Parallel Algorithms and Applications, Vol. 3, pp. 135–143
Reprints available directly from the publisher
Photocopying permitted by license only

EXACT SOLUTION OF LINEAR EQUATIONS ON DISTRIBUTED-MEMORY MULTIPROCESSORS

Ç. K. KOÇ, A. GÜVENÇ and B. BAKKALOĞLU

Department of Electrical and Computer Engineering, Oregon State University, Corvallis, OR 97331, USA

(Received October 11, 1993)

We present two new parallel algorithms for exact (error-free) solution of a system of linear equations on a distributed-memory multiprocessor. The exact solution is obtained using the congruence technique which consists of two steps: First, the system of linear equations is converted to systems of linear congruence equations with respect to several prime moduli, and each of these systems is solved on a separate processor. Then, these solutions are combined using the mixed-radix conversion algorithm to obtain the exact solution. The first step is completely (embarrassingly) parallel with no communication requirements among the processors. We improve our previous work and describe two efficient parallel algorithms for the second step. We present the results of our experiments on an Intel iPSC/860 with 8 processors. A linear system of dimension 128 with integer entries as large as 10^{577} is solved in about 195 seconds on 8 processors with an efficiency of 99.76%.

KEY WORDS: Congruence technique, parallel algorithm, mixed-radix conversion, single-node and multi-node broadcast.

C.R. CATEGORIES: E2.1, G.1.0, G.1.3.

1. INTRODUCTION

Exact solutions of a system of linear equations with integer or rational number entries can be found by using Gaussian elimination and multiple-precision arithmetic. However, this method breaks down even for systems of moderate size due to the so-called *intermediary coefficients swell*, i.e., excessive growth of the intermediate results even though the initial values as well as the final result have manageable size. Such situations often arise in scientific computing [2, 14] and in digital signal processing [5], where systems of equations with integer or rational number matrices need to be solved. Roundoff errors may make it impossible to solve an ill-conditioned problem, or it may cause a real-time digital signal processing system to be unstable. There are two viable techniques for dealing with such problems: the congruence technique and the p-adic expansions. Although some algorithmic questions remain, the basic mathematics of residue number and p-adic arithmetic are both well-known [7, 4, 14]. Efficient sequential algorithms and software for solving linear equations using the residue [12, 6, 2] and the p-adic [3] techniques have been developed in the last twenty years. Recently, parallel algorithms for exact solution of linear systems using the congruence [10, 8] and the p-adic techniques [13] have also been designed and implemented.

In this paper, we extend our previous work [10, 8] on parallelization of the congruence technique, and describe efficient parallel algorithms suitable for implementation on distributed-memory multiprocessors.

We consider the solution of the system of linear equations

$$\mathbf{Ax} = \mathbf{b}, \tag{1}$$

where $\mathbf{A}$ is a $k \times k$ invertible matrix and $\mathbf{b}$ is a k vector with all integer entries. Since the solution vector $\mathbf{x}$ will in general have rational number entries, we utilize the following trick of solving the linear system

$$\mathbf{Az} = d\mathbf{b}, \tag{2}$$

where $d = \det(\mathbf{A})$. In this case, the entries of the solution vector $\mathbf{z}$ will be integers, and the final solution vector $\mathbf{x}$ is obtained using floating-point arithmetic:

$$\mathbf{x} = \frac{1}{d}\mathbf{z}. \tag{3}$$

2. THE CONGRUENCE TECHNIQUE

In order to solve for Equation (2), we pick n prime numbers $m_1, m_2, \ldots, m_n$ such that their product M is larger than the largest entry of the matrix $\mathbf{A}$. The congruence method consists of two main steps:

1. Solve the n systems $\mathbf{Ay}_i = \mathbf{b} \pmod{m_i}$ using Gaussian elimination and also compute the determinant $d_i = \det(\mathbf{A}) \pmod{m_i}$ for $i = 1, 2, \ldots, n$. The solution of Equation (2) modulo m_i is obtained as $\mathbf{z}_i = d_i \mathbf{y}_i$ for $i = 1, 2, \ldots, n$.

2. Use the mixed-radix conversion algorithm to combine the solution vectors $\mathbf{z}_i$ into a single vector $\mathbf{z}$ such that $\mathbf{z} = \mathbf{z}_i \pmod{m_i}$ for $i = 1, 2, \ldots, n$. The solution of Equation (1) is then obtained as $\mathbf{x} = (1/d)\mathbf{z}$.

The first step of the congruence technique is completely parallel since the solution of equation

$$\mathbf{Ay}_i = \mathbf{b} \pmod{m_i}$$

is independent for every $i = 1, 2, \ldots, n$. Assuming the number of processors p is equal to n, we allocate processor i for modulo m_i computations: Processor i solves the system $\mathbf{Ay}_i = \mathbf{b} \pmod{m_i}$ and computes the determinant $d_i = \det(\mathbf{A}) \pmod{m_i}$, and then proceeds to compute $\mathbf{z}_i = d_i \mathbf{y}_i \pmod{m_i}$. This computation is simultaneously performed by all processors for $i = 1, 2, \ldots, n$. In general, given $p \leq n$ processors, we partition the moduli set in such a way that each processor receives at most q moduli where $q = \lceil n/p \rceil$. The above computations are then repeated q times. Since Gaussian elimination on a matrix of dimension k requires $O(k^3)$ arithmetic steps, we have the following theorem:

THEOREM 1 *Given $p \leq n$ processors, Step 1 of the congruence method requires $O(k^3 n/p)$ arithmetic steps.*

At the end of Step 1, we will have a k vector $\mathbf{z}_i$ and an integer d_i in processor i for all $i = 1,2,\ldots,n$. We now need to apply the mixed-radix conversion algorithm to compute a k vector $\mathbf{z}$ and an integer d. Let the $(k+1)$ vector $\mathbf{u}_i$ be

$$\mathbf{u}_i = \begin{bmatrix} \mathbf{z}_i \\ d_i \end{bmatrix}.$$

We use the mixed-radix conversion algorithm to compute the $(k+1)$ vector $\mathbf{u}$ such that

$$\mathbf{u} = \mathbf{u}_i \pmod{m_i}.$$

The mixed-radix conversion algorithm returns the vector $\mathbf{u}$, from which we extract the value of the determinant d and the elements of the vector $\mathbf{z}$. The final solution vector $\mathbf{x}$ is then computed using Equation (3).

3. MIXED-RADIX CONVERSION

Given the n vectors $\mathbf{u}_i$ (of dimension $k+1$) for $i = 1,2,\ldots,n$, the vector mixed-radix conversion algorithm computes a single vector $\mathbf{u}$ (of dimension $k+1$) such that

$$\mathbf{u} = \mathbf{u}_i \pmod{m_i}$$

for $i = 1,2,\ldots,n$. This is achieved in two steps. First, the values $\mathbf{u}_i$ are updated according to the following recursion:

The Sequential MRC Algorithm

```
for j = 1 to n − 1 do
    for i = j + 1 to n do
        u_i := (u_i − u_j)c_ij  (mod m_j)
```

where c_{ij} are the multiplicative inverses of m_i modulo m_j for $1 \le i < j \le n$, computed using the extended Euclidean algorithm [7, 11]. After the above update process, the elements of the vectors $\mathbf{u}_i$ are the mixed-radix coefficients of the elements of the final vector $\mathbf{u}$. Thus, the final vector $\mathbf{u}$ is obtained by computing

$$\mathbf{u} = \mathbf{u}_1 + m_1\mathbf{u}_2 + m_1m_2\mathbf{u}_3 + m_1m_2m_3\mathbf{u}_4 + \cdots + m_1m_2\cdots m_{n-1}\mathbf{u}_n. \tag{4}$$

Since the mixed-radix conversion of n integers requires $O(n^2)$ arithmetic operations [7, 11], we obtain the following result:

THEOREM 2 *The number of arithmetic operations required by the sequential MRC algorithm is* $O(kn^2)$.

The data dependences among the computations required to obtain the final vector $\mathbf{u}$ lend themselves to systolic implementation. This is achieved by first forming the data dependence graph of the algorithm, and then embedding this graph in space-

time in order to obtain time-optimal and spacetime-optimal systolic schedules. Several systolic schedules with the above properties are described in [9]. The parallel algorithms given in [10, 8] are derived from these systolic schedules. Although these algorithms were implemented on an Intel iPSC1 hypercube, they require only linear or ring connectivity among the processors, and therefore, do not fully exploit the hypercube connectivity. In the following, we describe two new parallel algorithms which are particularly suitable for implementation on distributed-memory architectures.

4. PARALLEL ALGORITHMS

We will describe two parallel algorithms which are termed as the *Single-Node Broadcast* (SNB) algorithm and the *Multi-Node Broadcast* (MNB) algorithm. Step 1 of the congruence method is parallelized exactly the same way for both of these algorithms. These algorithms differ only in the way the computational and communication requirements of Step 2 of the congruence method are handled.

4.1. Single-Node Broadcast

At the end of Step 1, we have the vector $\mathbf{u}_i$ in processor i for all $i = 1,2,\ldots,n$. The SNB algorithm is a direct parallel implementation of the sequential MRC algorithm, and goes through $n-1$ steps for $j = 1,2,\ldots,n-1$ where at step j processor j broadcasts its vector $\mathbf{u}_j$ to all processors whose index is larger than j. These processors update their $\mathbf{u}$ vectors with the vector $\mathbf{u}_j$ received. When the update process is completed, the last processor contains a copy of each of the $\mathbf{u}_i$ vectors for $i = 1,2,\ldots,n$. This processor then uses Equation (4) and Equation (3) in floating-point arithmetic to compute the vector $\mathbf{x}$. The pseudocode below describes these operations:

The SNB Algorithm

	for $j = 1$ **to** $n-1$ **do**
	begin
PROC j:	**send** $\mathbf{u}_j$ **to** PROC $j+1,\ldots,n$
PROC i:	$\mathbf{u}_i := (\mathbf{u}_i - \mathbf{u}_j)c_{ij} \pmod{m_i}$
$(i = j+1,\ldots,n)$	**end**
PROC n:	**compute** $\mathbf{x}$ using Eqs. (4) and (3)

This way we parallelize the i-loop of the sequential MRC algorithm, and obtain the following result:

THEOREM 3 *The SNB algorithm requires $O(kn)$ arithmetic steps with n processors.*

In order to calculate the communication penalty, we notice that at each step a single-node broadcast of a vector of dimension $k+1$ is performed. It is well-known that a single-node broadcast operation requires $\log_2 p$ routing steps on a hypercube architecture with p processors [1]. The algorithm is based on the spanning tree of the hypercube graph, rooted at the node which broadcasts its data. At each step a single-node broadcast of a vector of dimension $k+1$ is performed, however, the number of processors, to which the data is sent, decreases by one. Thus, we obtain the communication penalty of the SNB algorithm as

$$(k+1)\sum_{i=n-1}^{1}\lceil \log_2 i\rceil = O(kn\log n).$$

THEOREM 4 *The SNB algorithm requires $O(kn\log n)$ routing steps on a hypercube with n processors.*

4.2. Multi-Node Broadcast

The SNB algorithm interleaves the computational and communication steps of the mixed-radix conversion algorithm. The MNB algorithm first performs all necessary communications; it then proceeds to compute the elements of the **u** vector using the sequential MRC algorithm with the locally available data. In the beginning of the multi-node broadcast we have $\mathbf{u}_i$ in processor i for $i = 1,2,\ldots,n$. At the end, all processors have all of the vectors $\mathbf{u}_i$. If $k = n = p$, then processor i picks the ith and the last element (corresponding to the determinant) of the vectors $\mathbf{u}_1,\mathbf{u}_2,\ldots,\mathbf{u}_n$, and then performs two *sequential* scalar mixed-radix conversion operations in order to obtain the mixed-radix coefficients of the element $\mathbf{z}[i]$ and the determinant d, respectively. It then computes $\mathbf{x}[i]$ using Equations (4) and (3) in floating-point arithmetic.

The MNB Algorithm

	for $j = 1$ **to** n **do**
PROC j:	**send** $\mathbf{u}_j$ **to** PROC $1,\ldots,n$ except j
PROC i:	**compute** $\mathbf{z}[i]$ using sequential MRC
$(i = 1,\ldots,n)$	**compute** d using sequential MRC
	compute $\mathbf{x}[i]$ using Eqs. (4) and (3)

If $k \geq n = p$, then processor i performs $\lceil k/n\rceil$ mixed-radix conversions and obtains as many elements of the vector **z**. Thus, the number of mixed-radix conversions to be performed by processor i is equal to $\lceil k/n\rceil + 1$, which gives the total number of arithmetic operations as $O(n^2k/n) = O(kn)$.

THEOREM 5 *The MNB algorithm requires $O(kn)$ arithmetic steps with n processors.*

Table 1 Data Distribution in the Multi-Node Broadcast Algorithm

PROC.	000	001	010	011	100	101	110	111
Initial	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*
Step 1	*a* b	a b	c d	c d	e f	e f	g h	g h
Step 2	a b c d	a b c d	a b c d	a b c d	e f g h	e f g h	e f g h	e f g h
Step 3	a b c d e f g h	a b c d e f g h	a b c d e f g h	a b c d e f g h	a b c d e f g h	a b c d e f g h	a b c d e f g h	a b c d e f g h

Furthermore, the MNB algorithm needs to perform n broadcast operations in order to distribute the data to all processors. A naive method of accomplishing this task is to perform n sequentially-arranged single-node broadcast operations. A better strategy is to perform simultaneous broadcast operations in order to achieve maximum concurrency. Details of the multi-node broadcast operation on a hypercube computer can be found in [1]. We briefly explain the algorithm here: We assume the nodes of the d-dimensional hypercube are indexed using the binary numbers $(a_1 a_2 \cdots a_d)$ for $a_i \in \{0,1\}$. The multi-node broadcast algorithm completes in d steps for $j = 1,2,\ldots,d$. At step j, the nodes with indices $(a_1 \cdots a_{j-1} 0 a_{j+1} \cdots a_d)$ exchange all of their data with the nodes whose indices are $(a_1 \cdots a_{j-1} 1 a_{j+1} \cdots a_d)$. That is, at step j, all the nodes whose binary indices differ in the jth bit exchange all of their data. As an example, let $d = 3$. In step $j = 1$, the following pairs of processors exchange their data: (000,001), (010,011), (100,101), and (110,111). In step $j = 2$, the amount of the data to be exchanged is doubled and the following pairs are exchanging: (000,010), (001,011), (100,110), and (101,111). Finally, in step $j = 3$, pairs of processors exchanging their data are (000,100), (001,101), (010,110), and (011,111). The data distribution at the end of each step is illustrated in Table 1. Since there are exactly d steps and the amount of the data exchanged is doubled at each step, the total number of routing operations required to broadcast all vectors $\mathbf{u}_i$ is found as

$$(k+1)\sum_{k=1}^{d} 2^k = O(kn).$$

THEOREM 6 *The MNB algorithm requires $O(kn)$ routing steps on a hypercube with n processors.*

Table 2 Timings for the Sequential Algorithm

k	$n \rightarrow 8$	16	32	64	128
8	49	122	344	1100	3959
16	258	563	1313	3405	10136
32	1691	3470	7305	16123	38699
64	12315	24807	50332	103645	220005
128	94226	188802	379031	763978	1553183

Table 3 Timings for the SNB Algorithm

k	$n \rightarrow 8$	16	32	64	128
8	663	1268	2550	5198	10459
16	690	1377	2703	5458	11177
32	878	1752	3487	7066	14670
64	2209	4428	8879	18003	38033
128	12458	24950	50006	100832	207179

5. IMPLEMENTATION AND RESULTS

We have implemented the above algorithms on an Intel iPSC/860 multiprocessor with 8 processors, running the NX/860 operating system. The front-end processoris a 386-based computer that runs the UNIX System V operating system, augmented with iPSC system extensions and the networking software. The nodes of the system are equipped with 8 MBytes of memory each. Even though the architecture is based on the hypercube, with the Direct-Connect Module, it can be viewed as an ensemble of fully connected nodes with a uniform message latency. When a node sends a message to another node, the message goes directly to the receiving node without disturbing any of the other nodes. The message passing does not require any "store and forward."

In our experiments, we solve integer linear systems of equations with dimensions $k = 8$, 16, 32, 64, and 128. The number of prime moduli takes the values $n = 8$, 16, 32, 64, and 128. The prime numbers are selected such that the basic arithmetic operations $\{+, -, \times\}$ can be performed in single-precision integer arithmetic. Since the largest single-precision signed integer is equal to $2^{31} - 1$, we choose $m_i < 2^{15}$ in order to perform a modular multiplication operation. When the number of primes is equal to 128, the entries of the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ can be as large as $2^{15 \times 128} = 2^{1920} \approx 10^{577}$.

The values given in Tables 2, 3, and 4 are the average timings in milliseconds for the sequential algorithm, the SNB algorithm, and the MNB algorithm, respectively. The sequential time is found by running the congruence algorithm on a single node. A linear system of dimension 128 with integer entries as large as 10^{577} is solved in approximately 194.6 seconds by the MNB algorithm, while it is solved in 1,553.2 seconds by the sequential algorithm. This represents a speedup of 7.9815, or an efficiency of 99.76%. In Table 5, we list the efficiency values as a function of n

Table 4 Timings for the MNB Algorithm

k	$n \rightarrow 8$	16	32	64	128
8	264	277	318	470	1067
16	290	331	438	756	1794
32	470	695	1148	2343	5390
64	1796	3360	6560	13268	28015
128	12035	23858	47645	95802	194621

Table 5 Efficiency as a Function of n for $k = 128$

n	SNB	MNB
8	0.9454	0.9787
16	0.9459	0.9892
32	0.9475	0.9944
64	0.9471	0.9968
128	0.9371	0.9976

Table 5 Efficiency as a Function of k for $n = 128$

n	SNB	MNB
8	0.0473	0.4638
16	0.1134	0.7062
32	0.3298	0.8975
64	0.7231	0.9816
128	0.9371	0.9976

(the number of moduli) by fixing the system size at $k = 128$. Similarly, in Table 6, the efficiency values are listed as a function of k when n is fixed at 128. These efficiency values indicate the MNB algorithm as the fastest algorithm.

References

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

[2] S. Cabay and T. P. L. Lam, Congruence techniques for the exact solution of integer systems of linear equations, *ACM Transactions on Mathematical Software* **3**, 4 (December 1977), 386–397.

[3] J. D. Dixon, Exact solution of linear equations using p-adic expansions, *Numerische Mathematik* **40**, 1 (1982), 137–141.

[4] R. T. Gregory and E. V. Krishnamurthy, *Methods and Applications of Error-Free Computation*, New York, NY: Springer-Verlag, 1984.

[5] B. Harms and S. Keller-McNulty, Error-free solution to a Toeplitz system of equations, *IEEE Transactions on Signal Processing* **39**, 5 (May 1991), 1212–1215.

[6] J. A. Howell and R. T. Gregory, An algorithm for solving linear algebraic equations using residue arithmetic I–II, *BIT* **9**, 3–4 (1969), 200–224 and 324–337.

[7] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Reading, MA: Addison-Wesley, Second edition, 1981.

[8] Ç. K. Koç, A parallel algorithm for exact solution of linear equations via congruence technique, *Computers and Mathematics with Applications* **23**, 12 (1992), 13–24.

[9] Ç. K. Koç and P. Cappello, Systolic arrays for integer Chinese remaindering. In M. D. Ercegovac and E. Swartzlander, eds., *Proceedings, 9th Symposium on Computer Arithmetic*, pages 216–223, Santa Monica, CA, September 6–8, 1989. Los Alamitos, CA: IEEE Computer Society Press.

[10] Ç. K. Koç and R. M. Piedra, A parallel algorithm for exact solution of linear equations. In *Proceedings of International Conference on Parallel Processing*, Volume III, pages 1–8,St. Charles, IL, August 12–16, 1991. Boca Raton, FL: CRC Press.

[11] J. D. Lipson, *Elements of Algebra and Algebraic Computing*, Reading, MA: Addison-Wesley, 1981.

[12] M. Newman, Solving equations exactly, *Journal of Research of the National Bureau of Standards* **71B**, 4 (Oct.–Dec. 1967), 171–179.

[13] G. Villard, Exact parallel solution of linear systems. In J. Della Dora and J. Fitch, eds., *Computer Algebra and Parallelism*, pages 197–205. New York, NY: Academic Press, 1989.

[14] D. M. Young and R. T. Gregory, *A Survey of Numerical Mathematics*, volume 2. New York, NY: Dover Publications, 1988.