A Parallel Algorithm for Exact Solution of Linear Equations

Çetin Kaya Koç and *Rose Marie Piedra*
Department of Electrical Engineering
University of Houston
Houston, TX 77204

Abstract

We present a parallel algorithm for computing the exact solution of a system of linear equations via the congruence technique. The basic idea of the technique is to convert the original system of equations into a system of congruences modulo various primes, and combine the solutions by the application of the Chinese remainder theorem. The proposed parallel algorithm requires only local communication among the processors and is particularly suitable for implementation on distributed-memory multiprocessors and systolic computing systems. We have implemented the parallel congruence algorithm on an Intel cube and obtained up to 92 % efficiency.

Key Words: Congruence techniques, Chinese remainder theorem, Gaussian elimination, mixed-radix conversion algorithm, systolic schedule.

1 Introduction

We consider the solution of the system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \ , \tag{1}$$

where $\mathbf{A}$ is a $k \times k$ invertible matrix, and $\mathbf{x}$ and $\mathbf{b}$ are $k \times 1$ vectors. This problem has been studied intensively. There are several methods to solve (1), most notably the Gauss and the Gauss-Jordan elimination methods. There are situations, however, in which these methods are inadequate; for example, when one is interested in the *exact* solution of (1) or when $\mathbf{A}$ is ill-conditioned.

The exact solution of (1) can be found by either the direct computation method using multiple-precision integer arithmetic or the residue arithmetic techniques. The residue arithmetic techniques find the result by either using multiple moduli (the congruence technique) or single modulus (p-adic expansions). In general, these methods require that the entries of $\mathbf{A}$ and $\mathbf{b}$ be integers. However, this is not a serious restriction, since the rational number or the floating-point number entries of $\mathbf{A}$ and $\mathbf{b}$ can be converted to integers by scaling.

Algorithms using p-adic expansions are given by Dixon [7] and by Gregory and Krishnamurthy [10]. Parallel implementations of the p-adix expansion and the direct computation methods are given by Villard [20]. In this paper, we focus on parallel implementation of the congruence technique which solves (1) by first solving $n+1$ such systems in $\mathcal{Z}_{m_i}$ (the ring of integers modulo m_i) for $0 \leq i \leq n$. These results are then combined using the Chinese remainder theorem to find the solution in $\mathcal{Z}_M$ where $M = m_0 m_1 m_2 \cdots m_n$. The reader is referred to the books by Knuth [13], Lipson [17], Mackiw [18], and Young and Gregory [21] (and the references therein) for discussions on the congruence technique and various algorithms for computing exact solution of linear equations with integer and rational entries.

We would like to state the following assumptions for the timing analysis of the algorithm presented in this paper.

- We choose the moduli $m_i < W$ for $0 \leq i \leq n$. W is the wordsize of the computer, i.e., the largest integer which can be operated on by the arithmetic unit of the computer.
- An arithmetic operation ($\in \{+, -, \times\}$) on integers $< W$ is assumed to take 1 unit of time, defined as an arithmetic step.
- For operations on integers $> W$, we implement multi-precision arithmetic.
- The multiplicative inverse of $a < W$ in $\mathcal{Z}_b$ exists if $\gcd(a, b) = 1$. It is defined as the integer $x < W$ such that

  $$ax = 1 \pmod{b} \ ,$$

  and can be computed using the extended Euclid algorithm [13, 17]. We denote the operation to find the inverse of a in $\mathcal{Z}_b$ by INVERSE (a, b). If $a, b < W$, then Euclid's algorithm requires $O(\log W)$ arithmetic operations to compute INVERSE (a, b). Since W is independent of n, we assume that INVERSE operation on single-precision numbers takes only $O(1)$ arithmetic operations.
- Our computer is also capable of carrying out arithmetic operations ($\in \{+, -, \times, \div\}$) on floating-point numbers. A floating-point arithmetic operation is also assumed to take 1 step.

For the parallel implementation of the congruence technique, we assume that we have a distributed-memory multiprocessor with p processors. The processors are connected with communication links, forming a network topology. Examples of network topologies are linear arrays, rings, trees, 2-dimensional and multi-dimensional meshes, and hypercubes. The communication cost of the parallel algorithm is

measured by counting the total number of parallel routing steps, where a routing step is defined as the time required to send an operand (i.e., a single-precision integer) from a processor to one of its neighboring processors. If two communicating processors are not adjacent then the data is assumed to be forwarded by the processors on a path between these two processors, and the routing cost is taken to be the length of the path.

We describe the congruence technique in §2. When the solutions in $\mathcal{Z}_{m_i}$ are combined using the Chinese remainder theorem to find the solution in $\mathcal{Z}_M$, we may pick either the single-radix conversion algorithm or the mixed-radix conversion algorithm. The mixed-radix conversion algorithm, described in §3, is chosen for parallel implementation of the congruence technique. In §4, we present the parallel congruence algorithm and time-optimal and spacetime-optimal systolic schedules for parallel implementation of the mixed-radix conversion algorithm. We have implemented the congruence technique and obtained up to 92 % efficiency on an first generation Intel cube which is a very slow machine in terms of interprocessor communication. In §5, we report the implementation results. Finally, in §6 we discuss several variations of the parallel congruence algorithm and point out some future topics for research.

2 Congruence Algorithm

Let $m_0, m_1, m_2, \ldots, m_n$ be a set of pairwise relatively prime numbers and $M = m_0 m_1 m_2 \cdots m_n$. We denote the determinant of A by $d = \det(\mathrm{A})$ and the adjoint of A by $\mathrm{A}^{\mathrm{adj}}$. Since we assume $d \neq 0$, $\mathrm{A}^{\mathrm{adj}}$ is also a nonsingular integral matrix satisfying

$$\mathrm{A}\mathrm{A}^{\mathrm{adj}} = \mathrm{A}^{\mathrm{adj}}\mathrm{A} = d\,\mathrm{I} \ ,$$

where I is the $k \times k$ unit matrix. For a matrix $\mathrm{B} = [b_{ij}]$, we also define $\mathrm{MAX}(\mathrm{B}) = \max_{i,j} |b_{ij}|$. The solution of (1) can be written as

$$\mathrm{x} = \frac{1}{d}\mathrm{A}^{\mathrm{adj}}\mathrm{b} = \frac{1}{d}\mathrm{z} \ ,$$

where $\mathrm{z} = \mathrm{A}^{\mathrm{adj}}\mathrm{b}$.

The Congruence Algorithm

Step 0. Choose the moduli set $m_0, m_1, \ldots, m_n$ such that $M > 2\max(|d|, |\mathrm{MAX}(\mathrm{A})|)$ and $\gcd(M, d) = 1$.

Step 1. Solve the $n+1$ systems $\mathrm{A}\mathrm{y}_i = \mathrm{b} \pmod{m_i}$ for $0 \leq i \leq n$. Compute the determinant $d_i = \det(\mathrm{A}) \pmod{m_i}$ for $0 \leq i \leq n$. Also compute $\mathrm{z}_i = d_i \mathrm{y}_i \pmod{m_i}$ for $0 \leq i \leq n$.

Step 2. Use the Chinese remainder theorem to solve the simultaneous congruences $\mathrm{z} = \mathrm{z}_i \pmod{m_i}$ for $0 \leq i \leq n$. Also, the simultaneous congruences $d = d_i \pmod{m_i}$ for $0 \leq i \leq n$ are solved by the application of the Chinese remainder theorem.

Step 3. The solution of (1) is found as $\mathrm{x} = \frac{1}{d}\mathrm{z}$.

We have the following observations:

- The choice of m_i such that the conditions in Step 0 hold may be a difficult and time consuming process. It is possible to estimate the determinant by the use of Hadamard inequality. We then have to choose m_i such that $\gcd(m_i, d) = 1$ for $0 \leq i \leq n$. Newman suggests the use of a predetermined prime moduli set, with the full knowledge that in certain instances the method will fail [19]. By choosing large primes, the probability of the occurrence $\gcd(m_i, d) > 1$ can be made very small [19, 21].

- In Step 1, the operations are performed using single-precision integer arithmetic. We choose $m_i < W$ for $0 \leq i \leq n$, and then implement the Gaussian elimination algorithm in single-precision integer arithmetic. As matrix A is triangularized, determinant $d_i = \det(\mathrm{A}) \pmod{m_i}$ is also computed in single-precision.

- In Step 2, the Chinese remainder theorem is used to find the weighted-radix representation of the residue numbers d_i and z_i for $0 \leq i \leq n$. The methods for conversion of a residue number to a weighted number system are based on two different constructive proofs of the Chinese remainder theorem. In the first case, the number is converted to a *single-radix* weighted number system, whereas in the second case it is converted to a *mixed-radix* weighted number system. Since during the computation of d or z, the intermediate and the resulting values can be larger than W, the single-radix conversion algorithm requires implementation of multi-precision integer arithmetic.

- In Step 3, floating-point arithmetic is used to compute the solution x

$$\mathrm{x} = [x_0, x_1, \ldots, x_{k-1}]^T = \frac{1}{d}[z_0, z_1, \ldots, z_{k-1}]^T$$

Theorem 1 *The congruence algorithm finds the solution of (1) using $O(nk^3 + n^2k)$ arithmetic steps.*

Proof The solution of the systems

$$\mathrm{A}\mathrm{y}_i = \mathrm{b} \pmod{m_i}$$

and the computation of the determinant

$$d_i = \det(\mathrm{A}) \pmod{m_i}$$

for $0 \leq i \leq n$ are achieved by the use of the Gaussian elimination algorithm. First the coefficient matrix is triangularized; then backward-substitution is performed to solve for y_i. During these computations, multiplicative inverses of the nonzero elements of A are computed using the extended Euclid algorithm. It becomes evident that one such system is solved in $O(k^3)$ arithmetic steps (see also, e.g., [19, 17]). Thus, the solution $n+1$ of these systems requires $O(nk^3)$ arithmetic steps. Since the computation of $\mathrm{z}_i = d_i \mathrm{y}_i \pmod{m_i}$ requires $O(nk)$ arithmetic steps for

$0 \leq i \leq n$, the number of arithmetic operations required in Step 2 is $O(nk^3)$.

In Step 2, we use either the single-radix conversion algorithm or the mixed-radix conversion algorithm to find the weighted-radix representation of the numbers $\mathbf{z}_i$ and d_i for $0 \leq i \leq n$. The mixed-radix conversion algorithm (as well as the single-radix conversion algorithm) requires $O(n^2)$ arithmetic steps to find the integer d using the residue numbers $d_i = d \pmod{m_i}$ for $0 \leq i \leq n$. Since the vector $\mathbf{z}_i$ has k components, we see that Step 3 of the congruence algorithm requires $O(n^2k)$ arithmetic operations.

In Step 3, we compute the solution by performing k floating-point division operations. According to our assumption, this takes $O(k)$ arithmetic steps.

Thus, the total number of arithmetic operations required is found to be $O(nk^3 + n^2k)$. □

3 Mixed-Radix Conversion

We now describe the mixed-radix conversion algorithm and give a theorem regarding the number of arithmetic operations required. The detailed proof of this theorem can be found in [16, 2, 17]. We assume that we are given the residues u_i of a weighted number u with respect to each modulus m_i, i.e.,

$$u = u_i \pmod{m_i} \text{ for } 0 \leq i \leq n .$$

The mixed-radix conversion algorithm computes the mixed-radix coefficients $(v_0, v_1, \ldots, v_n)$ of u. Once the mixed-radix coefficients have been obtained, u is written in terms of these coefficients and the moduli as

$$\begin{aligned} u &= v_0 + v_1 m_0 + v_2 m_0 m_1 + v_3 m_0 m_1 m_2 + \cdots \\ &\quad \cdots + v_n m_0 m_1 \cdots m_{n-1} . \end{aligned} \tag{2}$$

The Mixed-Radix Conversion Algorithm

Step 1. Compute the inverses c_{ij} for $0 \leq i < j \leq n$ where

$$c_{ij} = \text{INVERSE}\ (m_i, m_j) .$$

Step 2. Set $v_0 = u_0 \pmod{m_0}$, and for $k = 1, 2, \ldots, n$ compute

$$\begin{aligned} v_k &= (\cdots((u_k - v_0)c_{0k} - v_1)c_{1k} - \cdots \\ &\quad \cdots - v_{k-1})c_{k-1,k} \pmod{m_k} . \end{aligned}$$

Theorem 2 *Given the moduli $m_0, m_1, \ldots, m_n$ and the remainders $u_0, u_1, \ldots, u_n$ such that $m_i < W$ for $0 \leq i \leq n$, the mixed-radix number representation $(v_0, v_1, \ldots, v_n)$ of u can be computed in $O(n^2)$ arithmetic steps with the mixed-radix conversion algorithm.*

The mixed-radix representation can be converted to single-radix representation by applying Horner's algorithm to formula (2). If $v_i, m_i < W$ for $0 \leq i \leq n$, then the application of Horner's algorithm to compute single-radix representation of u requires $O(n^2)$ arithmetic steps using multi-precision arithmetic [17].

4 Parallel Congruence Algorithm

A remarkable property of the congruence algorithm is its parallelism. Since the solution of equation

$$\mathbf{A}\mathbf{y}_i = \mathbf{b} \pmod{m_i}$$

is independent for every $i = 0, 1, 2, \ldots, n$, the algorithm is very suitable for implementation on parallel computers. Once the solutions in $\mathcal{Z}_{m_i}$ are computed, we need to apply a Chinese remaindering algorithm to compute the solution in $\mathcal{Z}_M$. Let $n + 1 = qp$ where $n + 1$ is the number of moduli, p is the number of processors and $q \geq 1$ is an integer. For the time being we assume that $q = 1$. We partition the moduli set in such a way that processor i executes Step 1 of the algorithm modulo m_i, and thus, solves system $\mathbf{A}\mathbf{y}_i = \mathbf{b} \pmod{m_i}$ and computes determinant $d_i = \det(\mathbf{A}) \pmod{m_i}$. It then proceeds to compute $\mathbf{z}_i = d_i \mathbf{y}_i \pmod{m_i}$. This computation is performed by all processors for $i = 0, 1, \ldots n$.

Thus at the end of Step 1, we will have a $k \times 1$ vector $\mathbf{z}_i$ and an integer d_i in processor i for all $0 \leq i \leq n$. We now need to apply the Chinese remainder theorem to compute a $k \times 1$ vector $\mathbf{z}$ and an integer d. Notice that if the single-radix conversion algorithm is implemented, then the components of $\mathbf{z}$ and the determinant d are multi-precision integers. If the mixed-radix conversion algorithm is employed then we can avoid the implementation of the multiple-precision arithmetic, and compute the mixed-radix coefficients in single-precision. We can then use floating-point arithmetic to compute the solution vector $\mathbf{x}$ with these mixed-radix coefficients.

Let the $(k + 1) \times 1$ vector $\mathbf{u}_i$ be

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

This vector is available in processor i. We now use the mixed-radix conversion algorithm to compute the $(k+1) \times 1$ vector $\mathbf{u}$ such that $\mathbf{u} = \mathbf{u}_i \pmod{m_i}$.

The distributed mixed-radix conversion algorithm picks the rth element of each vector $\mathbf{u}_i$ from processor i for $0 \leq i \leq n$ and computes the mixed-radix coefficients of the rth element of the vector $\mathbf{u}$ for all $r = 1, 2, 3, \ldots, k + 1$. Denote the rth component of vector $\mathbf{u}_i$ with u_{ir}. We need to compute v_{ir} for $0 \leq i \leq n$ and $1 \leq r \leq k + 1$ such that

$$\begin{aligned} u_{ir} &= v_{0r} + v_{1r} m_0 + v_{2r} m_0 m_1 + \cdots \\ &\quad + v_{nr} m_0 m_1 \cdots m_{n-1} \pmod{m_i} . \end{aligned}$$

We define the $(k + 1) \times 1$ vector $\mathbf{V}_{ij}$ for $0 \leq i < j \leq n$ such that $\mathbf{V}_{-1,i} = \mathbf{V}_i = \mathbf{u}_i$ for $0 \leq i \leq n$ and $\mathbf{V}_{i-1,i} = \mathbf{v}_i$ for $0 \leq i \leq n$. $\mathbf{V}_{ij}$ for $0 \leq i < j \leq n$ are the temporary values of $\mathbf{v}_j$ resulting from the operations in Step 2 of the mixed-radix conversion algorithm. We construct a triangular table of values with diagonal entries $\mathbf{v}_i = \mathbf{V}_{i-1,i}$ for $0 \leq i \leq n$. For $n = 3$, it is as follows:

$$V_{03} = (V_3 - V_0)c_{03} \pmod{m_3} \quad V_{13} = (V_{03} - V_{01})c_{13} \pmod{m_3} \quad V_{23} = (V_{13} - V_{12})c_{23} \pmod{m_3}$$

$$V_{02} = (V_2 - V_0)c_{02} \pmod{m_2} \quad V_{12} = (V_{02} - V_{01})c_{12} \pmod{m_2}$$

$$V_{01} = (V_1 - V_0)c_{01} \pmod{m_1}$$

The mixed-radix conversion algorithm computes V_{ij} for $0 \leq i < j \leq n$ by performing the following operations on single-precision integer operands:

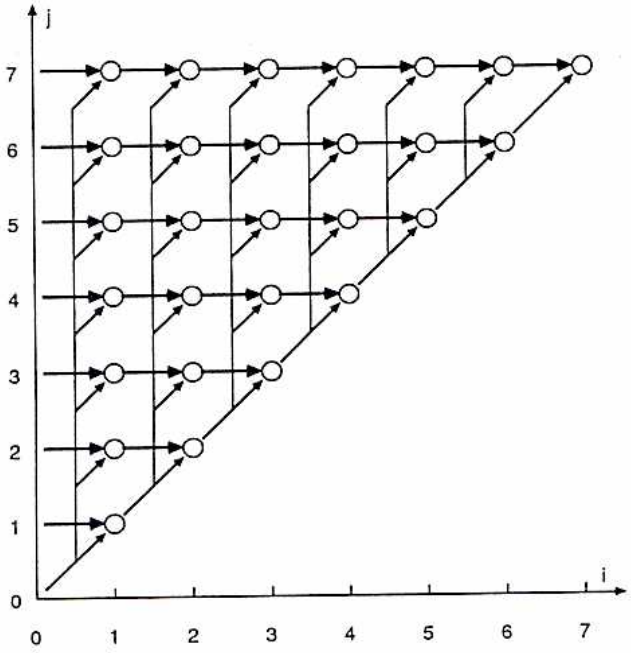
$$c_{ij} = \text{INVERSE}\,(m_i, m_j)\,, \tag{3}$$

$$V_{ij} = (V_{i-1,j} - V_{i-1,i})c_{ij} \pmod{m_j}\,, \tag{4}$$

where (4) is performed for all $k+1$ entries of vector V_{ij}.

The data dependences among the entries in the above table lend themselves to systolic implementation. The first step in achieving a systolic implementation is to form the *process dependence graph* of the mixed-radix conversion algorithm. Coefficient c_{ij} is in column i and row j. The positions of terms V_{ij} are arranged as follows: First, a term of the form $V_{i-1,i}$ is computed on the diagonal, then this term is used in every operation along the ith column. Based on these observations, Figure 1 presents the process dependence graph of the mixed-radix conversion algorithm for $n = 7$. The graph is drawn on the (i, j) coordinate system. The nodes of this directed acyclic graph (dag) represent the operations, and the arcs correspond to dependences between the variables used in the operations. The node at point (i, j) computes V_{ij} by performing the operations given in (3) and (4).

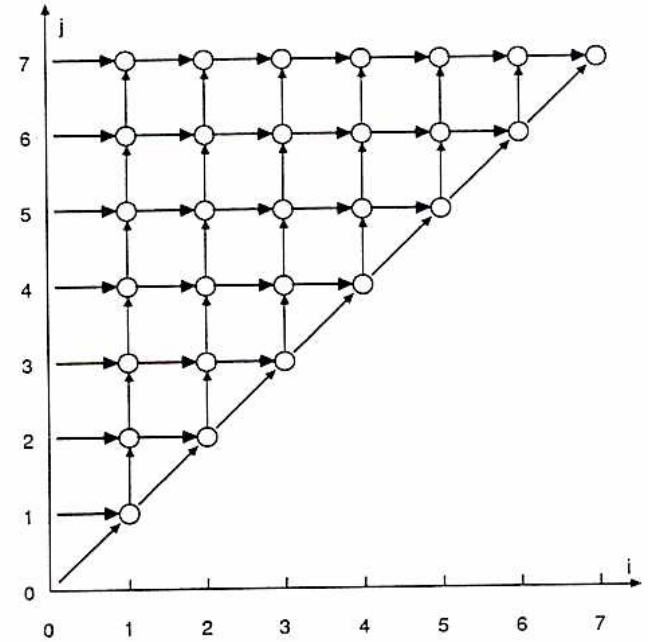
Figure 1. The process dependence graph of the mixed-radix conversion algorithm for $n = 7$.



Several time-optimal and spacetime-optimal systolic arrays for the mixed-radix conversion algorithm are given by Koç and Cappello [14]. The arrays and their corresponding schedules in [14] assume that m_i and $\mathbf{u}_i$ are being fed to the array from the south-end. For our problem, however, this is not the case; m_i and $\mathbf{u}_i$ for $0 \leq i \leq n$ are already available in ith processor after Step 1 of the congruence algorithm. In the following, we introduce two new systolic arrays which use the data available in the processors.

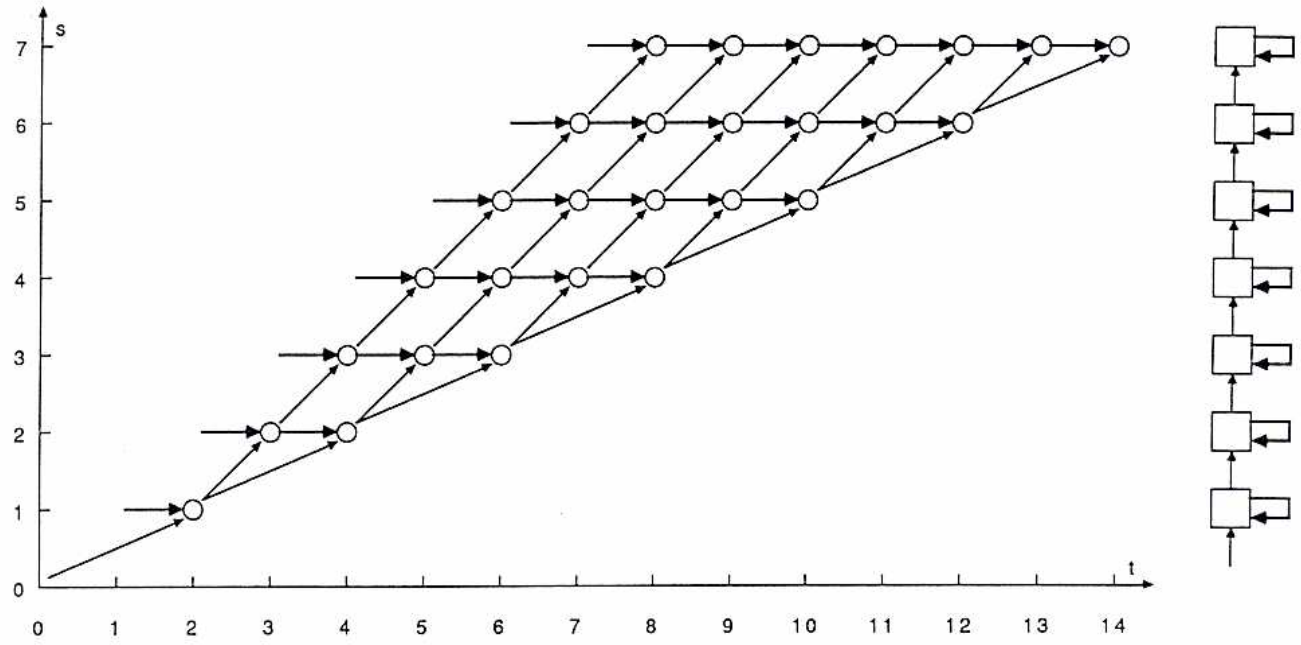
As a first step, we modify the data dependence graph in Figure 1 in order to achieve local dependence, i.e., a node is dependent only on its neighboring nodes. Discussions and several examples on transformation of data dependence graphs to achieve local dependence can be found in [15]. The resulting dag representing the operations performed by the mixed-radix conversion algorithm is given in Figure 2. We then embed the localized process dependence graph of the mixed-radix conversion algorithm in spacetime to produce a time-optimal and a spacetime-optimal systolic array. The reader is referred to [15] (and the references therein) for spacetime embedding techniques.

Figure 2. The graph in Figure 1 with localized dependence.



A Time-Optimal Systolic Schedule: We embed the process dependence graph for the mixed-radix conversion algorithm in spacetime. The abscissa is interpreted as time t; the ordinate as space s. The linear embedding E_1 is as follows:

Figure 3. A time-optimal systolic schedule for the mixed-radix conversion algorithm.



$$t := i + j \ ;$$
$$s := j \ .$$

The result, depicted in Figure 3 for $n = 7$, is a time-optimal array. Data that flows west $\rightarrow$ east in Figure 2 flows in the direction of time (perpendicular to space) in this design: it is in the processors' memory. Data that flows south $\rightarrow$ north in Figure 2 flows up through the array. Data that flows south $\rightarrow$ east in Figure 2 also flows up through the array, but at half the speed of the south $\rightarrow$ north data.

Process (i, j) is executed at time $i + j$ in processor j. By inspection, we see that the array uses n processors, finishing the computation in $2n$ steps. The number of vertices (processes) in a longest directed path in any process dependence graph is a lower bound on the number of steps of any schedule for computing the processes. In our graph, the number of vertices in a longest path is $2n$. Thus, this array uses a spacetime embedding that is optimal with respect to the number of steps used. Such an embedding is referred to as *time-optimal.*

A Spacetime-Optimal Systolic Schedule: We now give another embedding that achieves spacetime-optimality, i.e., it is space-minimal among all designs that are time-optimal. We nonlinearly embed the process dependence graph as follows:

$$t := i + j \ ;$$
$$s := j \bmod \left\lfloor \frac{n}{2} \right\rfloor \ .$$

This embedding E_2 is illustrated in Figure 4 for $n = 7$. Its data flow characteristics are identical to those of embedding E_1, except that the upper processor is attached to the lower processor forming a ring topology, and data movement wraps around the array.

This embedding results in a computation of the process dependence graph that uses $2n$ steps and a ring array of $\lceil \frac{n+1}{2} \rceil$ processors.

The embedding E_2 is space-minimal in addition to being time-optimal. In order to see this, consider time step 8 in which all 4 processors are used. To reduce the number of processors, it is necessary that the process depicted by node $(8, 3)$ be rescheduled onto another processor. However, node $(8, 3)$ is on a longest directed path in the process dependence graph. This means that it cannot be rescheduled for earlier completion without violating a dependence. Neither can it be scheduled for later completion without either violating a dependence, or extending the overall completion time, violating time-optimality. The number of processors therefore cannot be reduced: the design is spacetime-optimal.

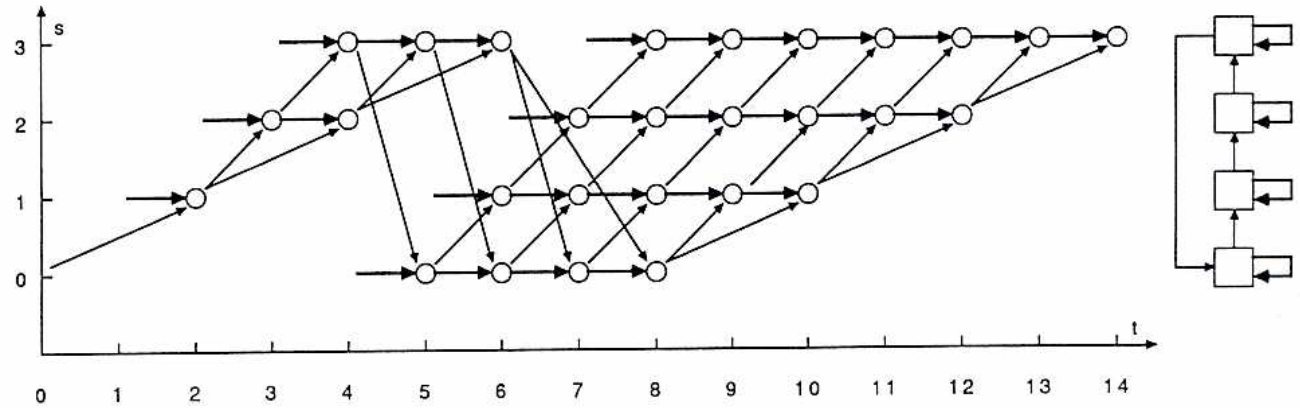
Any spacetime embedding of this process dependence graph that completes in $2n$ cycles, must use at least $\lceil \frac{n+1}{2} \rceil$ processors [14].

Thus, the optimal value of q is equal to 2, i.e., we partition the moduli set such that each processor executes Step 1 of the congruence algorithm for 2 moduli.

Theorem 3 *The parallel congruence algorithm requires $O(k^3 + nk)$ arithmetic and $2n(k+1)$ routing steps on a ring array with $p = \lceil \frac{n+1}{2} \rceil$ processors.*

Proof Assuming that $n+1$ is even, we allocate the moduli set and A and b such that processor i receives m_i and m_{i+p} for $i = 0, 1, 2, \ldots, p$ where $n + 1 = 2p$. Processor i computes $\mathbf{y}_i$ and $\mathbf{y}_{i+p}$ by solving the systems $\mathbf{A}\mathbf{y}_i = \mathbf{b} \pmod{m_i}$ and $\mathbf{A}\mathbf{y}_{i+p} = \mathbf{b} \pmod{m_{i+p}}$. Simultaneously, the determinants $d_i = \det(\mathbf{A}) \pmod{m_i}$ and $d_{i+p} = \det(\mathbf{A}) \pmod{m_{i+p}}$ are computed. Thus, the computation of $\mathbf{y}_i$ and d_i for all $0 \le i \le n + 1$ will take $O(k^3)$ arithmetic steps since all $p = \lceil \frac{n+1}{2} \rceil$ processors work simultaneously.

Figure 4. A spacetime-optimal systolic schedule for the mixed-radix conversion algorithm.



Similarly, the computation of $\mathbf{z}_i = d_i \mathbf{y}_i \pmod{m_i}$ $\mathbf{z}_{i+p} = d_{i+p}\mathbf{y}_{i+p} \pmod{m_{i+p}}$ takes $O(k)$ arithmetic steps.

We then start the parallel mixed-radix conversion algorithm which requires $2n$ steps, where at each step the operations given by (3) and (4) are performed, and a vector of dimension $k+1$ may be sent. Since vectors V are of dimension $k+1$, we perform $O(nk)$ arithmetic operations altogether. Furthermore, at most $2n(k+1)$ routing steps are needed by the parallel mixed-radix conversion algorithm.

After the execution of the mixed-radix conversion algorithm, processor $p-1$ has vectors $\mathbf{V}_{i,i+1} = \mathbf{v}_i$ for $0 \le i \le n$. We compute $\mathbf{u}$ (i.e., $\mathbf{z}$ and d) in processor $p-1$ using floating-point arithmetic with Horner's algorithm as

$$\mathbf{u} = \mathbf{v}_0 + \mathbf{v}_1 m_0 + \mathbf{v}_2 m_0 m_1 + \cdots + \mathbf{v}_n m_0 m_1 \cdots m_{n-1} .$$

This step is completely sequential and requires $O(nk)$ arithmetic operations.

Therefore, the parallel congruence algorithm requires a total of $O(k^3 + nk)$ arithmetic and $2n(k+1)$ routing steps on a ring array of $\lceil \frac{n+1}{2} \rceil$ processors. □

5 Implementation

The parallel congruence algorithm described in this paper is suitable for implementation on distributed-memory multiprocessors and systolic computing systems. The processors of the parallel computer system must be powerful enough to solve a system of linear equations. Examples of commercially available distributed-memory multiprocessors are Intel iPSC series, NCUBE, and Ametek. Examples of software-oriented systolic/wavefront computing systems include an array of Transputers [12], the Warp [3], and the Matrix-1 [8].

We have implemented the parallel congruence algorithm on a first generation Intel hypercube with $p = 8$ processors (iPSC/d3). The ring topology required for the spacetime-optimal systolic implementation of the mixed-radix conversion algorithm is easily embedded in the hypercube by using the binary-reflected Gray code [6].

The timing results for the time-optimal and spacetime-optimal schedules are summarized in Figure 5. The values of $n+1$ and k were limited to those given due to memory limitations (512 KBytes per node on this particular Intel cube).

The time for the sequential version of the algorithm is measured by executing the sequential congruence algorithm on one of the eight identical processors. In order to obtain a fair comparison between the sequential and the parallel algorithm, we measure the total execution time, but not the initial loading of the data and the final unloading of the results. The efficiency of the parallel congruence algorithm is plotted in Figures 6.

Theoretically, the parallel congruence algorithm should achieve linear speedup (constant efficiency). However, we observe a decline in the efficiency as n increases. This is due to the fact that $2n(k+1)$ routing operations required by the parallel algorithm start taking a significant amount of time for large n. Furthermore, the slow communication (*store-and-forward*) mechanism of the first generation hypercubes constitutes an obstacle to high efficiency. We note that the newer message-passing multiprocessors have more advanced and thus much faster data communication mechanisms [4]. The efficiency becomes higher for larger k, since Step 1 (which requires no communication and $O(k^3)$ parallel arithmetic steps) starts dominating Step 2 (which requires $2n(k+1)$ routing and $O(nk)$ parallel arithmetic steps).

6 Conclusions

Several extensions of the parallel congruence algorithm can be proposed. An important issue arises when the number of processors in the parallel computing system does not match the size of the problem to be solved. There are two cases:

Fewer processors ($2p < n+1$): Let $n+1 = qp$ where $q > 2$. We partition the moduli set into $2p$ groups where each group contains $\frac{q}{2}$ moduli. The allocation of the data is similar to the case $q = 2$, however, now processor i is assigned to perform computations using the moduli in groups i and $i+p$ for $i = 0, 1, \ldots, p-1$.

More processors ($2p > n+1$): In this case, the best approach is to exploit parallelism inherent in the Gaussian elimination step, and also in vector operations required by the congruence algorithm. As an example, assume that we have p processors where $2p = k(n+1)$. We can allocate k processors for each of the linear systems of equations solved in Step 1 of the congruence algorithm. We can use these k processors to reduce the number of arithmetic operations required by the Gaussion elimination algorithm from $O(k^3)$ to $O(k^2)$. Furthermore, the parallel mixed-radix conversion algorithm will now require only $O(n)$ arithmetic operations instead of $O(nk)$. Parallel algorithms for Gaussian elimination and LU decomposition are well-known [6]. However, the parallel congruence algorithm requires Gaussian elimination over the ring of integers $\mathcal{Z}_{m_i}$ (or the Galois field $GF(m_i)$ when m_i is prime). Thus, pivoting or partial pivoting is necessary, since every m_ith element is zero. Systolic algorithms for Gaussian elimination without pivoting [9, 1] and with partial pivoting [5, 11] have been proposed. Thus, we see that, depending on the number of processors available in the parallel computer system, different levels of parallelism already inherent in the congruence algorithm can be exploited.

When the number of processors available is more than $\lceil \frac{n+1}{2} \rceil$, we can allocate more than one processor for each congruent linear system solved, and thus reduce the amount of memory required for each node. This is due to that fact each node receives fewer than k rows, allowing a larger system to be solved for a given memory space.

Before concluding, we note that the hypercube network is richer in connectivity than a linear (or ring) array. Thus, it seems worthwhile to investigate how the total number of routing operations required can be reduced by utilizing the additional connections. Other distributed-memory architecture topologies (e.g., two or three dimensional mesh, binary tree) can also be used for implementing the congruence algorithm.

References

[1] H. M. Ahmed, J. M. Delome, and M. Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Computer*, 15:65–82, 1982.

[2] A. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Co., 1974.

[3] A. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. Webb. The WARP computer: Architecture, implementation, and performance. *IEEE Transactions on Computers*, 36(12):1523–1538, December 1987.

[4] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–25, August 1988.

[5] H. Barada and A. El-Amawy. Systolic architecture for matrix triangularisation with partial pivoting. *IEE Proceedings, Part E: Computers and Digital Techniques*, 135(4):208–213, July 1988.

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

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

[8] D. E. Foulser and R. Schreiber. The Saxpy Matrix-1: A general-purpose systolic computer. *IEEE Computer*, 20(7):35–43, July 1987.

[9] W. M. Gentleman and H. T.Kung. Matrix triangularisation by systolic arrays. In *Proc. SPIE 298, Real-Time Signal Processing IV*, pages 19–26, San Diego, CA, 1981.

[10] R. T. Gregory and E. V. Krishnamurthy. *Methods and Applications of Error-Free Computation.* Springer-Verlag, 1984.

[11] B. Hochet, P. Quinton, and Y. Robert. Systolic Gaussian elimination over $GF(p)$ with partial pivoting. *IEEE Transactions on Computers*, 38(9):1321–1324, September 1989.

[12] INMOS Ltd., Almondsbury, Bristol, UK. *IMS T800 Transputer*, November 1986.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley Publishing Co., second edition, 1981.

[14] Ç. K. Koç and P. R. Cappello. Systolic arrays for integer Chinese remaindering. In M. D. Ercegovac and E. Swartzlander, editors, *Proceedings of the 9th Symposium on Computer Arithmetic*, pages 216–223, Santa Monica, California, September 6–8 1989. IEEE Computer Society Press.

[15] S. Y. Kung. *VLSI Array Processors.* Prentice-Hall, 1988.

[16] J. D. Lipson. Chinese remainder and interpolation algorithms. In *Proc. 2nd Symp. Symbolic and Algebraic Manipulation*, pages 372–391, 1971.

[17] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley Publishing Co., 1981.

[18] G. Mackiw. *Applications of Abstract Algebra.* John Wiley and Sons, Inc., 1985.

[19] M. Newman. Solving equations exactly. *Journal of Research of the National Bureau of Standards*, 71B(4):171–179, October–December 1967.

[20] G. Villard. Exact parallel solution of linear systems. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 197–205. Academic Press, 1989.

[21] D. M. Young and R. T. Gregory. *A Survey of Numerical Mathematics*, volume 2. Dover Publications, Inc., 1988.

Figure 5. Timing results (in milliseconds) for the sequential (T_{seq}), the time-optimal (T_{to}), and the spacetime-optimal (T_{sto}) schedules.

	$k = 10$			$k = 20$			$k = 30$		
$n+1$	T_{seq}	T_{to}	T_{sto}	T_{seq}	T_{to}	T_{sto}	T_{seq}	T_{to}	T_{sto}
16	3205	615	600	16475	2450	2450	48440	6620	6550
32	8235	1700	1720	36450	5810	5700	102040	14560	14310
48	15135	3290	3365	59995	10100	9870	160920	23925	23315
64	23965	5450	5400	87210	15495	14870	225215	34890	33580
80	34765	8150	7800	118195	21920	20710	295075	47370	45225
96	47555	11390	10990	152980	29370	27675	370540	61370	58180
112	63050	15220	14875	192930	37955	35485	453635	77040	72275

Figure 6. Efficiency of the parallel congruence algorithm.