, \cdots, n$) step of the elimination is as follows. Here we set $\tilde{k} = k - 1 \pmod{p}$. Note that the k-th column is stored in processor $\tilde{k}$.

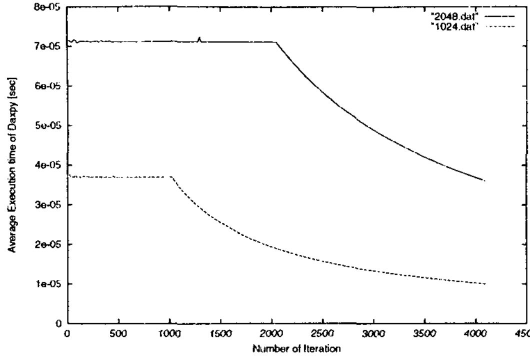

Figure 2: Average execution time [sec] for a single call of DAXPY. Horizontal axis is the length of the loop which calls DAXPY inside. Broken and solid lines show the result for the vector length 1024 and 2048, respectively.

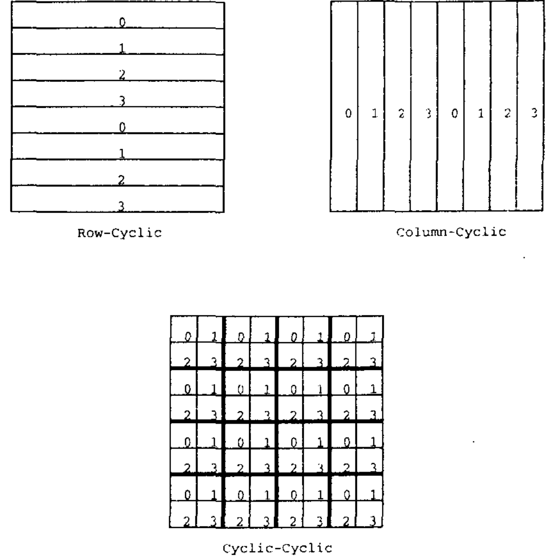

Figure 3: Schema of the three storages for $p = 4, n = 8$. The number on each vector (matrix element) represents the label of processor which stores it.

Column-cyclic storage:

1. Processor $\tilde{k}$ performs (global) partial pivoting and renews the index vector.
2. Processor $\tilde{k}$ sends the index vector to all other processors by using MPI_Bcast function.
3. Processor $\tilde{k}$ calculates the ratios (multipliers m in Fig.1) between the pivot element and other elements in k-th column (except the pivot element in Gauss-Jordan elimination, while below the pivot element in Gaussian elimination).
4. Processor $\tilde{k}$ sends the multipliers to all other processors by using MPI_Bcast function.
5. Each processor performs the elimination procedure by using DAXPY.

Two calls of MPI_Bcast are required at each step of the elimination for column-cyclic storage.

In cyclic-cyclic storage, the grid structure of processors naturally induces the matrix notation in labeling processors; $p(i;j)$, $(i, j = 0, 1, 2, \cdots, \sqrt{p} - 1)$. For instance, $p(0;0) = 0$, $p(0;1) = 1$, $p(1;0) = 2$ and $p(1;1) = 3$ in Fig.3. In addition to the standard global communicator MPI_COMM_WORLD, each set of the processors $\{p(i;j) | j = 0, 1, 2, \cdots, \sqrt{p} - 1\}$, $(i = 0, 1, 2, \cdots, \sqrt{p} - 1)$ constitutes a communication universe in the row direction. As well, each set of $\{p(i;j) | i = 0, 1, 2, \cdots, \sqrt{p} - 1\}$, $(j = 0, 1, 2, \cdots, \sqrt{p} - 1)$ constitutes a communication universe in the column direction. The parallel algorithm at k-th $(k = 1, 2, \cdots, n)$ step of the elimination is shown below. Here we set $\tilde{k} = k - 1 \pmod{\sqrt{p}}$. Note that the elements of k-th column are stored in processors $p(i;\tilde{k})$, $(i = 0, 1, 2, \cdots, \sqrt{p} - 1)$.

Cyclic-cyclic storage:

1. Processors $p(i;\tilde{k})$, $(i = 0, 1, 2, \cdots, \sqrt{p} - 1)$ perform local partial pivoting and send the local pivot element to processor $p(0;\tilde{k})$ by using MPI_Reduce function with MPI_MAXLOC operand.
2. Processor $p(0;\tilde{k})$ determines the global pivot row and renews the index vector.
3. Processor $p(0;\tilde{k})$ sends the index vector to all other processors by using MPI_Bcast function.
4. Processors which have the elements of the global pivot row, say $p(\tilde{i};j)$, $(j = 0, 1, 2, \cdots, \sqrt{p} - 1)$ with some fixed $\tilde{i}$, send them to all other processors $p(i;j)$, $(i = 0, 1, 2, \cdots, \tilde{i} - 1, \tilde{i} + 1, \cdots, \sqrt{p} - 1)$ in the same column by using MPI_Bcast function.
5. Processors $p(i;\tilde{k})$, $(i = 0, 1, 2, \cdots, \sqrt{p} - 1)$ calculate the multipliers and send them to all other processors $p(i;j)$, $(j = 0, 1, 2, \cdots, \tilde{k} - 1, \tilde{k} + 1, \cdots, \sqrt{p} - 1)$ in the same row by using MPI_Bcast function.
6. Each processor performs the elimination procedure by using DAXPY.

One call of MPI_Reduce and three calls of MPI_Bcast are required at each step of the elimination for cyclic-cyclic storage.

Table 2 shows the execution time of parallel Gauss-Jordan and Gaussian eliminations for each data storage. The number of processors is fixed at $p = 64$, which is nearly the best for problem size $n \sim 2000$ for each storage. It can be seen that the row-cyclic storage gives rise to the best performance both for Gauss-Jordan and Gaussian eliminations. This is mainly because the core loop consists of a row operation (1) both for the eliminations. Also, in case of row-cyclic storage, partial pivoting as well as the calculation of multipliers can be performed completely in parallel. This is not the case for column-cyclic and cyclic-cyclic storages. In column-cyclic storage, in particular, partial pivoting as well as the calculation of multipliers concentrates on a single processor.

Table 2: Dependence on data storage. The number of processors is fixed at $p = 64$. Execution time is given in units of second.

Gauss-Jordan elimination			
n	row-cyclic	column-cyclic	cyclic-cyclic
1024	1.98	3.71	4.97
2048	6.42	18.09	18.98
Gaussian elimination			
n	row-cyclic	column-cyclic	cyclic-cyclic
1024	2.14	3.68	3.67
2048	7.23	13.26	13.90

Table 3: Execution time [sec] of parallel Gauss-Jordan and Gaussian eliminations with row-cyclic storage.

Gauss-Jordan elimination			
p	$n = 4096$	$n = 8192$	$n = 16384$
16	62.31	461.69	3583.63
32	38.47	245.88	1871.39
64	27.60	150.51	1000.89
128	23.61	107.80	588.70
256	23.33	91.38	410.28
Gaussian elimination			
p	$n = 4096$	$n = 8192$	$n = 16384$
16	65.34	471.14	3615.06
32	41.73	259.47	1925.48
64	31.00	164.83	1056.87
128	27.08	122.63	648.37
256	26.74	105.97	470.18

Let us proceed to examine the efficiency of parallel Gauss-Jordan and Gaussian eliminations. In the following, we assume the row-cyclic storage. Table 3 shows the execution time for both eliminations in case of problem size $n = 4096$, 8192 and 16384. For each case, the number of processors is changed as $p = 16, 32, 64, 128$ and 256. One can see that Gauss-Jordan elimination surpasses Gaussian elimination in the whole range of n and p. As the number of processors increases, the number of rows on each processor becomes small ($1/p$ compared to a single processing). As a result, the acceleration by DAXPY discussed in the previous section is weakened in parallel processing. However, it still remains especially in Gauss-Jordan elimination; The DAXPY routine is called n/p times on each processor at each step of Gauss-Jordan elimination, while in Gaussian elimination, the number of calls of DAXPY becomes smaller, as the elimination proceeds.

In case of row-cyclic storage, the execution time of Gaussian elimination with partial pivoting depends on the coefficient matrix A, since it eliminates the elements only below the pivot row. The Frank matrix causes no interchange of rows and as a consequence the load balance is completely kept among processors throughout the elimination. This means that the performance with the Frank matrix is the *best* for Gaussian elimination with partial pivoting. Note that the performance does not depend on the coefficient matrix in Gauss-Jordan elimination (even with partial pivoting). This indicates that the efficiency of Gauss-Jordan elimination becomes even higher for generic coefficient matrices.

It is important to estimate the execution time $T(n,p)$ for a given problem size n and the number of processors p. Based on a semi-theoretical argument, we have shown in [1] that the execution time of Gauss-Jordan elimination with row-cyclic storage is estimated as

$$T(n,p) = \alpha \frac{n^3}{p} + \beta n^{1+\gamma} \ln p, \qquad (2)$$

where the first and second terms on RHS describe the computation and communication times, respectively. The three parameters can be easily determined by simple benchmarks; $\alpha = 1.32 \times 10^{-8}$ [sec], $\beta = 0.58 \times 10^{-5}$ [sec], $\gamma = 0.58$ on SR2201 [1]. The estimate (2) is quite satisfactory all over the range of n and p in Table 3. We also stress that Eq.(2) makes it possible to determine the optimum number of processors for a given problem size *before* the actual execution.

5. Conclusion

We have discussed Gauss-Jordan and Gaussian eliminations for dense linear systems on distributed memory parallel computers. Contrary to a native expectation from the number of floating-point operations, Gauss-Jordan elimination is highly efficient and it indeed surpasses the Gaussian elimination in execution time. Concerning the data storage, the row-cyclic type is the most suitable because the core loop of both the eliminations consists of a row-vector operation. Numerical experiments on HITACHI SR2201 confirm the validity of our argument.

References

[1] S. Nishimura, T. Shigehara, H. Mizoguchi, and T. Mishima, "Efficiency of Gauss-Jordan Elimination for Dense Linear Systems on Distributed Memory Parallel Computers," Proceedings of ITC-CSCC'99, vol.2, pp.812-815, Sado, July, 1999.

[2] G. Golub and J.M. Ortega, "Scientific Computing — An Introduction with Parallel Computing," Academic Press, London, 1993.

[3] J.J. Dongarra and R.A. van de Geijn, "Reduction to condensed form for the eigenvalue problem on distributed memory architectures," Parallel Computing, vol.18, pp.973-982, 1992.

[4] H.Y. Chang, S. Utku, M. Salama, and D. Rapp, "A parallel Householder tridiagonalization stratagem using scattered square decomposition," Parallel Computing, vol.6, pp.297-311, 1988.

[5] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst, "Solving Linear Systems on Vector and Shared Memory Computers," SIAM, Philadelphia, 1991.

[6] Message Passing Interface Forum, "MPI: A message passing interface standard," Special issue on MPI, International Journal of Supercomputer Applications, vol.8, no.3/4, 1994.