Sparse Implementation of Revised Simplex Algorithms on Parallel Computers*

Wei Shu and Min-You Wu

Abstract

Parallelizing sparse simplex algorithms is one of the most challenging problems. Because of very sparse matrices and very heavy communication, the ratio of computation to communication is extremely low. It becomes necessary to carefully select parallel algorithms, partitioning patterns, and communication optimization to achieve a speedup. Two implementations on Intel hypercubes are presented in this paper. The experimental results show that a nearly linear speedup can be obtained with the basic revised simplex algorithm. However, the basic revised simplex algorithm has many fill-ins. We also implement a revised simplex algorithm with LU decomposition. It is a very sparse algorithm, and it is very difficult to achieve a speedup with it.

1 Introduction

Linear programming is a fundamental problem in the field of operations research. Many methods are available for solving the linear programming problems, in which the simplex method is the most widely used one for its simplicity and speed. Although it is well-known that the computational complexity of the simplex method is not polynomial in the number of equations, in practice it can quickly solve many linear programming problems. The interior-point algorithms are polynomial and potentially easy to parallelize [7, 1]. However, since the interior-point algorithm has not been well-developed yet, the simplex algorithm is still applied to most real applications.

Given a linear programming problem, the simplex method starts from an initial feasible solution and moves toward the optimal solution. The execution is carried out iteratively. At each iteration, it improves the value of the objective function. This procedure will terminate after a finite number of iterations. The simplex method was first designed in 1947 by Dantzig [3]. The revised simplex method is a modification of the original one which significantly reduces the total number of calculations to be performed at each iteration [4]. In this paper, we will use the revised simplex algorithm to solve the linear programming problem.

Parallel implementations of the linear programming algorithms have been studied on different machine architectures, including both distributed and shared memory parallel machines. Sheu *et al.* presented a mapping technique of linear programming problems on the BNN Butterfly parallel computer [10]. The performance of parallel simplex algorithms was studied on the Sequent Balance shared memory machine and a 16-processor Transputer system from INMOS Corporation [12]. The parallel simplex algorithms for a loosely coupled message-passing based parallel systems were presented in [5]. The implementations

*This work appeared in the proceeding of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, March 22-24, 1993, Norfolk. This research was partially supported by NSF grants CCR-9109114 and CCR-8809165

of simplex algorithm on the fixed-size hypercubes was proposed by Ho *et al.* [6]. The performance of the simplex and revised simplex method on the Intel iPSC/2 hypercube was examined in [11]. All existing works are for dense algorithms and there is no sparse implementation.

Parallel computers are used to solve large application problems. Large-scale linear programming problems are very sparse. Therefore, dense algorithms are not of practical value. We implemented two sparse algorithms. The first one, the revised simplex algorithm, can be a good parallel algorithm but has many fill-ins. It is easy to parallelize and can yield good speedup. The second one, the revised simplex algorithm with LU decomposition [2], is one of the best sequential simplex algorithms. It is a sophisticated algorithm with few fill-ins, however, it is extremely difficult to parallelize and to obtain a speedup, especially on distributed memory computers. Actually, it is one of the most difficult algorithms we ever implemented.

2 Revised Simplex Algorithm

The linear programming problems can be formulated as follows:

$$solve \quad Ax = b \quad while \quad minimizing \quad z = c^T x$$

where A is a $m \times n$ matrix representing m constraints, b is the right hand side vector, c is the coefficient vector that defines the objective function z, and x is a column of n variables.

A fundamental theorem of linear programming states that an optimal solution, if it exists, occurs when $(n - m)$ components of x are set to zero. In most practical problems, some or all constraints may be specified by linear inequalities, which can be easily converted to linear equality constraints with the addition of nonnegative *slack variables*. Similarly, *artificial variables* may be added to maintain x nonnegative [8]. We use the two-phase revised simplex method, in which the artificial variables are driven to zero in the first phase, and an optimal solution is obtained in the second phase.

A *basis* consists of m linearly independent columns of matrix A, represented as $B = [A_{j_1}, A_{j_2}, ..., A_{j_m}]$. A *basic solution* can be obtained by setting $(n - m)$ components of x to zero and solving the resulting $m \times m$ linear equations. These m components are called the *basic variables*; the rest of $(n - m)$ components are called the *nonbasic variables*. Each set of basic variables represents a solution, but not necessarily the optimal one. To approach the optimal, we need to vary the basis by exchanging a basic variable with a nonbasic variable each time, such that the new basis represents a better solution than the one formed previously. The variable that enters the basis is called the *entering variable*, and the variable that leaves the basis is called the *leaving variable*. This process forms a kernel of the revised simplex method algorithm as shown in FIG 1. It will be iteratively repeated in both the phases. The slack and artificial variables are commonly chosen as an initial basis, which always results in a basic feasible solution.

3 Parallelization of Revised Simplex Algorithm

As shown in FIG. 1, the principle cost of determining the entering variable and the leaving variable is the same — a vector-matrix multiplication followed by a search for minimum value. These two steps contribute to the major computation part, more than 98% of the total computation in large test data. The third step updates the basic variable and basis inverse. The parallelization of the revised simplex algorithm, therefore, turns to applying the parallel vector-matrix multiplication and minimum reduction repeatedly for each iteration.

1. Determine the entering variable by using optimality condition,

$$c'_s = \min\{c'_j,\ \ j = 1, 2, \ldots, n \text{ and } x_j \text{ is nonbasic }\},\ \text{where } c'_j = c_j + \sum_{i=1}^{m} \pi_i a_{ij}$$

   If $c'_s \geq 0$, then the current value is optimal and algorithm stops.

2. Determine the leaving variable by using feasibility condition,

$$\frac{b_r}{y_{rs}} = \min\{\frac{bi}{y_{is}},\ i = 1, 2, \ldots, m \text{ and } y_{is} > 0\},\ \text{where } Y_s = y_{is}, i = 1, 2, \ldots, m \text{ and } Y_s = B^{-1}A_s$$

   If $y_{is} \leq 0$, then the problem is unbounded and algorithm stops.

3. The basic variable in row r will be replaced by variable x_s and the new basis

$$B = [A_{j1}, \ldots, A_{j_{r-1}}, A_s, A_{j_{r+1}}, \ldots, A_{jm}]$$

   Update the basis inverse B^{-1}, the vector of simplex multipliers π, and the right hand side vector b by

$$B^{-1} = B^{-1} - Y_s B^{-1}(r, :)/y_{rs}$$

$$\pi = \pi - c'_s\ B^{-1}(r, :)$$

$$b = b - Y_s\ \frac{b_r}{y_{rs}}$$

FIG. 1. *The Revised Simplex Algorithm*

A is a sparse matrix with two properties, (1) it is accessed through column exclusively, therefore, using a column-by-column storage scheme; and (2) it is read only during the entire computation. The nonzero elements of matrix A are packed into a one-dimensional array aA, where $aA(k)$ is the kth nonzero element in A with column-major order. Another array iA, with the same size of aA, contains the row indices which correspond to the nonzero elements of A. In addition, a one-dimension array jA is used to point out the index of aA where each column of A begins.

Matrix A is partitioned column-wise, due to the nature of multiplication of vector and matrix. If the row partitioning is chosen, a row-wise reduction will introduce heavy communication. Notice that the sparsity of matrix A is not evenly distributed. For example, the right portion of matrix A, consisting of slack and artificial variables with only one nonzero element for each column, is usually more sparse compared to the left portion. In order to make computation density evenly distributed, a scatter column partitioning scheme is used [9].

Vector π is duplicated in all nodes to eliminate communications while doing the multiplication. Besides, vector π is scattered into a full-length vector of storage. Thus, the multiplication of vector π and matrix A can be performed efficiently because of (1) searching for π_i is not needed; (2) the amount of work performed depends only on the number of nonzero elements in matrix A. The entire multiplication is then performed in parallel without any communication involved, leaving the result vector c' distributed. A reduction is performed to find out the minimal value from c'. This completes the first multiplication and reduction.

Compared to the first multiplication, the second one is a multiplication of a matrix to a vector, instead of a vector to a matrix. Matrix B^{-1} is then partitioned by row to avoid communications. Vector A_s is also duplicated by broadcasting column s of A once the entering variable is selected. This broadcast serves as the only communication interface between the two steps and the rest is the same as in the first one.

In the third step, the basic variable in row r will be replaced by variable x_s and the new basis B is given by

$$B = [A_{j_1}, ..., A_{j_{r-1}}, A_s, A_{j_{r+1}}, ..., A_{j_m}]$$

The basis B is not physically represented. Instead, the basis inverse B^{-1} is manipulated and needs to be updated. This update may generate many fill-ins, therefore, the linked list scheme is used for B^{-1} storage.

With the scatter partitioning technique, the load can be well balanced. The performance in TABLE 1 shows that a nearly linear speedup can be obtained with this algorithm.

TABLE 1

Execution Time (in seconds) of the Revised Simplex Algorithm

	Matrix Size	Number of Processors 1	2	4	8	16	32
SHARE2B	97*79	9.02	4.96	3.66	3.44	3.32	2.58
SC105	106*103	15.6	8.58	5.73	4.46	3.01	2.90
SC205	206*203	158	88.7	43.5	26.5	15.6	11.5
BRANDY	221*249	278	162	90.7	59.9	38.0	27.5
BANDM	306*472	1330	718	378	209	127	80.2

4 Revised Simplex Algorithm with LU decomposition

The standard revised simplex algorithm mentioned above is based on constructing the basis inverse matrix B^{-1} and updating the inverse after every iteration. While this standard algorithm bares the advantages of its simplicity and easiness for parallelization, it is not the best one in terms of its sparsity and its round-off error behavior. This algorithm has many fill-ins in B^{-1}. Large number of fill-ins not only increase the execution time, but also occupy too much memory space. Consequently, large test data cannot fit in the memory. Another alternative is the method of Bartels and Golub that is based on the LU decomposition of the basis matrix B with row exchanges [2]. Compared to the standard one, the new one differs in the way to solve two linear equations:

$$\pi^T B = -c_B^T \qquad and \qquad BY_s = A_s$$

With the original Bartels-Golub algorithm used, the LU decomposition needs to be applied for the newly constructed basic matrix B in each iteration. This operation is computationally more expensive than the one used to update the basis inverse B^{-1} in the standard case. To compromise, a better implementation is to take the advantage of the fact that the newly constructed B_k differs from the preceding B_{k-1} in only one column after k iterations. Here, we can construct an *eta matrix* E, which differs from the identity matrix in only one column, referred to as its *eta column*, to satisfy

$$B_k = B_{k-1}E_k = B_{k-2}E_{k-1}E_k = ... = B_0E_1E_2...E_k$$

This *eta factorization*, therefore, suggests another way of solving the two linear equations at iteration k:

$$((((\pi^T B_0)E_1)E_2)...)E_k = -c_B^T \qquad and \qquad B_0(E_1(E_2(...(E_kY_s)))) = A_s$$

where B_0 will be decomposed into L and U. The E solver is a special case of the linear system solver, which consists of a sequence of k *eta* column modification to the solution vector. Assume we use an array p to reserve each *eta* column position in its original *eta* matrix. To solve the original *eta* matrix system, the lth modification can be described as follows,

$$E_l(E_{l+1}(...(E_k X))) = Z_{l-1} \quad \text{can be modified to} \quad E_{l+1}(...(E_k X)) = Z_l$$

$$\text{where } z_{l,p(l)} = z_{l-1,p(l)}/e_{l,p(l)} \quad \text{and} \quad z_{l,i} = z_{l-1,i} - e_{l,i} \times z_{l,p(l)}, i \neq p(l)$$

After k modifications, Z_k is the resulting vector.

However, the number of nonzero elements of E grows with the number of iterations. As k increases, the storage of all *eta* matrices becomes large and the time spent on the E solver is slow too. Therefore, after a certain number of iterations, say r, the basis B_0 needs to be *refactorized* [2]. That is, all the *eta* matrixes will be discarded. The current B_k is treated as a new B_0 to be decomposed. Such an LU decomposition needs to be conducted every r iterations to keep the time and storage of E solver within an acceptable range. This improved algorithm can be summarized in FIG 2. In our implementation, r is set to 40.

---

1. Determine the entering variable by using optimality condition,
$$c'_s = \min\{c'_j, \; j = 1, 2, \ldots, n \text{ and } x_j \text{ is nonbasic }\}, \text{ where } c'_j = c_j + \sum_{i=1}^{m} \pi_i a_{ij}$$
If $c'_s \geq 0$, then the current value is optimal and algorithm stops.

2. Solving $B_0(E_1(E_2(...(E_k Y_s)))) = A_s$
If $y_{is} \leq 0$ for $i = 1, 2, \ldots, m$, then the problem is unbounded and algorithm stops.

3. Determine the leaving variable by using feasibility condition,
$$\frac{b_r}{y_{rs}} = \min\{\frac{bi}{y_{is}}, \; y_{is} > 0 \text{ and } i = 1, 2, \ldots, m\}$$

4. The basic variable in row r will be replaced by variable x_s and the new basis
$$B = [A_{j_1}, \ldots, A_{j_{r-1}}, A_s, A_{j_{r+1}}, \ldots, A_{j_m}]$$

5. If $(k \geq r)$ then treat B_k as the new B_0 and decompose B_0 to get the new matrices L and U, and reset k to 1, else store $E_{k+1} = Y_s$ and increment k

6. Solving $((((\pi^T B_0)E_1)E_2)...)E_k = -c_B^T$

FIG. 2. *The Revised Simplex Algorithm with LU Decomposition*

5 Parallelization of Revised Simplex Algorithm with LU Decomposition

Parallelization of the simplex algorithm with LU decomposition is more difficult. As experiments indicated, the major computation cost is distributed among the matrix multiplication in *step 1* of FIG 2; the linear system solvers and E solvers for Y_s in *step 2* and one for π in *step 6*, respectively, and the LU decomposition in *step 5* if applied. The parallelization of the matrix multiplication of *step 1* is the same as the one in the standard algorithm, as well as the storage scheme and the data partitioning strategy.

For the two linear system solvers, the basis matrix B_0 serves as the coefficient matrix for r iterations before the next refactorization takes place. We do not expect a good speedup from the triangular solver, because the matrices L and U are very sparse and the computation time spent on the triangular solvers is small compared to the other steps. In consideration of scalability, we distribute the factorized triangular matrices L and U to minimize the overhead of $2r$ triangular solvers. For either L or U, partitioning by row is a simple way to reduce overhead. Furthermore, a block partitioning is used to increase the grainsize between two consecutive communications. Although a scatter partitioning scheme can improve load balance theoretically, a large number of communication, proportional to the number of rows m, will overwhelm this potential, especially for a sparse matrix case. Here, with a block-row partitioning scheme, the total number of communications for each time of triangular solver is equal to the number of physical processors used, independent of m.

It is not conceivable to parallelize the E solver, because (i) the lth *eta* column modification depends on the results from the $l-1$th one; (ii) the modification of each element depends on the value of the diagonal element in its *eta* column; and (iii) a small amount of computation is involved in each modification. Due to such heavy data dependences and small grainsizes, the E solver, if executed in parallel, will not be beneficial; therefore, it is currently executed sequentially and all the *eta* columns are stored in a single physical processor, with two assumptions:

- if time spent on this sequential part is too long, we can adjust the frequency of refactorization to make it under control.
- in consideration of the scalability, if needed, we can always distribute k *eta* columns and have the column modifications executed in turn for each processor;

The basis refactorization is performed every r iterations. A new algorithm was used to factorize the basis matrix efficiently [13]. The idea is to take advantages of the fact that the basis matrix B is extremely sparse and has many column singletons and row singletons. As part of the matrix is phased out by rearranging singletons, the remaining matrix to be factorized may have new singletons generated. Therefore, the column or row exchange of singletons should be reapplied continuously until no more singleton exists. At that time, the rest of the matrix can be factorized in a regular manner. By the column and row exchanges of singletons, we can reduce the size of the submatrix that is actually factorized, and sometimes, eliminate the factorization completely. During the period of exchanging of singletons, first, no fill-ins will be introduced since no updating; and secondly, searching for the singletons can be easily conducted in parallel since there is no updating, therefore, no dependency.

In general, a simple application algorithm has only one single computation task on a single data domain. For this class of problems, the major data domain should be distributed such that the computation task can be executed most effectively in parallel. The LU decomposition, matrix multiplication, triangular solver, etc. belongs to this class. However, a complex application problem, such as the revised simplex method with LU decomposition, consists of many computation steps and many underlying data domains. Usually, these computation steps depend on each other. If we simply distribute the underlying data domains in consideration of parallelization of individual computation steps, it may ends up a mismatch between two underlying data domains. A global optimization may not be reached simply with local optimal data distributions for each step. Sometimes a sub-optimal local data distribution must be used to reduce interface overhead.

There are two major interfaces in this algorithm. The first one is the interface between matrix A and basis B. The current new basis is constructed by collecting m columns from matrix A. Remember that matrix A has been distributed with scatter-column partitioning.

One way to construct matrix B is to extract the columns of the basic variables from A and redistribute them to obtain local optimal data distribution. However, data redistribution between different computation steps can be beneficial only when it is crucial for performance of the next step. Here, the refactorization is not expected for a perfect speedup and the benefit from redistribution is less than the cost of redistribution. Therefore, we leave the extracted columns in the physical processor where they reside. With the distribution inherited from matrix A, the basis matrix B is not necessarily evenly distributed among processors, leading to a sub-optimal data distribution.

The second interface is between basis B and matrices L and U after refactorization. Since L and U are to be used in the following triangular solvers for $2r$ times, it is worthwhile to carry out the redistribution. The unbalanced scatter-column basis B is redistributed to a block-row scheme for L and U, which is an optimal scheme used for the triangular solver. Without redistribution, matrices L and U were distributed in column-scatter scheme and the communication cost in the following triangular solvers would be even worse.

This algorithm is very sparse, therefore, difficult to parallelize due to small granularity. The triangular solvers and E solvers are hard to get speedup and redistribution involves in large communication overhead. The basic strategy used in this implementation is to apply the best possible parallelization techniques for the most computational dense part — matrix multiplication — to obtain maximum speedup. This part spends more than 70% of the total execution time and has speedup of around 7.5 on 8 processors. The LU decomposition part spends less than 10% of total time and rarely has a speedup. The triangular solvers and the E solvers spend about 10% of total time and the parallel execution time increases with the number of processors, dominating the overall performance. The performance is shown in TABLE 2. Compared it with Table 1, BRANDY and BANDM are executed much faster on a single processor, but have almost no speedup. When the number of processors is larger than 8, these two data sets run slower. For large data, such as SHIP08L, SCSD8, and SHIP12L, up to two-fold speedup can be obtained. More importantly, these data cannot be run with the basic revised simplex algorithm since they cannot fit in the memory.

TABLE 2

Execution Time (in seconds) of the Revised Simplex Algorithm with LU decomposition

	Matrix Size	Number of Processors			
		1	2	4	8
BRANDY	221*249	72.5	84.2	–	–
BANDM	306*472	371	344	331	342
SHIP08L	779*4283	523	407	329	334
SCSD8	398*2750	674	590	560	682
SHIP12L	1152*5427	2104	1344	1076	1020

6 Conclusions

Two revised simplex algorithms, with and without LU decomposition, have been implemented on the Intel iPSC/2 hypercube computers. The test data sets are from netlib. These data sets represent realistic problems in industry applications, ranging from small-scale to large-scale. The execution time is from a few seconds to thousands of seconds. Although the basic revised simplex algorithm is relative easier to implement, both of them need to be carefully tuned to get the best possible performance. We emphasized on sophisticated parallelization techniques, such as partitioning patterns, data distribution, communication reduction, and interface between computation steps. Our experience shows that for a complicated problem, each step of computation requires different data distribution. The interfaces between steps affect the decision of data distribution, and consequently, communications.

In summary, parallelizing sparse simplex algorithms is not easy. The experimental results show that a nearly linear speedup can be obtained with the basic revised simplex algorithm. On the other hand, because of very sparse matrices and very heavy communication, the revised simplex algorithm with LU decomposition is hard to achieve good speedup even with carefully selected partitioning patterns and communication optimization.

Acknowledgments

The authors wish to thank Yong Li for his contribution on linear programming algorithms and sequential codes.

References

[1] G. Astfalk, I. Lusting, R. Marsten and D. Shanno, *The Interior-Point Method for Linear Programming*, IEEE Software, July, 1992, pages 61–67.

[2] R. H. Bartels and G. H. Golub, *The Simplex Method of Linear Programming Using LU Decomposition*, Comm. ACM, 12, pages 266-268, 1969.

[3] G. B. Dantzig. *Linear Programming and Extensions*, Princeton University Press, New Jersey, 1963.

[4] G. B. Dantzig and W. Orchard-Hays, *The product form for the inverse in the simplex method*, Mathematical Tables and Other Aids to Computation, 8, pages 64-67, 1954.

[5] R. A. Finkel, *Large-grain Parallelism – Three Cases Studies*, The Characteristics of Parallel Algorithms, L. H. Jamieson ed., The MIT Press, 1987.

[6] H. F. Ho, G. H. Chen, S. H. Lin, and J. P. Sheu, *Solving Linear Programming on Fixed-Size Hypercubes*, pages 112-116, ICPP'88.

[7] N. Karmarkar, *A New Polynomial-Time Algorithm for Linear Programming*, Combinatorica, Vol. 4, No. 8, 1984, pages 373-395.

[8] Y. Li, M. Wu, W. Shu and G. Fox, *Linear Programming Algorithms and Parallel Implementations*, SCCS Report 288, Syracuse University, May 1992.

[9] D. M. Nicol and J. H. Saltz. An analysis of scatter decomposition. *IEEE Trans. Computers*, C–39(11):1337–1345, November 1990.

[10] T. Sheu and W. Lin, *Mapping Linear Programming Algorithms onto the Butterfly Parallel Processor*, 1988.

[11] C. B. Stunkel and D. C. Reed, *Hypercube Implementation of the Simplex Algorithm*, Prof. 4th Conf. on Hypercube Concurrent Computer and Applications, March 1989.

[12] Y. Wu and T. G. Lewis, *Performance of Parallel Simplex Algorithms*, Department of Computer Science, Oregon State University, 1988.

[13] M. Wu, and Y. Li, *Fast LU Decomposition for Sparse Simplex Method*, SIAM Conference on Parallel Processing for Scientific Computing, March 1993.