A Linear Equation Solution Algorithm on a Coarse-grained Distributed-memory Parallel System

Qiang Li, David Barton, Naphtali Rishe
School of Computer Science
Florida International University –
The State University of Florida at Miami
University Park, Miami, FL 33199

Abstract

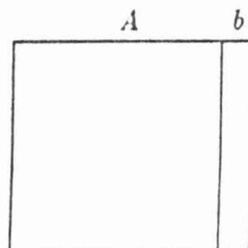
Solving large scale simultaneous linear equations is an often encountered problem in engineering and scientific computations. It is normally a time consuming process. This paper presents a practical algorithm which can efficiently utilize the coarse-grained distributed-memory parallel systems to speed up the computation. Experimental data and timing analysis are also presented. An interesting phenomenon that the times of speed-up was more than the number of processors employed is discussed. Although the algorithm is designed for a coarse-grained distributed-memory system, the principle can be used in a shared-memory system and it should tend to reduce the memory contention.

1 Gauss-Jordan elimination method

In this section, we briefly describe the classical Gauss-Jordan Elimination method for solving linear equations. We only discuss how the process evolves and how the data changes. The detailed Gauss-Jordan algorithm can be found in most numerical analysis textbook, e.g. [1]. Given a set of simultaneous linear equations of N variables

$$A\mathbf{x} = \mathbf{b}$$

they can be represented by a matrix M of $N \times (N+1)$ as show in the following:



The matrix is constructed by appending the b vector to the right-hand side of A matrix. Gauss-Jordan elimination can be viewed as working row by row on the matrix M starting from the first row. The following is the algorithm:

1. FOR $i = 1$ TO N DO
2. Divide the elements of the i-th row by $M[i, i]$
3. For each j in $1..N$ and $j \neq i$, subtract $M[j, i] \times M[i]$ from $M[j]$

During the Gauss-Jordan elimination process, the elements of M changes in a systematic fashion. After the first iteration, matrix M changes to:

$$
\begin{array}{c|c}
A & b \\
\hline
\begin{matrix} 1 & \cdots \\ 0 & \\ 0 & \\ \vdots & \\ 0 & \end{matrix} &
\end{array}
$$

The first element of the first row becomes 1 since it was divided by itself in step 2. The first elements of every other rows become zero because each of them was subtracted from itself in step 3. In the same way, after the second iteration, matrix M becomes the following:

$$
\begin{array}{c|c}
A & b \\
\hline
\begin{matrix} 1 & 0 & \cdots \\ 0 & 1 & \cdots \\ 0 & 0 & \\ \vdots & \vdots & \\ 0 & 0 & \end{matrix} &
\end{array}
$$

After the N-th iteration, A becomes a unit matrix and b contains the solution vector:

$$
\begin{array}{c|c}
A & b \\
\hline
\begin{matrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{matrix} &
\end{array}
$$

To minimize the effect of round-off errors, row pivoting are used in most cases. In addition, partial column pivoting is also used from time to time. Although there are many other more sophisticated methods for solving simultaneous linear equations, Gauss-Jordan elimination still plays a major roll in numerical processing.

2 The target parallel systems

The parallel systems which we are aimed at can be characterized as follows. Each of the processors is fairly powerful itself and it has its own basic operating system. Each processor is equipped with an exclusive memory module. In other words, each processor is more or less a stand-alone computer. The processors are interconnected into a network of certain topology.

The systems in question are "coarse-grained" systems. In general, a coarse-grained parallel system can be roughly defined as a system in which the data processing rate of the processor is much higher than the data transfer rate between the processors. Many commercial processors fall into this category.

The topologies of the interconnection networks vary from system to system. Among the parameters of a network, the connectivity is particularly important. Given the capacity of each communication channel of a system, higher connectivity usually implies a higher system communication power and, on the down side, higher cost. The connectivity of a processor in a system with P processors can vary from 1, the least connected, to P, i.e. completely connected.

Among different kinds of parallel systems, our study focuses on the most practical ones, i.e., the coarse-grained ones with distributed memories, simple topology and low connectivity. The reason is that they are widely available and relatively inexpensive. For example, INMOS Transputer[2] is one of them.

As the words "coarse-grained" imply, in our solution, that the size of the equations (number of simultaneous variables) is much greater than the number of processors in the system. Again, it is a typical practical situation.

To simplify our discussion, the sample system used in this paper is a one-directional ring network, each node of the network is a processor.

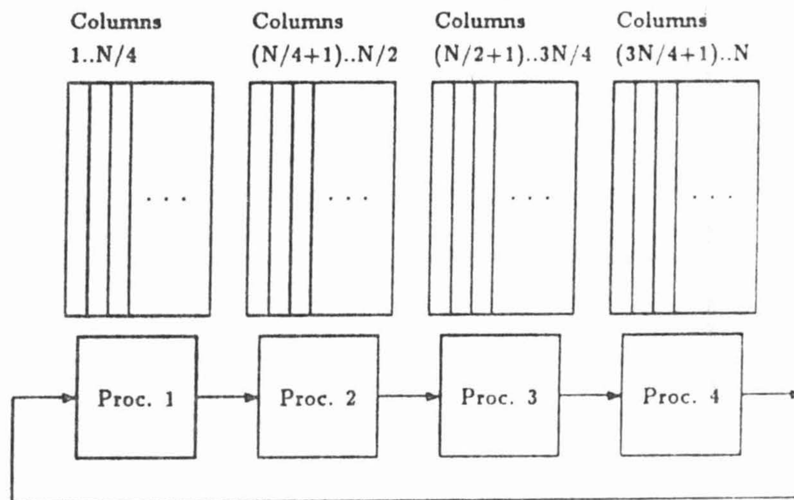


Figure 1: A distribution of data over the processors

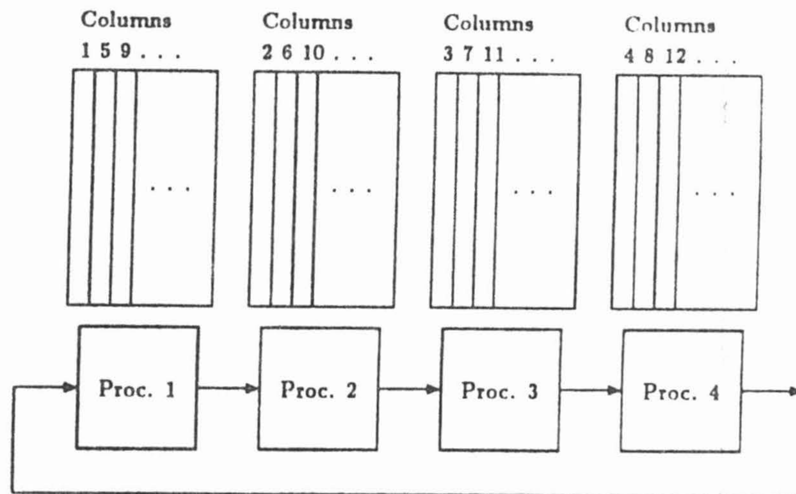


Figure 2: The improved data partitioning

3 Design of the algorithm

The parallelism residing in the linear equation solutions have been well studied[3]. Like other parallel computing problems, there are two critical points involved in utilizing the parallelism on a coarse-grained parallel system: minimizing the data flow between processors and balancing the processing loads of the processors. Sometimes the two points are in conflict with each other. The main issue is how to partition the data.

We begin with a study of an example. Assume that we have a one-directional ring network with P processors and the matrix M mentioned earlier has N columns and $N-1$ rows. Also assume that, without loss of generality, N is multiple of P.

One possible and straightforward partition of the data on a 4 processor system is shown in Figure 1. In general, each processor has N/P columns of the matrix M in its memory. The first processor has the first N/P columns, and the second processor has columns $N/P+1$ through $2 \times N/P$, and so on.

The system works as follows. For the first iteration, Processor 1 passes the first column (which is the only information needed for the other processors to process their columns) to Processor 2, and Processor 2 passes it to Processor 3, and so on. After passing data to next processor, each processor processes the columns that it has as in the normal Gauss-Jordan elimination. Then, Processor 1 initiates the second iteration by passing the second column to others. Processor 1 repeats the process until N/P iterations are done. At that point, Processor 2 will start to initiate iterations by passing its columns to others. The process goes on until the last processor finishes its last column, and the results are sent to the host.

In this way, the initial amount of data is balanced between the processors, but the processing loads are not. At the beginning of the Gauss Jordan elimination process, all of the P processors are working. However, each iteration of the elimination process produce a column in M, on the

```
• Configuring the system
• Loading data
• WHILE NOT done
      Receiving message;
      Distance := Distance - 1;
      If Distance > 0, pass the message to next node;
      CASE Message-type OF
         initiate : Pass a process message containing
                              the column indicated by CurrentColumn;
                    CurrentColumn := CurrentColumn + 1;
                    Process my columns;
                    CurrentRow := CurrentRow + 1;
                    If CurrentRow > N, send finish message to next node;
         process :  Process my columns using data in Vector;
                    CurrentRow := CurrentRow + 1;
         finish :   Terminate;
      END CASE;
   END LOOP;
• Send result to host
```

Figure 3: The parallel algorithm

left-hand side, with all zeros except the element on the diagonal. The column produced will not be used in the remaining part of the elimination process. Therefore, to start with, Processor 1 will have less and less work to do, and eventually has no data to process after N/P iterations, and then it happens to Processor 2, and so on. On the average, each processor works at most half of the processing time. In other words,the processor efficiency is at most 50%.

The following is a much more efficient data partitioning which causes the processing loads being balanced throughout the entire process. The data partitioning is illustrated in Figure 2.

The columns are still evenly distributed among the processors. The difference is that instead of consecutive columns, each processor has columns with numbers N/P apart. The i-th processor has the columns with numbers $n \times P + i$, where $n = 0, 1, 2, .., (N/P - 1)$.

The process is similar to the one described before except that the sequence in which the columns are processed. Processor 1 will start the first iteration, and Processor 2 will start the second iteration, and so on. After the last processor, Processor P, finishes the P-th iteration, Processor 1 will start the $(P+1)$-th iteration. The process repeats until all the iterations are done.

As mentioned above, when a processor initiates an iteration, it will have one more column which will never be used again. Since the processors initiate iterations in turn, their data will shrink evenly. The maximum difference in number of columns yet to be used on the processors is 1. Since number of columns on each processor is large, the difference is negligible to the overall performance.

Figure 3 shows the algorithm in Pascal-like pseudo code. To simplify the discussion, the row pivoting part of the algorithm is omitted. Each processor has two status variables, *CurrentRow* and *CurrentColumn*. *CurrentRow* is equal to the number of iterations that have been completed, and it has the same value for all processors. *CurrentColumn* is the number of the column to be passed if the processor is to initiate an iteration. The messages passing between the processors has the following format:

Message-type	Distance	Vector

Message-type has the following values: *initiate, process, finish.* Distance indicates the number of nodes that the message yet to travel through. A processor receives a message with distance 1 will not pass it along. The third part, Vector is present only when Message-type=*process*. In that case, it will contain the column necessary to process the columns held by the processor.

Although a horizontal partitioning (by rows) is possible and is advantageous in some cases, it makes the row pivoting difficult and inefficient.

4 Experimental data

The algorithm has been implemented and tested on a system consists of INMOS Transputer T800 and T414 processors. The transputer is a 32 bits RISC processor with a peak speed of 7.5 MIPS. Each transputer has its own memory module. The inter-processor communication is done through the transputer links. Each link consists two serial channels, one in each direction. The serial channel has a top speed of 20 Mega-baud. The effective data transfer rate is about 6 Mega-bit per second or 800 Kbyte per second. Each transputer has four such links. The processors are linked into a ring network in our experiments.

The performance data have been generated on both the single processor system and the multi-processor system. The algorithms in the two cases are slightly different to ensure that the implementations are optimal under each case. The code has been written in Occam.

The following table shows the execution times under different data sizes (number of columns in the matrix M) and number of processors. The processors are the INMOS T800 transputers. To facilitate the comparison, the data sizes have been selected to be multiples of 8.

Data size	1 processor	4 processors	8 processors
160	12.35 sec.	3.11 sec.	1.74 sec.
200	24.06 sec.	6.11 sec.	3.34 sec.
248	45.76 sec.	11.47 sec.	6.19 sec.

The times of speed-up comparing to the single processor system are presented in the following table. The processors efficiencies under each case are also included in the table (in parentheses). The processor efficiency is defined as $E=$ (factors of speed-up) / (number of processors).

Data size	4 processors	8 processors
160	3.97 (99.3%)	7.09 (88.7%)
200	3.94 (98.4%)	7.20 (90.0%)
248	3.99 (99.7%)	7.39 (92.4%)

As can be seen in the table, the speed-up is close to the number of processors used, especially in the case of 4 processors. As expected, when the number of processors increases to 8, the efficiency decreases due to the heavier interprocessor communication. Also, when the data size increases, the efficiency is higher since the CPU computation time out-weights the interprocessor communication time.

Those who have some experiences with parallel computing will probably suspect some of the extremely high efficiency figures shown in the table, 99.7% for instance. As a matter of fact, in a separate experiment[4], we observed an even higher "efficiency". In that experiment, the linear equations to be solved are under a residue system, i.e. all the arithmetic operations are under a residue system with moduli m. For example, if $m = 17$, then $5 \times 6 = 13$. Such a system is used to solve equations with residue arithmetic[6]. Under such system, the Gauss-Jordan elimination process is very similar, except that all the operations are integer operations and the division operation is replaced by multiplying by the "multiplicative inverse", which, as far as we are concerned, are some integer multiplications and divisions. When the size of the linear equations is 95 and T414 transputers are used, the execution times are 10.95 seconds for one processor and 2.44 seconds for 4 processors. A 4.49 times speed-up with 4 processors and a literally 112% "efficiency"!

This seemingly impossible result is no mistake. Researchers of INMOS Corporation made a similar observation on a graphics system where multiple processors are used. The explanation is the existence of high speed on-chip cache memory of the transputers. When the data is distributed over a number of processors, the data size for each processor is much smaller than that in the case of a single processor. Therefore, the "cache hit rate" can be much higher, and the effective speed of each processor is in fact higher than that of the single processor system.

Such phenomenon is not consistent from problem to problem. It depends on the memory reference pattern and the CPU computation load. We have compared the previous two experiments, the one with floating point operations and the other with integer operations. Since the floating point operations are time consuming, the CPU computation time is the dominant factor during the execution. The memory reference is less influential there. On the other hand, in the experiment involving only integer operations, the memory reference carries more weight. The extra speed-up phenomenon is more likely to happen in this case.

5 Timing analysis

Figure 4 shows the timing patterns when the algorithm runs on a system with 4 processors linked into a one-directional ring network.

Note that the drawing is only illustrative and it was not drawn to the exact scale. t_1 is the time used to pass a column from one processor to another; t_2 is the time used to process the columns on a processor during an iteration, which is the productive time; t_3 is the processor idle time and is approximately $P \times t_1$, where P is the number of processors. When the number of processors grows, t_3 will increase and the processor efficiency will decrease. The processor efficiency is:

$$E = \frac{t_2}{t_1 + t_3 + t_2} = \frac{t_2}{(P + 1) \times t_1 + t_2}$$

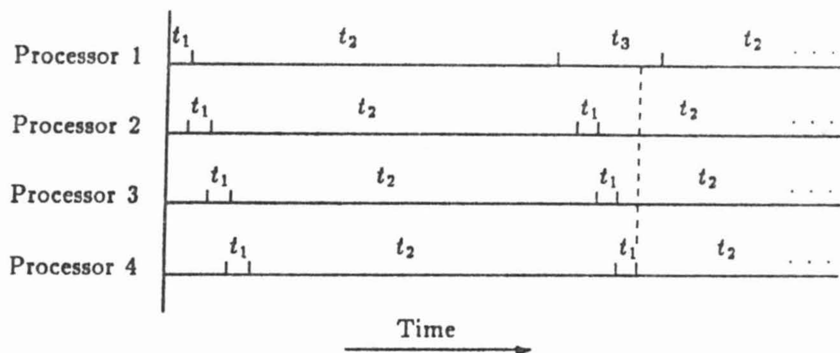


Figure 4: Timing patterns of the algorithm on a 4-processor one-directional ring network

When processors are linked into a topology with higher connectivity, t_3 can be much shorter, since some of the t_1's can be overlapped. In general,

$$t_3 = L \times t_1$$

and the processor efficiency is:

$$E = \frac{t_2}{t_1 + t_3 + t_2} = \frac{t_2}{(L+1) \times t_1 + t_2}$$

where L is the longest distance between two processors in a network. For example, on a system of P processors linked into a hypercube network, $t_3 = log_2 P \times t_1$. A better performance over the ring network is expected.

6 Conclusion

A practical algorithm for solving large scale linear equations on a parallel system has been presented. The speed-up can be close to a factor of the number processors. A better connected network can produce higher performance. An immediate candidate is the hypercube network. Since the algorithm treats the processors in a sequence, a sequence needs to be defined among the processors in the hypercube. Such a definition is available [5].

Although the algorithm is designed for a distributed memory system, it can be adapted into a shared-memory system. Since the algorithm tends to reference different modules or parts of the memory exclusively, it will reduce the memory contention. Since the inter processor communication time is reduced drastically, the algorithm is expected to work even better in a shared-memory system.

References

[1] R.L. Burden and J.D. Faires. *Numerical Analysis.* PWS-KENT Publishing Company, Boston, Massachusetts, 4th edition, 1988.

[2] INMOS Corporation. *Transputer Architecture Reference Manual.* INMOS Corporation, Bristol, U.K., 1986.

[3] J. McClelland D. Rumelhart and The PDP Research Group. *Parallel Distributed Processing.* Volume 1, The MIT Press, Cambridge, Massachusetts, 1986.

[4] Q Li, W.B. Feild, and D. Klein. An method for solving linear equations on a transputer system. In *The 3rd US Occam User Group Meeting*, Chicago, IL, Sept 1987.

[5] N. Rishe, D. Tal, and Q. Li. A sequenced hypercube topology for a massively-parallel database computer. In Proceedings of Second Symposium on the Frontiers of Massively Parallel Computation, Oct. 1988. Fairfax, Va.

[6] D. Young and R. Gregory. *A Survey of Numerical Mathematics.* Volume 2, Addison-Wesley Publishing Company, Inc., 1973.