JOURNAL OF THE
CHUNGCHEONG MATHEMATICAL SOCIETY
Volume 6, June 1993

A Parallel Algorithm for Large-Scale Linear Programs with a Special Structure

SEYOUNG OH

ABSTRACT. A new sequential algorithm and computational results for large-scale linear programs with a special structure were presented in the previous paper [9]. In this paper, a parallel version of the algorithm was developed for a hypercube multiprocessor architecture NCUBE2. Computational results using 128 processors are presented for a randomly generated large-scale sparse or dense problems with the number of variables up to 256 and constraints up to 5 million.

1. Introduction.

A problem considered here is an important type of large-scale linear program with a special structure is that with relatively few variables but many inequality constraints. A structure of particular interest occurs when the constraint matrix consists of K block matrices, each $m \times m$, where K may be 1000, or larger. Furthermore, each $m \times m$ block is diagonally dominant with positive entries on the diagonal.

The problem can be stated as follows:

$$(1) \qquad \begin{array}{ll} \max & b^T x \\ \text{subject to} & (C^j)^T x \le d^j , \quad j = 1, \ldots, K, \end{array}$$

where $b \ge 0$, x and the d^j are vectors in R^m , and each $(C^j)^T$ is an $m \times m$ matrix which is strictly diagonally dominant, with positive diagonal elements and nonpositive off-diagonal elements.

Received by the editors on June 30, 1993.
1980 *Mathematics subject classifications*: Primary 49M35.

A linear program with this structure represents production problems or the infinite horizon, discounted Markov decision problem, as described in [1], [3] and [10]. For these problems, the number of constraints grows as the product of the state space size and the control space size, and therefore may become very large.

An efficient sequential algorithm and computational results on CRAY-2 were shown in the previous paper [9]. In the paper, the problem (1) was considered as the dual problem and the equivalent primal with only m rows should be solved. The new method presented in the paper can also be thought of as a simplex type algorithm applied to the equivalent primal. The primary difference is that instead of only one pivot, multiple pivots may be carried out at each iteration. This caused typically a dramatic reduction in the total number of iterations required, as compared to the Simplex Method.

In this paper, a parallel version of the new algorithm was developed for a hypercube multiprocessor architecture NCUBE2. Computational results using 128 processors are presented for randomly generated large-scale sparse or dense problems.

In Section 2, the necessary notation is introduced and the parallel algorithm described.

In Section 3, an efficient way to solve linear systems on hypercube architectures. A number of large-scale sparse or dense problems generated for the test of the efficiency of the algorithm are described in Section 4.

In Section 5, computatinal results using 128 processors of NCUBE2 are presented for a number of randomly generated large-scale sparse or dense problems with number of variables up to 256 and constraints up to 5 million. The time to solve the largest problem ($m = 128$, $K = 19200$) with 128 processors was 2.86 seconds and the scaled speedup achieved was 107 whose efficiency is 84%.

2. Reformulation and Parallel Algorithm Description.

2.1 Reformulation.

To simplify the presentation, each row of the system of inequalities (1) is rescaled so that the diagonal elements of $(C^j)^T$ are all equal to unity. The rescaled system (1) becomes

$$(2) \qquad (\underline{C}^j)^T x \leq \underline{d}^j \quad , \quad j = 1 ,\dots, K \, ,$$

where each row has been divided by the original diagonal element. As a result, each column of $\underline{C}^j$ has a maximum element equal to unity, with the sum of the absolute values of all other elements less than unity.

The mK columns of the matrices $\underline{C}^j$, $j = 1 ,.., K$, are now regrouped into m matrices with K columns each. Specifically, let $A^i \in R^{m \times K}$, denote the matrix with the i^{th} column from each of the matrices $\underline{C}^1$, $\underline{C}^2$,..., $\underline{C}^K$. The elements of $\underline{d}^j$ are regrouped in a compatible way to give vectors $c^i \in R^K$, $i = 1 ,\dots, m$. The system (1) or (2) can now be written as :

$$(3) \qquad (A^i)^T x \leq c^i \, , \; i = 1 ,\dots, m \, .$$

The columns of A^i will be represented by a^i_j , $j = 1 ,.., K$, with the i^{th} element of each vector a^i_j equal to unity. Now let

$$A = [\, A^1 \;\, A^2 \, \dots . \, A^m \,] \in R^{m \times mK}$$

$$c^T = [\, (c^1)^T \;\, (c^2)^T \, \dots . \, (c^m)^T \,] \in R^{mK} \; .$$

Then the original problem (1) can be written as

$$(4) \qquad \max_x \; \{ \; b^T x \mid A^T x \leq c \; \} \, .$$

This original problem will be considered as a dual problem with m variables and mK linear inequality constraints. The equivalent primal is :

$$(5) \qquad \min_{y} \{ \; c^T y \mid A\,y = b \, , \, y \geq 0 \; \} \quad , \quad y \in R^{mK} \, .$$

Let $B \in R^{m \times m}$ denote a feasible basis for (5).

2.2 Parallel Algorithm Description.

Since the algorithm allows multiple pivots at each iteration and each pivot occurs in each of the m blocks of the problem (4) or (5), a parallel implementation can be done efficiently on a hypercube MIMD computer. Distributed-memory multiprocessor and multicomputer architectures like NCUBE2 with 1024 processors, are an intuitive choice for the new algorithm for the large-scale LP problems (1). Because the processors are consistent with the blocks in the reformulated problem (4), whose constraint matrix is not necessarily sparse in structure, and the problem size grows as the product of m and K, it requires a big memory size for one processor. A parallel version of the new sequential algorithm in the previous paper [9] was developed for a hypercube multiprocessor architecture NCUBE2 in the Sandia National Laboratory.

NCUBE2 is one of MIMD high performance parallel computers with a hypercube architecture which has highly concurrent multiprocessors based on the binary n-cube topology which is well known for its powerful interconnection features and suitable for parallel speedup research. Every 1024 processor has a fast 32-bit and 64-bit floating point arithmetic to be competitive with conventional supercomputers on an absolute performance basis. The processors share information through the communication I/O channels. The cost parameters for running a program on NCUBE2 are mainly floating-point operations, message transfers for interprocessor communication, load imbalance and synchronization. The effects of these parameters can be reduced by users according to their computing environment. Efficient broadcast algorithm, message organization and careful ordering of reads and writes can yield considerable communication overlap and idle of processors, which reduces the time spent on interprocessor communication, see [6].

Assigning each node into one or more blocks of the problem (5), we compute θ_i using

$$\theta_i = \min_j \phi_j^i(\bar{x}) = \phi_{q_i}^i(\bar{x}) \ . \tag{6}$$

and check the dual feasibility with the current dual solution x which broadcasts to all nodes. If the dual feasibility is not satisfied, then pick the most violated constraints from each block which was assigned into a node. At step 3, by sending a pivot column from the root node to all of the other nodes, we update the basis B locally in all nodes. As in practical problems, when the number of blocks K is relatively larger than the number of variables m, the cost of communication is acceptable for the parallel algorithm because the amount of work done between synchronizations is relatively large. For the Markov decision problem which is a special case of the LP model (1), where all entries of constraint matrix C in the problem (1) are less than one, the amount of time for checking the dual feasibility at each node and the stored data in all of the nodes are well balanced. For the general practical problems, the main difficulty of the parallel implementation is a load imbalance in the case that m is not a multiple of the number of processors. By choosing the number of processors p so that a residual of the division m by p is close to m , the load imbalance can be resolved and the idle time of processors is reduced.

ALGORITHM 2.1. Host program

Hstep 1. Read the dimension of cube, the number of processors , the number of blocks K and the number of variables m and the density ρ of constraint matrix, and parameters for generating problems.

step 2. Open a hypercube and load the program to all nodes.

Hstep 3. Send the input values in Hstep 1 to node 0.

Hstep 4. Receive a message indicating a termination of iterations from node 0.

Hstep 5. Close the hypercube. □

ALGORITHM 2.2. Node program

Nstep 1. Start the timer.

Nstep 2. Get my node number (gray code) and the dimension of cube.

Nstep 3. At node 0, receive the input values (Hstep 1.) from the host. If my node is not 0, then receive the input values from node 0 by fan-out broadcast.

Nstep 4. Create the block number(s) between 0 and m which is(are) allocated by the order of gray code and block interleaved storage scheme. Generate the constraints of corresponding block(s) with the parameter and the density of constraint matrix.

Nstep 5. Record the time. Choose any column vector(s) from the assigned constraint block(s) for the first feasible basis B .

Nstep 6. For $i = m$,... , 2 , if my node has column i , broadcast the pivot row to all nodes using logarithmic fan-out algorithm. Update the column(s) from Nstep 5 by Gaussian elimination.

Nstep 7. For $i = 1$,... , m, if my node has the column i , compute x_i and send x_i to all nodes using logarithmic fan-out broadcast to backsolve for x .

Nstep 8. Test for dual feasibility. If not satisfied with the dual feasibility choose that column from A^i corresponding to the most violated constraint, that is $a^i_{q_i}$ and exchange it with column a^i_j in B at my node. If satisfied for all constraints at my node, then my node is complete.

Nstep 9. Send a message indicating my node complete (or not) to node 0 using fan-in broadcast. At node 0 collect the messages, check if all nodes are done or not, and send a message of continuation or termination to all nodes. If all of the nodes are not done, then go to Nstep 6.

Nstep 10. Stop the node timer. Send a message of termination from node 0 to the host. Gather all of the timing statistics at node 0 by logarithmic fan-in algorithm.

Nstep 11. Print out the output data at node 0. (this data includes solution of the problem, error of feasibility, number of iterations and execution time) □

To handle all synchronizations, fan-in and fan-out algorithms were implemented by using spanning tree of the hypercube. The algorithm broadcasts information from any single node to all of the other nodes in a given n-cube in $\log_2(p)$ steps, where p is the number of processors. At each iteration of Gaussian elimination, the node, which has to send its pivot row to all of the other nodes, must first send it to a neighbor which will have to send the pivot row for the next update. This can be obtained by rotating the order in the hypercube dimensions of the data paths taken at each of the $\log_2 p$ steps. This also ensures that the messages have arrived by the time the corresponding reads are executed at almost of all nodes, see Algorithm 2.3. To obtain the message of completion or continuation of iterations from all nodes, a fan-in algorithm was used in reverse of the fan-out algorithm.

If the number of the dual variables m is a multiple of the number of processors p, then the blocks will be assigned to the processors evenly and the load is balanced except for the load imbalance which is caused by data-dependent difference in arithmetic times. Since the constraints were stored by a block interleaved scheme (each block has $K \times m$ matrix), the performance will be reduced when the remainder of m divided by p is small.

ALGORITHM 2.3. For a given d-dimensional hypercube, a message is broadcasted from a source processor to all of the other processors by rotating the dimension $(0, \dots, d-1)$ according to gray code. The *ieor* function is exclusive OR. The *igc* function is the inverse of a gray code mapping *gc* .

```
next.gray = $igc(\text{source}) + 1$
$k = \log_2(|gc(\text{next.gray}) - \text{source}|)$
mynode.new = $ieor(\text{mynode, source})$

For  $i = 0, d-1$
    $j = k + i \bmod d$
    if $2^j \leq \text{mynode.new} < 2^{(j+1)}$
       nbr = mynode.new $- 2^j$
       send.node = $ieor(\text{nbr, source})$
       *receive a message from* send.node
    elseif mynode.new $< 2^j$
       nbr = mynode.new $+ 2^j$
       get.node = $ieor(\text{nbr, source})$
       *send a message to* get.node
    endif
Endfor
```

3. Solving linear systems.

We next consider an implementation of an algorithm for linear systems on the parallel computer NCUBE2. A number of papers have appeared in recent years describing various parallel LU factorization schemes on distributed memory architectures, for example see [2], [4], [5], and [7]. Assuming that $l = \lfloor \frac{m}{p} \rfloor$, in the previous section the i^{th} block of constraint matrix was assigned to the processor of j^{th} order of gray code, where $j = mod(i,p) + 1$ and $1 \leq i \leq K$. When the most violated constraint is picked from each block at each node (total m constraints are picked globally), columns $1, p+1, 2p+1, ..., lp+1$ are already in the first processor of gray code, columns $2, p+2, 2p+2, ..., lp+2$ are in the second processor, and so on. It is not necessary to move any columns to another processor to solve the linear system of the current step. One can take advantage of this storage scheme to solve a linear system itself in parallel which is explained at the end of this section. The algorithm adopted *Row Storage with Row or Column Pivoting*, but when the problem (1) is regrouped into the problem (2) in Section 2.1, all basis matrices become well pivoted with unity diagonal elements and the absolute values of the off-diagonal elements less than 1. Therefore, no pivot is required. Let $Bx = c_B$ be the linear system to solve, $B = (\beta_{ij})$ a feasible basis, and $c_B = (c_i)$ the corresponding primal cost coefficient vector. Each processor executes Algorithm 3.1 and Algorithm 3.2.

ALGORITHM 3.1. LU factorization with interleaved row storage scheme

```
do $k = 0, m-1$
  if *processor owns row k*
    *fan-out broadcast row k to all processors*
  else
```

```
            receive row $k$
        endif
        for all rows $i > K$ that processor owns
            $\lambda_{ik} = \beta_{ik}/\beta_{kk}$
            do $j = k+1, m-1$
                $\beta_{ij} = \beta_{ij} - \lambda_{ik}\beta_{kj}$
            end
    end
```

These schemes for solving a linear system in parallel have a drawback which decreases the number of active processors by one at each step. However, our storage scheme which is called *Pattern Wrapped Interleaved Storage* alleviates the problem of processors becoming idle and keeps all processors working almost until the end of the reduction. However, there still exists some imbalances in the workloads of processors. For example, at each step one row is no longer needed. These imbalances affect a different speed-up when we choose a different number of processors p for the same problem, which are shown in Table 2 - 4 later in this section.

ALGORITHM 3.2. Back substitution with a interleaved row storage scheme

```
    do $k = m, 1$
        if processor owns row $k$
            $x_k = c_k/\beta_{kk}$
            fan-out broadcast $x_k$ to all processors
        else
            receive $x_k$
        endif
        for all rows $i < k$ that processor owns
            $c_i = c_i - \beta_{ik}x_k$
```

end

4. Test Problems.

In real life problems, both the number of blocks K and the number of variables m may be very large. In addition, the problems have usually full dense constraint matrices. For the purpose of comparison with the computational results on several nodes and one node, the entire constraint matrix is stored in one processor when the algorithm uses only one node. However, due to the restricted local storage of nodes for solving a big problem on one node, if K and m are getting larger, the density of the problem should be getting smaller. This will occur even if we use a sparse structure storage scheme of data for a large-scale sparse problem. The test problems were generated randomly in a given hypercube with K ranging from 150 to 19200, m ranging from 20 to 256, and density ranging from 1 to 100 % which is as big as the memory allows. The constraints in the problem (4) are stored in the local memory of node(s) of NCUBE2 using a sparse row oriented scheme. Nonzero elements of constraint matrix A and the right hand side c are stored in a real array. These nonzero elements were randomly generated in each of the processors assigned by the host program with all of the blocks of matrix A using the block interleaved storage scheme as explained in the previous section. For each size, 15 problems are generated so that they are satisfied with the diagonal dominance and the other properties of the problem (1). In order to check the performance of the algorithm when the constraints have sharp corners, the difference between the diagonal element and the absolute sum of off-diagonal elements was randomly chosen from the interval $(0,1)$. The constraints which have a sharp corner are illustrated by less diagonal dominance property which causes to one drawback of the successive approximation methods, see [10].

5. Computational Results and Analysis.

The parallel algorithm in Section 2.2 was designed to solve a class of randomly generated problems described in Section 1 and to be implemented on a hypercube multiprocessor. It was implemented for a second generation hypercube NCUBE2 with 1024 processors maintained by the Sandia National Laboratory using the NCUBE F77 Fortran compiler.

15 problems for each size with K ranging from 150 to 19200, m ranging from 20 to 256, and density ranging from 1 to 100 % were tested on the number of nodes ranging from one to 128 processors of NCUBE2.

The more theoretical measure of speed-up is a comparison of the best possible algorithm on the fastest sequential machine versus the best possible parallel algorithm on a p-processor machine. For the class of LP problems like (1), the sequential code running on CRAY-2 is the fastest code. Table 1-4 shows the time to execute the sequential algorithm in the previous paper [9] on CRAY-2 and the time to execute the parallel algorithm on NCUBE2 with 64 nodes. The large and dense problem which can be run on a large number of processors does not fit into the memory of a single processor. Therefore, the larger the problem that can be run, the smaller the density that was used. This is shown in Table 1-4. This small density for the large problem degraded speedup much more than in the case of full density for the same size of problem. The reason is that a sparse row oriented storage scheme for the constraint matrix was used. If the density decreases as the size of problem increases, then the number of nonzero elements in the constraints will not be changed much. From the point of view of a number of operations, each iteration for the sequential algorithm requires $\frac{2m^3}{3} + 2\rho m(m-1)K$, where ρ is a density of the constraint matrix. If values of $\rho m(m-1)K$ for two different size of problems are almost the same, then the degree of parallelism will depend on the solver of linear systems. This does not take advantage of a major part of the parallel implementation for the class of the problem (1). So it is not reasonable to keep the number of sequential operations fixed to analyze performance, because the usual parallel overhead is known to have more effect on smaller problems than on larger problems. To handle this kind of problem, we compute a scaled speed-up by allowing the problem to be as large as the memory of the nodes we used. So we can define a scaled speed-up by comparing time to execute a problem of maximum size that can be solved on p-processor versus time to execute a problem of maximum size for 1-processor. Also we need an adjustment factor which is the ratio of the number of operations required to solve those problems. In other words, we can define the speed-up by measuring a computation rate as the size of problem and the number of processors increase, see [8]. The number of operations required by the algorithm at each iteration can be defined as a function of m and K,

$$f(m,K) \approx \frac{2m^3}{3} + 2\rho m(m-1)K.$$

If we choose a maximum problem size (m_1, K_1), suitable for a single processor, then the appropriate problem size for p processors (m_p, K_p) would satisfy $f(m_p, K_p) \approx \gamma p f(m_1, K_1)$, that is,

$$(7) \quad \frac{2{m_p}^3}{3} + 2\rho m_p(m_p-1)K_p \approx \gamma p \left(\frac{2{m_1}^3}{3} + 2\rho m_1(m_1-1)K_1 \right),$$

where γ is a ratio of the number of iterations for the problem of size (m_1, K_1) to that for the problem of size (m_p, K_p). From the computational results for many test problems in Section 4, we notice that γ is very close to 1. For an example, assuming we have a dense problem with a maximum size (20, 100) for one processor, we try to find a problem size that can be solved using 8 processors. In order to achieve a perfect speed-up, the problem size increases so that 8 times of the number of operations can be done in the same amount of time that is taken to solve a problem of size (20, 100). From the equation (7), when $m_1 = m_8 = 20$, $K_1 = 100$ and $\gamma = \frac{5}{6}$ from Table 0, K_8 will be 728. Thus from the execution time for both problems in Table 0 the scaled speedup will be

$$\frac{T_1(20,100)\times 8}{T_8(20,728)} = \frac{1.363\times 8}{1.783} = 6.12,$$

where $T_p(m_p, K_p)$ is the execution time for the problem size (m_p, K_p) using p processors.

Table 0: Execution time (sec) on NCUBE2 for simple example problems

K Blocks	m Variables	Iterations	1 node	8 nodes
100	20	5	1.363	-
728	20	6	11.090	1.783

Table 2 shows the effects of number of blocks while number of variables is fixed. In Table 3, the effects of number of dual variables m while number of blocks K is fixed are given. In Table 4, test problems were developed to experimentally characterize the effects of densities of the constraint matrices when the problem size was fixed. We notice in this table that if the constraint matrix is fully dense, then efficiency increases up to 99% by using an appropriate number of processors. Table 4 shows that we can obtain a high degree of parallelism for this algorithm when most of the practical problems are fully dense.

Table 1: Comparison of new parallel algorithm on NCUBE2 and CRAY-2.

Problem Size					Time (sec)	
No. of Blocks (K)	Dual Var.s (m)	K/m	Density	Iter-ations	NCUBE2 64 nodes	CRAY-2
412	252	1.6	0.4	7	20.74	8.27
443	76	5.8	1.0	7	4.35	2.28
1703	124	13.7	0.5	10	17.40	18.54
3409	124	27.5	0.2	8	12.36	23.38
4035	68	59.3	0.3	8	9.88	14.60

Table 2: Dependence of time and scaled speed-up on number of variables for fixed number of blocks on NCUBE2.

Problem Size				
No. of Blocks (K)	Dual Var.s (m)	Processors	Time(sec) per iteration	Scaled Speed-up
150	128	1	4.958	1
300	128	2	5.052	1.45
600	128	4	3.938	3.07
1200	128	8	3.351	4.96
2400	128	16	3.073	13.20
4800	128	32	2.938	26.71
9600	128	64	2.881	53.61
19200	128	128	2.862	107.07

Table 3: Dependence of time and scaled speed-up on number of blocks for fixed number of variables on NCUBE2.

Problem Size		Processors	Time(sec)	Scaled
No. of Blocks (m)	Dual Var.s (m)		per iteration	Speed-up
1500	20	1	2.769	1
1500	28	2	2.617	2.11
1500	40	4	2.592	4.30
1500	56	8	2.594	8.61
1500	80	16	2.677	16.82
1500	128	32	2.942	30.56
1500	192	64	3.496	53.53
1500	256	128	4.082	92.86

Table 4: Dependence of time and scaled speed-up on density for fixed size of problem on NCUBE2.

Problems are of $K = 1500$, $m = 128$.

Density	Processors	Time(sec) per iteration	Scaled Speed-up
0.01	1	4.106	1.0
0.03	2	3.385	1.72
0.05	4	3.417	3.31
0.12	8	3.378	6.88
0.27	16	3.207	14.77
0.55	32	3.063	31.51
1.00	64	2.725	63.61
1.00	128	1.589	109.11

References

1. Bertsekas, D.P., *Dynamic Programming : Deterministic and Stochastic Models*, Prentice-Hall, Englewood Cliffs, NJ., 1987.
2. Chu, E. and George, A., *Gaussian elimination with partial pivoting and load balancing on a multiprocessor*, Parallel Comput. 5 (1987), 65–74.

3. Deonardo, E.V., *Dynamic Programming, Models and Applications.*, Prentice-Hall, Englewood Cliffs, NJ., 1982.
4. Geist, A. and Heath, M., *Matrix factorization on a hypercube*, Hypercube Multiprocessors 1986, M.T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, 1986, pp. 161–180.
5. Geist, A. and Romine, C., *LU factorization algorithms on distributed memory multiprocessor architectures.*, SIAM J. Sci. Statist. Comput. **9** (1989), 639–649.
6. Gustafson, John L., et al, *Development of parallel methods for a 1024-processor hypercube*, SIAM J. Sci. Stat. Comput. **9** (1988), 609–638.
7. Ispen, I., Saad, Y., and Schultz, M., *Complexity of dense linear system solution on a multiprocessor ring*, Linear Algebra Appl. **77** (1986), 205–239.
8. Ortega, J., *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum Press, New York and London, 1988.
9. Rosen, J.B. and S. Oh, *An efficient algorithm for large-scale linear programs with a special structure*, Advances In Optimization And Parallel Computing 1992, P.M. Pardalos, ed., NORTH-HOLAND, The Netherlands, 1992, pp. 247–266.
10. Tseng, P., *Distributed computation for linear programming problems satisfying a certain diagonal dominance condition*, Mathematics of Operation Research **15** (1990), 33–48.

DEPARTMENT OF MATHEMATICS
CHUNGNAM NATIONAL UNIVERSITY
TAEJON, KOREA