Performance and Scalability of the Parallel Simplex Method for Dense Linear Programming Problems An Extended Abstract

George Karypis and Vipin Kumar
Computer Science Department
University of Minnesota

May 21, 1994

1 Introduction

Linear programming is of fundamental importance in many optimization problems. The simplex method [4] is a commonly used way of solving linear programming problems. Solving large instances of linear programming problems takes significant time; thus, solving linear programming problems in parallel offers a viable path for speeding up this type of computation.

A standard-form linear programming problem is specified as follows:

Given an $m \times n$ matrix A, a $m \times 1$ vector b, and an $1 \times n$ vector c, the goal is to

$$\begin{array}{rl} \text{minimize} & z = cx \\ \text{subject to} & Ax = b \text{ and} \\ & x \geq 0 \end{array}$$

The vector c is called the cost vector, while z is called the objective function.

In this paper, we are mainly concerned with linear programs whose constraint matrix is dense. Such problems are commonly solved using the tablue method [4]. Many parallel simplex formulations for dense linear programming problems have been developed [16, 17, 10, 5]. In this paper we provide a comprehensive performance and scalability analysis of many parallel formulations of the simplex method for a variety of architectures. Some of the parallel formulations of the simplex method we analyze were previously proposed [16, 10, 5] while the remaining are new schemes. We present analysis for a variety of architectures including hypercube, mesh, and network of workstations. Our analysis shows that a network of workstations has better scalability than the other architectures for some parallel simplex formulations. The fact that a network of workstations connected via standard Ethernet is a reasonably good architecture for the simplex method is of great significance, due to its wide availability.

Most of the linear programming problems are sparse, and sequential codes for them exploit sparsity to speedup computation. We have recently developed a highly parallel formulation for sparse linear programming problems using interior point methods [11]. Nevertheless, dense linear programming problems have some genuine applications. For instance the Dantzig-Wolfe decomposition [4] or Benders decomposition [1] generate highly dense master problems. More applications leading to dense linear programming problems are discussed in [5]. Furthermore, in problems in which the nonzero elements are a fixed percentage d of the total number of elements, the formulations presented here lead to scalable algorithms.

2 Parallel Simplex Formulations

Parallel formulations of the simplex method vary in the way the tablue T is distributed among processors. We assume that m is of the same order as n, but in absolute terms, m is smaller than n. This is true for most linear programming problems. A similar analysis can be done if m is much smaller than n. We consider both one- and two-dimensional partitionings of the tablue. In the one-dimensional partitioning, the tablue is distributed among p processors along the x dimension in such way, so that each processor gets $n/p+1$ columns of the tablue. The first n/p columns are part of the A matrix, while the last columns is the vector b. The columns can be assigned to each processor in either a *blocked fashion*, where each processor gets n/p consecutive columns of T, or in a *cyclic fashion* where each processors gets every pth column of T.[1]

In the two-dimensional partitioning, the p processors are arranged in a logical grid of $p_y \times p_x$ processors (*i.e.* $p = p_y p_x$). Each processor $P_{i,j}$ is assigned an $\frac{m+1}{p_y} \times \frac{n}{p_x}$ portion of the tablue. Again, the vector b is replicated in each column of the processor grid. Since in most linear programming problems, n is larger than m, the processor grid should be such that $p_x > p_y$. As it was the case with one-dimensional partitioning, the rows and columns of the tablue can be distributed to the processor grid in either blocked or cyclic fashion.

In both the one-dimensional partitioning along the x dimension and the two-dimensional partitioning cyclic distribution performed considerably better that blocked. This is because cyclic partitioning does a better job in load balancing the computation for the following reason. In each iteration of the simplex method, only $n-m$ columns of the tablue need to participate in the Gauss-Jordan step. Therefore, depending on how the columns of the basic matrix is distributed among the processors, this might lead to load imbalances. Initially, the m columns of the basic matrix are stored in the first m columns of the tablue. If the tablue is distributed using block partitioning, these m columns are assigned to a small number of processors. In subsequent iterations of the simplex method, these processors contain a large number of columns from the basic matrix, therefore perform significantly less computation during the pivoting. As the simplex method progresses, the load balance will improve because columns that were originally in the basic matrix will leave, while other columns will enter. As a result, the computation performed during pivoting will increase at the first processors.

If cyclic partitioning is used, the m basic columns will be initially distributed evenly to all the p processors. In fact, each processor will get m/p such columns. Therefore, when the iterations of the simplex method begin, the computation will be perfectly balanced. Each processor performs on the average $\Theta(m(n/p - m/p))$ computations. Our experiments have shown that if Dantzig's rule is used for selecting entering variables, the computation remains balanced as the algorithm progresses.

[1] An alternative one-dimensional partitioning is obtained by partitioning the tablue along its y dimension. In that case each processor gets m/p rows of T. However, since in most problems $m < n$, this partitioning scheme exploits less parallelism than the partitioning along the x dimension.

Parallel Formulation	Architecture		
	Hypercube	Mesh	Network of Workstations
Blocked 1D	[16, 17, 5]	New scheme	New scheme
Cyclic 1D	[16, 17]	New scheme	New scheme
Blocked 2D	[10, 5]	New scheme	New scheme
Cyclic 2D	New Scheme	New scheme	New scheme

Table 1: Summary of parallel simplex formulations.

3 Previous Work on Parallel Simplex

A variety of parallel formulations of the simplex method for dense problems have been proposed. Onaga and Nagayasu [15] proposed a VLSI wavefront array for the simplex method, and Bertossi and Bonuccelli [2] proposed a VLSI mesh of trees implementation. Stunkel and Reed [16, 17] proposed an one-dimensional partitioning of the tablue for a hypercube-connected computer. They investigated both column-wise and row-wise partitions, and experimentally evaluated their partitioning schemes on a 16-processor iPSC/2. Ho, Chen, Lin, and Sheu [10] proposed a two-dimensional partitioning of the tablue for a hypercube-connected computer and analyze its performance. Recently, Eckstein, Polymenakos, Qi, Ragulin, and Zenios [5] presented data-parallel implementations of dense linear programming algorithms. Table 1 provides a summary of the various parallel simplex formulations proposed so far. However, most of the work done so far is based on experimental evaluation and the scalability of the various formulations have not been fully investigated.

4 Scalability Analysis

We analyzed the scalability of all parallel formulations of the simplex method presented in Section 2 using the isoefficiency function [13, 6]. The *isoefficiency function* of a parallel algorithm determines the rate at which the problem size should increase with respect to the number of processors to maintain certain efficiency. For example, if W is the sequential complexity of an algorithm, then an isoefficiency function of $f(p)$ indicates that in order to maintain fixed efficiency E, as the number of processors increase, W should increase at the rate of $f(p)$. Isoefficiency analysis [7, 14, 9, 8, 12] has been successfully used in the past to analyze the scalability of a wide range of parallel algorithms.

Since the simplex method is an iterative algorithm, we concentrate on the scalability of the iterations of the simplex method. The scalability of the entire simplex method can be easily determined if we know the number of iterations. The basic communication operations in each iteration of the simplex method are (a) finding a minimum of $n-m$ elements of a row of the tablue; (b) performing a one-to-all broadcast of a column of the tablue. In one-dimensional partitioning, the $n-m$ elements are equally distributed among all processors and the column to be broadcasted is stored at one processor. In the two-dimensional partitioning, the $n-m$ elements of the row and the m elements of the column are distributed among p_x and p_y processors, respectively. The isoefficiency functions for a hypercube, mesh, and a network of workstations are shown in Table 2. For network of workstations we considered two architectures: (a) the standard Ethernet specified by the IEEE 802.3, and (b) the token ring specified by IEEE 802.5.

For one-dimensional partitioning, our scalability analysis shows that a network of workstations connected via Ethernet has better scalability than a hypercube- or a mesh-connected computer. The reason is that the Ethernet supports one-to-all broadcast of M elements to p processors in $\Theta(M)$

				Network of Workstations	
Partitioning	Hypercube	Mesh-SF	Mesh-CT	Ethernet	Token Ring
1D	$p^2 \log^2 p$	p^3	$p^2 \log^2 p$	p^2	p^4
2D	$p_x^2 \log^2 p_x$	p_x^4	p_x^4	p^2	p^4

Table 2: Isoefficiency functions for the one- and two-dimensional partitionings of the simplex method. The isoefficiency functions of the one-dimensional partitioning along the x dimension is the same as that along the y dimension. CT stands for cut-through routing while SF stands for store-and-forward routing.

time. In contrast, the same operation takes $\Theta(M \log p)$ on a hypercube [13]. The scalability is the same for mesh with cut-through routing (mesh-CT) and hypercube although it is worse for mesh with store-and-forward routing (mesh-SF). The reason is that one-to-all broadcast of M elements takes $\Theta(M \log p + \sqrt{p})$ time on a mesh-CT [13] and $\Theta(M\sqrt{p})$ on a mesh-SF. If M is sufficiently larger than $\sqrt{p}$ (which is the case if $m = \Omega(\sqrt{n})$), then one-to-all broadcast takes $\Theta(M \log p)$ time on a mesh-CT, which is of the same order as the time taken by this operation on a hypercube. Note that all current general-purpose parallel computers have CT routing (*e.g.*, Paragon, CM-5, nCUBE 2).

For two-dimensional partitioning a hypercube-connected computer is more scalable than a network of workstations. The reason is that a hypercube is able to perform many one-to-all broadcasts concurrently, whereas on a network of workstations, only one broadcast operation can take place at any time. For two-dimensional partitioning, mesh-CT does not have the same performance as a hypercube because the size of the messages used in one-to-all broadcast is too small.

5 Experimental Results

We performed experiments on the nCUBE 2 hypercube-connected computer and a network of SUN workstations. We used P4 [3] to implement the one-dimensional simplex formulations. Some of our preliminary experimental results are shown in Table 3. From this table we see that on nCUBE 2, the cyclic one-dimensional partitioning yields good speedups, and the speedup increases linearly as the number of processors increase particularly for large problem instances. On the network of workstations, for small problems, the speedup increases slowly as the number of processors increase. However, for sufficiently large problems, the speedup increases linearly as more processors are used. These performance results are preliminary and do not precisely follow the pattern predicted by the analysis. The reason is that the one-to-all broadcast in P4 is not optimized for network of workstations.

To study the relative performance of blocked and cyclic partitionings, we experimented with the blocked one-dimensional partitioning as well. As discussed in Section 2, block distribution of columns leads to load imbalances. In our experiments, for the 128×512 problem, the difference in the work performed by processors was 155% on eight processors. In contrast, for the cyclic column distribution, the work difference was only 6%. As a result of the high load imbalance for the blocked distribution, the speedup is only 5.9, whereas the cyclic partitioning yielded a speedup of 7.4. Similar load imbalances were observed for the other problems.

We are currently in the processes of porting the P4 code to use the explicit communication libraries provided for nCUBE 2. Furthermore, we are also planning to perform experiments on a

Problem Size	Number of nCUBE 2 Processors						Number of SUNs		
$m \times n$	2	4	8	16	32	64	2	4	8
64×256	1.92	3.64	6.64	12	21.76	38.4	1.4	1.96	2.8
64×512	1.98	3.84	7.2	13.6	25.28	45.44	1.7	2.32	3.36
128×512		3.96	7.44	14.08	26.88	48.64			
128×1024			7.6	14.72	28.16	51.84	1.88	3.52	5.2
256×1024			0.99	15.68			1.94	3.68	6.4

Table 3: Experimental speedups obtained on nCUBE 2 and a network of SUN workstations, using the cyclic one-dimensional partitioning along the x dimension.

network of 20 SUNs, an 256-node nCUBE, and a 256 node CM-5. These results will be reported in the full paper.

References

[1] J. F. Benders. Partitioning procedures for solving mixed variable programming problems. *Numerische Mathematik*, (4):238–252, 1962.

[2] A. A. Bertossi and M. A. Bonuccelli. A VLSI implementation of the simplex algorithm. *IEEE Transactions on Computers*, C-36(2):241–247, February 1987.

[3] R. Butler and E. Lusk. User's guide to the P4 parallel programming system. Technical Report ANL–92/17, Argonne National Laboratory, Argonne, IL, 1992.

[4] G. B. Dantzig. *Linear Programming and Extensions.* Princeton Iniversity Press, Princeton, NJ, 1963.

[5] J. Eckstein, L. C. Polymenakos, R. Qi, V. I. Ragulin, and S. A. Zenios. Data-parallel implementations of dense linear programming algorithms. Technical Report TMC-230, Thinking Machines Corporations, Cambridge, MA, 1992.

[6] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, 1 (3):12–21, 1993.

[7] Ananth Grama and Vipin Kumar. Scalability analysis of partitioning strategies for finite element graphs. In *Supercomputing '92 Proceedings*, pages 83–92, 1992.

[8] Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[9] Anshul Gupta, Vipin Kumar, and Ahmed Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. Technical Report TR 92-64, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1992. A short version appears in the *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, 1993.

[10] H. F. Ho, G. H. Chen, S. H. Lin, and J. P. Sheu. Solving linear programming on fixed-size hypercubes. In *International Conference on Parallel Processesing*, pages 112–116, 1988.

[11] George Karypis, Anshul Gupta, and Vipin Kumar. A parallel formulation of interior point algorithms. Technical Report (TR 94-20), Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994.

[12] George Karypis and Vipin Kumar. Unstructured Tree Search on SIMD Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 1994 (to appear). available as Technical Report TR 92-21, Computer Science Department, Un iversity of Minnesota.

[13] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[14] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, October 1991. A short version appears in the *Proceedings of the International Conference on Parallel Processing*, 1990.

[15] K. Onaga and H. Nagayasu. A wavefront-driven algorithm for linear programming on dataflow processor-arrays. In *Proceedings of International Computer Symposium*, pages 739–746, 1984.

[16] C. B. Stunkel and D. A. Reed. Hypercube implementation of the simplex aglorithm. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume ——, pages 1473–1482, 1988.

[17] C. B. Stunkel, D. C. Rudolph, W. K. Fuchs, and D. A. Reed. Linear optimizations: A case study in performance analysis. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, volume I, pages 265–268, 1989.