Management Science Research Report No. MSRR 610

SOLVING LINEAR PROGRAMS USING UNDISTRIBUTED PARALLEL COMPUTING

by

N. R. Natraj*
G. L. Thompson**
F. Harche**

November 1994

* Graduate School of Industrial Administration
Carnegie Mellon University

** Stern School of Business
New York University

This report was prepared as part of the activities of the Management Science Research Group, Carnegie Mellon University, under contract No. N00014-85-K-0198 NR 047=048 with the Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U. S. Government.

Management Sciences Research Group
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

Solving Linear Programs Using Distributed Parallel Computing *

N. R. NATRAJ

G. L. THOMPSON

Graduate School of Industrial Administration

Carnegie Mellon University

F. HARCHE

Stern School of Business

New York University

November 1994

*The authors thank Peter Steenkiste of the Computer Science Department at Carnegie Mellon for his help and suggestions regarding the implementation on Nectar

Abstract

Parallel and distributed computing has attracted a lot of attention from researchers over the past few years. It is a promising new approach for solving large problems that were hitherto considered very difficult to solve using traditional serial computers. The advancement of technology and the introduction of fiber-optic networks for high speed data transmission has made distributed computing on a network of computers very attractive. In this paper we address the problem of solving linear programs using distributed computing. We present a 2-phase parallel standard simplex algorithm for solving linear programs with single upper bounded variables. The parallel simplex algorithm was implemented on NECTAR (NEtwork CompuTer ARchitecture) a collection of SUN4/330 workstations on a fiber-optic network and evaluated using randomly generated problems and those available from the *netlib* database.

Parallel and distributed computing has attracted a lot of attention over the past several years. It offers a new, promising approach to solving large problem that were, till now, considered difficult to solve using conventional serial computers. The advancement of communications network technology and the introduction of high speed fiber-optic data transmission networks has made distributed computing over a network of computers very attractive. Parallel/distributed computing may potentially be utilized to more efficiently solve difficult linear programming problems. Linear programming is frequently used as a subroutine for solving more difficult optimization problems like integer programs. Since linear programs are widely solved by matrix manipulation, and since matrix algorithms are excellent candidates for parallelizing, it seems natural to parallelize the simplex method.

However, the efficacy of the parallelized simplex method may depend on the density of the constraint matrix. In practice, most of the linear programs solved have a sparse constraint matrix so that the revised simplex method [4] runs substantially faster than the standard simplex method. We believe that parallelizing the revised simplex method for distributed computing involving high communication overheads may not be very successful. When the constraint matrix is dense it is very likely that the standard simplex method runs faster than the revised simplex method. This might be the case after the addition of a large number of strong cuts when linear programming is used as a subroutine in a branch-and-cut [14] framework to solve integer programming problems or after addition of valid inequalities to tighten the initial linear programming relaxation of the problem. In dense problems, lack of enough memory can cause the revised simplex method to do memory paging to disk resulting in considerable degradation in performance. These factors point to the usefulness of parallelizing the standard simplex method to solve dense problems. Distributed computing also offers an opportunity to make better use of the available computing resources that are idle during certain periods in the day.

Despite the enormity of literature on parallel algorithms for various problems (see [3] for a partial list), very little work has been done on parallelizing the simplex method for

linear programs. Most of the work on parallelizing the simplex method has concentrated on solving problems with special structure like network flow problems [1], staircase linear programs [15, 5, 6, 9], two stage stochastic linear programs [6] or transportation problems [12].

In general, an approach to solving large problems in parallel consists of breaking the problem into a number of small tasks each of which can be manipulated by its own processor. These tasks are coordinated using a control mechanism. The most commonly used parallel structure is the master-slave structure. Here many slave tasks work synchronously under the control of the master task that has close control over the computations performed by slave tasks. Since one of the objectives of parallel processing is to solve large problems in a shorter period of time ("real") the performance of such systems is measured by the *speedup*, S_p, defined as the ratio of the solution time using the serial algorithm to the time using a parallel algorithm on p processors.

In this paper we present a 2-phase parallel standard simplex algorithm for solving linear programs with SUB variables. The algorithm is described in section 1. In section 2, we present details of the NECTAR architecture on which the algorithm was implemented. In section 3, we present some computational experience on the implementation of the parallel simplex algorithm on NECTAR using Nectarine [16] on randomly generated problems. We also present results on the performance of parallel simplex on a few problems from the *netlib* database [8, 10].

1 Parallel Simplex Algorithm

During the last two decades, researchers have been quite successful in converting some of the familiar serial algorithms to efficient parallel algorithms. Since solving linear programs using the simplex method is essentially a matrix algorithm, the parallel algorithms profit by taking advantage of the parallelism inherent in matrix manipulation.

Consider a linear program (P), stated as follows:

$$
\begin{aligned}
\max \quad & cx \\
\text{subject to} \quad & \\
& A^{(1)}x \leq b^{(1)} \qquad (P) \\
& A^{(2)}x = b^{(2)} \\
& 0 \leq x \leq u
\end{aligned}
$$

where $b^{(1)}$ and $b^{(2)}$ are not necessarily non-negative, $A^{(1)}$ is an $m_1 \times n$ matrix, $A^{(2)}$ is an $(m - m_1) \times n$ matrix and all other matrices are of appropriate dimension.

Since the right hand side vectors are not restricted to be non-negative such linear programs are solved using a two phase simplex method. The purpose of phase 1 in a two phase simplex method is to find a basic feasible solution to the problem. Most commercial implementations of the two phase simplex method use the objective of minimizing the sum of infeasibilities during phase 1 [7, 11, 13]. This objective function is defined in terms of the variables that violate either the upper bound or the lower bound.

Our parallelization technique consists of partitioning the tableau row-wise (horizontal parallelism) into p independent subtableaus, where p denotes the number of processors working simultaneously on the problem. Let I_k denote the indices of the rows in processor k. Each subtableau is treated like an individual tableau on a serial processor and the operations involved in the simplex method are applied to it. Since we have two different objective functions for phase 1 and phase 2, the number of rows in the compact representation of tableau is $m + 2$ and the number of columns is $n + 1$. Henceforth we use a_{ij} to denote any element in the tableau with a_{*0} denoting the right hand side vector b. The $(m + 2)$ rows are divided as equally as possible between the p processors. The *master processor* contains both the

objective function rows. We will call the other $p-1$ processors, *slave processors.*

The Simplex method [4] searches for an optimal solution by going from one basic solution to another making sure that the new solution has an objective function value that is no worse than the current solution. The procedure terminates when the objective function row indicates that there are no other solutions that are better than the current solution. Going from one basic solution to another involves the following steps

1. Identifying a variable that can enter the basis
2. Identifying a variable that can leave the basis
3. Performing a pivot and updating the tableau for the new basic solution.

1.1 Choosing the entering variable and pivot column

During the process of choosing the entering variable or the pivot column J, only the master processor is at work because the objective function rows are stored only in the master processor. Choosing the pivot column involves finding the smallest element (most negative element) in the appropriate objective function row determined by the current phase of the simplex method.

In phase 1 of the simplex method we use an objective of minimizing the sum of infeasibilities. The objective function row is calculated as follows. Let

$B = \{\beta_1, \beta_2, \ldots, \beta_m\}$ denote the indices of the variables in the current basis and N denote the indices of the non-basic variables.

Let us define

$B_1 = \{i | \beta_i \in B$ and x_{β_i} violates its upper bound$\}$

$B_2 = \{i | \beta_i \in B$ and x_{β_i} violates its lower bound$\}$

Then the sum of infeasibilities can be written as

$\sum_{i \in B_1} (x_{\beta_i} - u_{\beta_i}) - \sum_{i \in B_2} x_{\beta_i}$

$= \sum_{i \in B_1} (a_{i0} - \sum_{j \in N} a_{ij} x_j - u_{\beta_i}) - \sum_{i \in B_2} (a_{i0} - \sum_{j \in N} a_{ij} x_j)$

In other words, the phase 1 objective function can be calculated by adding the rows of the tableau belonging to B_1 and subtracting the rows of the tableau belonging to B_2. In the parallel implementation, each slave processor performs this operation independently on its rows and sends a message containing its contribution to the objective function to the master processor. The master processor sums the objective function row from each one of the slave processors along with its own to determine the phase 1 objective function row for that iteration.

Repeating this for a number of iterations, we will either find a feasible basis or conclude that the linear program is infeasible because there is no entering variable that reduces the sum of infeasibilities. If we find a feasible basis we restore the original objective function and begin phase 2 of the simplex method.

The master processor identifies the pivot column by finding the most negative element in the appropriate objective function row and communicates it to all the slave processors. Upon receipt of the pivot column information all processors work in parallel to find the pivot row.

1.2 Choosing the leaving variable and pivot row

We identify the leaving variable in phase 1 using the following two-pass procedure [13].

Let $B = \{\beta_1, \beta_2, \ldots, \beta_m\}$ denote the indices of the variables in the current basis, J denote the index of the entering column and γ_J denote the index of the entering variable. For each basic variable one can determine a set of upto two threshold levels. These threshold levels determine when the basic variable reaches one of its bounds as we increase the value of the non-basic variable. They can be calculated as follows:

If $x_{\beta_i} > u_{\beta_i}$ and $a_{iJ} > 0$ the two thresholds are $t_1 = \frac{x_{\beta_i} - u_{\beta_i}}{a_{iJ}}$ and $t_2 = \frac{x_{\beta_i}}{a_{iJ}}$

else if $x_{\beta_i} < 0$ and $a_{iJ} < 0$ the two thresholds are $t_1 = \frac{x_{\beta_i}}{a_{iJ}}$ and $t_2 = \frac{x_{\beta_i} - u_{\beta_i}}{a_{iJ}}$

else if $0 < x_{\beta_i} < u_{\beta_i}$ and $a_{iJ} < 0$ the threshold is $t = \frac{x_{\beta_i} - u_{\beta_i}}{a_{iJ}}$

else if $0 < x_{\beta_i} < u_{\beta_i}$ and $a_{iJ} > 0$ the threshold is $t = \frac{x_{\beta_i}}{a_{iJ}}$

The two pass procedure to determine the leaving variable is as follows:

- Pass 1: For each basic variable $x_{\beta_i}, i = 1, \ldots, m$ determine a set of upto two threshold levels from the associated bounds and define the corresponding triples of the form $[t, i, \alpha]$ where t determines the threshold level as determined above, i denotes the row index of the basic variable x_{β_i} and $\alpha = |a_{iJ}|$.

- Pass 2: Sort the entire list of triples in increasing values of t. Let us denote the elements of this sorted list by $[t_k, i_k, \alpha_k], k = 1, \ldots, K$. Let σ_J^1 denote the reduced cost of the variable corresponding to the entering column J. Calculate the partial sums,

  $S_k = \sigma_J^1 + \sum_{i=1}^{k} \alpha_i$ for $k = 0, \ldots, K$ where $\sum_{i=1}^{0} \alpha_i = 0$.

  Find the index $\hat{k}$ such that $S_{\hat{k}-1} < 0$ and $S_{\hat{k}} \geq 0$. The row corresponding to the leaving variable is $i_{\hat{k}}$.

In the parallel implementation every processor executes Pass 1 of the above procedure independently on its set of rows. Each slave processor then sends a message containing all the triples obtained from its rows to the master processor. The master processor collects all the triples and performs Pass 2 of the above procedure to identify the row corresponding to the leaving variable i.e., the pivot row. The master processor then sends a message to each slave processor identifying the processor containing the pivot row and the index of the pivot row.

In phase 2 of the simplex method the leaving variable is identified using the standard minimum ratio rule. Since we have a feasible basis and want to maintain a feasible basis in every iteration of phase 2 we calculate upto one threshold level for each basic variable as described below.

If $a_{iJ} > 0$ the threshold is $t = \frac{x_{\beta_i}}{a_{iJ}}$ resulting in a regular pivot.

else if $a_{iJ} < 0$ the threshold is $t = \frac{x_{\beta_i} - u_{\beta_i}}{a_{iJ}}$ resulting in a basic complementation pivot.

The threshold level from to the entering variable, x_{γ_J}, is u_{γ_J} resulting in a non-basic complementation pivot.

In the parallel implementation each processor calculates the minimum such threshold level for its rows. Each slave processor then sends a message containing the minimum threshold level, the index of the corresponding row (pivot row) and the kind of pivot (regular, basic complementation, non-basic complementation) corresponding to the minimum threshold level to the master processor. The master processor sorts the threshold levels from all the processors to identify the minimum ratio and sends a message to each slave processor containing information about the processor owning the pivot row, the pivot kind and the index of the pivot row.

1.3 Performing the pivot

The bulk of the computation in the simplex method is done during the process of pivoting and updating the tableau. Updating the tableau exhibits a large amount of inherent parallelism and the parallel simplex method exploits it. Since we are using the simplex method for SUB variables there are three possible kinds of pivots.

1. Regular pivot
2. Basic complementation pivot
3. Nonbasic complementation pivot

Based on the message sent by the master processor in the previous step, all the processors have information about the processor owning the pivot row, the corresponding pivot kind and the index of the pivot row. Let $\hat{p}$ denote the processor that owns the pivot row indexed by h. During the course of a regular pivot the following steps are carried out. Processor $\hat{p}$ updates the pivot row and communicates the row to all the other processors. Upon receipt of the row all p processors simultaneously update the tableau as follows.

$$a'_{ij} = a_{ij} - \frac{a_{iJ}}{a_{hJ}} a_{hj},\ i \in I_k,\ i \neq h,\ j \neq J \text{ and } k = 1, \ldots, p.$$
$$a'_{iJ} = -\frac{a_{iJ}}{a_{hJ}},\ i \in I_k,\ i \neq h, \text{ and } k = 1, \ldots, p.$$
$$a'_{hJ} = \frac{1}{a_{hJ}}$$

During the course of a nonbasic complementation pivot the tableau is updated by all the processors in parallel as follows.

$$a'_{iJ} = -a_{iJ},\ i \in I_k, \text{ and } k = 1, \ldots, p.$$
$$a'_{i0} = u_{x_J} a_{iJ} - a_{i0},\ i \in I_k, \text{ and } k = 1, \ldots, p.$$

During the course of a basic complementation pivot the tableau is updated only by processor $\hat{p}$ as follows.

$$a'_{hj} = -a_{hj},\ j = 1, \ldots, n.$$
$$a'_{h0} = u_{x_h} - a_{h0}.$$

It should however be noted that the iteration immediately following a basic complementation pivot on a row indexed h will always result in a regular pivot on the same row h.

The flow charts in figures 2 and 3 summarize the activities performed by the master processor and the slave processors, respectively.

2 About Nectar

In this section we present some details about the Nectar system. NEtwork CompuTer ARchitecture (Nectar) [16] is a collection of SUN 4/330 workstations on a fiber optic network. Is is a aysnchronous message-passing model where activities are coordinated on the basis of received messages. It consists of a crossbar based network and Communication Accelerator Boards (CABs) that connect nodes (workstations) to the network (Figure 1). The CABs are built around a general-purpose processor (the SPARC) and have memory to store messages.

The main function of CAB is to execute communication protocols. Communication in Nectar is based on variable-length, untyped messages that are sent to or received from buffers in CAB memory.

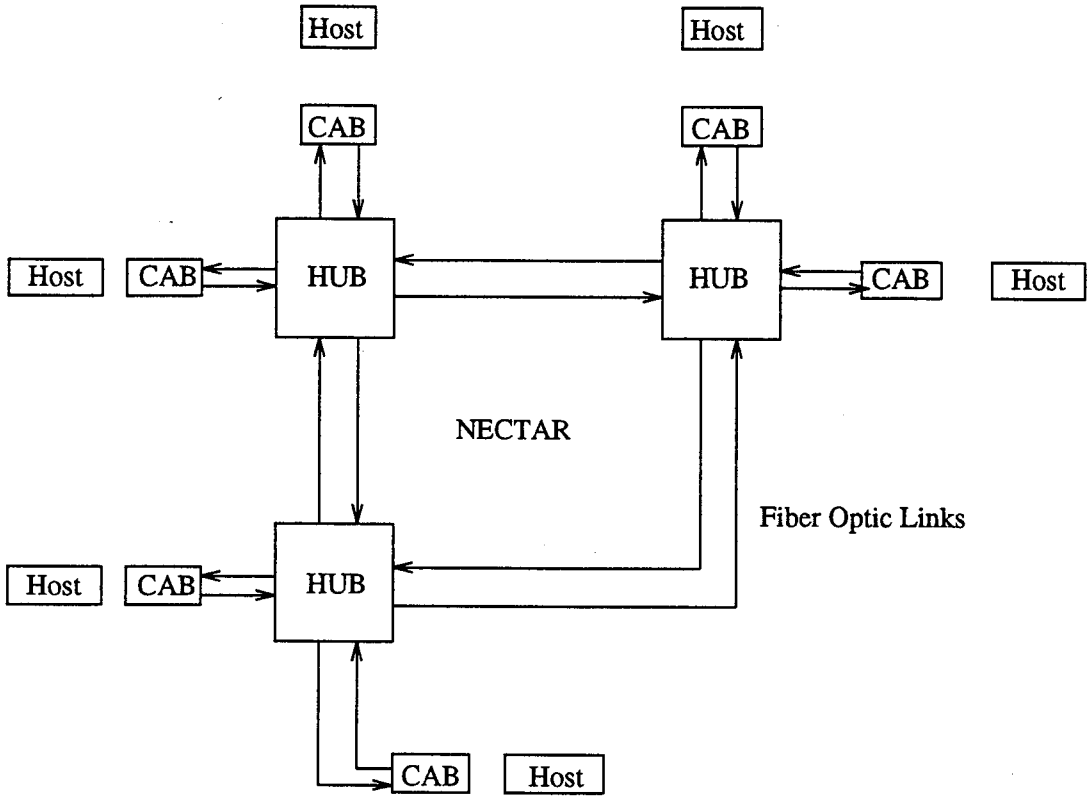

Figure 1: Nectar System (from [16])

Nectarine (Nectar Interface) is a low-level programming interface that gives applications full access to the Nectar hardware and low-level software. A task is an invocation of an executable piece of code that has the Nectarine library linked in. An application consists of a number of tasks executing on the nodes and CABs. Tasks communicate with each other by placing messages in and retreiving messages from buffers located in the CAB memory.

3 Computational Results

In this section, we discuss computational experience with an implementation of the parallel simplex algorithm in the C programming language using Nectarine to access the Nectar hardware and low-level software.

Since an over-riding reason for using a multiprocessor system is to decrease the time needed to solve problems, concern with the performance of such systems in solving those problems is quite natural. The most commonly used guage of a parallel algorithm's performance is speed-up. We use the following definitions in reporting our computational results. Let T_p be the computation time to solve the problem if $p \geq 1$ processors are used. The speed-up S_p of a parallel algorithm is defined by $S_p = \frac{T_1}{T_p}$. The corresponding efficiency E_p of the computation is given by $E_p = \frac{S_p}{p}$. One would hope to achieve an efficiency of 1, but this is usually impossible.

The computational experiment was designed to study the effect of problem size, problem shape and problems density on speed-up. All the experiments were performed on a lightly loaded system using randomly generated problems. The problems were generated as follows. The cost coefficients were chosen from U[10,20], the nonzero constraint matrix coefficients were chosen from U[10,20] and the nonzero right hand side of the constraints were chosen from U[25,80]. The nonzero constraint coefficients were chosen such that there was an equal distribution of positive and negative elements. All variables had a upperbound of 10. Figure 4 shows the effect of problem size on speed-up. As can be seen from the figure the speed-up obtained increases with problem size. However, the increase in speed-up obtained seems to tail-off as problem size increases. It should also be noted that as the number of processors increases the difference between the speed-up obtained and the ideal speed-up increases. This can be explained by the fact that as we increase the number of processors more time is spent waiting to synchronize computation thereby increasing communication overhead and decreasing efficiency. Figure 5 shows the effect of density on speed-up. The amount of

speed-up obtained seems to be consistent over all densities. This is probably because the tableau fills up quite fast during the pivoting process and hence the amount of work done is more or less identical irrespective of problem density. Figure 6 shows the effect of problem shape on speed-up. Keeping the number of elements in the tableau constant, the amount of speed-up obtained increases as the number of columns in the problem decreases. This can be explained by the fact that amount of time required to communicate the pivot row decreases as the number of columns in the problem decreases.

Recent literature on reporting computational results on parallel computing has suggested that reporting speedups should be done using the fastest available serial code [2]. Figures 7,8 and 9 compare the performance of parallel simplex with CPLEX 2.1. Since there is no easy way to make sure that the number of iterations required to solve a problem using parallel simplex and CPLEX 2.1 are identical, the results reported are based on time taken per iteration. Figure 7 shows the effect of problem size on speed-up with respect to CPLEX 2.1. The speed-up obtained decreases as problem size increases. This is because the number of iterations taken to solve the problem using parallel simplex increases proportional to the problem size, whereas with CPLEX 2.1 the increase is less than proportional to the problem size. Figure 8 shows the effect of problem density on speed-up. For the same problem size, the speed-up obtained with respect to CPLEX 2.1 increases with problem density. This is because the computational efficiency of the revised simplex method used by CPLEX 2.1 decreases with increasing problem density. Figure 9 shows the effect of problem shape on speed-up with respect to CPLEX 2.1. For the same number of elements in the tableau, the speed-up obtained increases as the number of columns decreases. As explained before, this is due to a decrease in the communication time with a decrease in the number of columns.

Next, we report some computational results obtained by using the parallel simplex method to solve three problems from the *netlib* database. Table 1 gives some details about the three problems and Table 2 gives the speed-up obtained with respect to serial simplex for different numbers of processors. These results illustrate the effect of the number phase

Problem	Size	Density
grow22	440×946	1.98%
scfxm2	660×914	0.86%
fff80000	524×854	1.39%

Table 1: Problem details

1 iterations on speedup. As can be seen from the table, for the same number of processors, the speedup of our parallel versus our serial code decreases across the three problems. This can be attributed to the fact that the number of phase 1 iterations increase across the three problems. As mentioned in section 1.2, since the slave processors communicate a lot of data to the master processor in each iteration of phase 1 the communication time and overhead increases resulting in a decrease in efficiency and speed-up. Unfortunately we were not able to obtain any speed-up with respect to CPLEX 2.1 because the problem density was very low.

Number of Processors	Speedups		
	grow22	scfxm2	fff80000
2	1.8	1.8	1.8
3	2.7	2.6	2.5
4	3.5	3.4	3.1
5	4.2	4.1	3.7

Table 2: Performance of Parallel simplex on 3 problems from the *netlib* database

4 Conclusions

In this paper we have developed a 2-phase parallel simplex algorithm and studied an implementation of the parallel algorithm on NECTAR. Computational results on an implementation of the parallel simplex method using Nectar indicates that consistent speed-ups can

be obtained with respect to the serial version when the number of phase 1 iterations is not much greater than the number of phase 2 iterations. In comparing parallel simplex with CPLEX 2.1 we were able to obtain a speed-up of about 2 on reasonably dense problems ($\geq$ 30%) that have a lot more rows than columns. Due to the sparsity of the problems in the *netlib* database performance of parallel simplex with respect to CPLEX 2.1 was rather poor.

References

[1] Barr, R.S., and B.L. Hickman, A New Parallel Network Simplex Algorithm and Implementation for Large Time-Critical Problems, Tech. Report 89-CSE-37, Southern Methodist University (November 1990).

[2] Barr, R.S., and B.L. Hickman, Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions, *ORSA Journal on Computing*, **5** (1993) 2-18.

[3] Bertsekas, D.P., and J.N.Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, New Jersey (1989).

[4] Chvatal, C, *Linear Programming*, W. H. Freeman, New York (1983).

[5] Entriken, R., A Parallel Decomposition Algorithm for Staircase Linear Programs, Tech. Report, SOL 88-21, Stanford University (December 1988).

[6] Entriken, R., The Parallel Decomposition of Linear Programs, Research Report RC, IBM Corp. Research Division (1989).

[7] Fourer, R., A Simplex Algorithm for Piecewise-linear Programming I: Derivation and Proof, *Mathematical Programming* **3** (1985) 205-233.

[8] Gay, D. M, Electronic Mail Distribution of Linear Programming Test Problems, *Mathematical Programming Society COAL Newsletter*, (December 1985).

[9] Ho, J. K., T. C. Lee, and R. P. Sundarraj, Decomposition of Linear Programs using Parallel Computation, *Mathematical Programming* **24** 391-405.

[10] Lustig, I. J, An Analysis of an Available Set of Linear Programming Test Problems, *Computers and Operations Research* **16** (1989) 173-184.

[11] Maros, I., A General Phase-I Method in Linear Programming, *European Journal of Operations Research* **23** (1986) 64-77.

[12] Miller, D. L., J. F. Pekny, and G. L .Thompson, Solution of Large Dense Transportation Problems Using a Parallel Primal Algorithm *Operations Research Letters* **9** (1990) 319-324.

[13] Nazareth, J. L, *Computer Solution of Linear Programs*, Oxford University Press, New York (1987).

[14] Nemhauser, G. L., and L. A. Woolsey, *Integer and Combinatorial Optimization*, Wiley, New York (1988).

[15] Rosen, J.B., and R. S. Maier, Parallel Solution of Large-Scale, Block-Angular Linear Programs, Tech. Report. TR-89-56, University of Minnesota (August 1989).

[16] Steenkiste, P, Nectarine - A Nectar Interface, School of Computer Science, Carnegie Mellon University, Pittsburgh PA (November 1990).

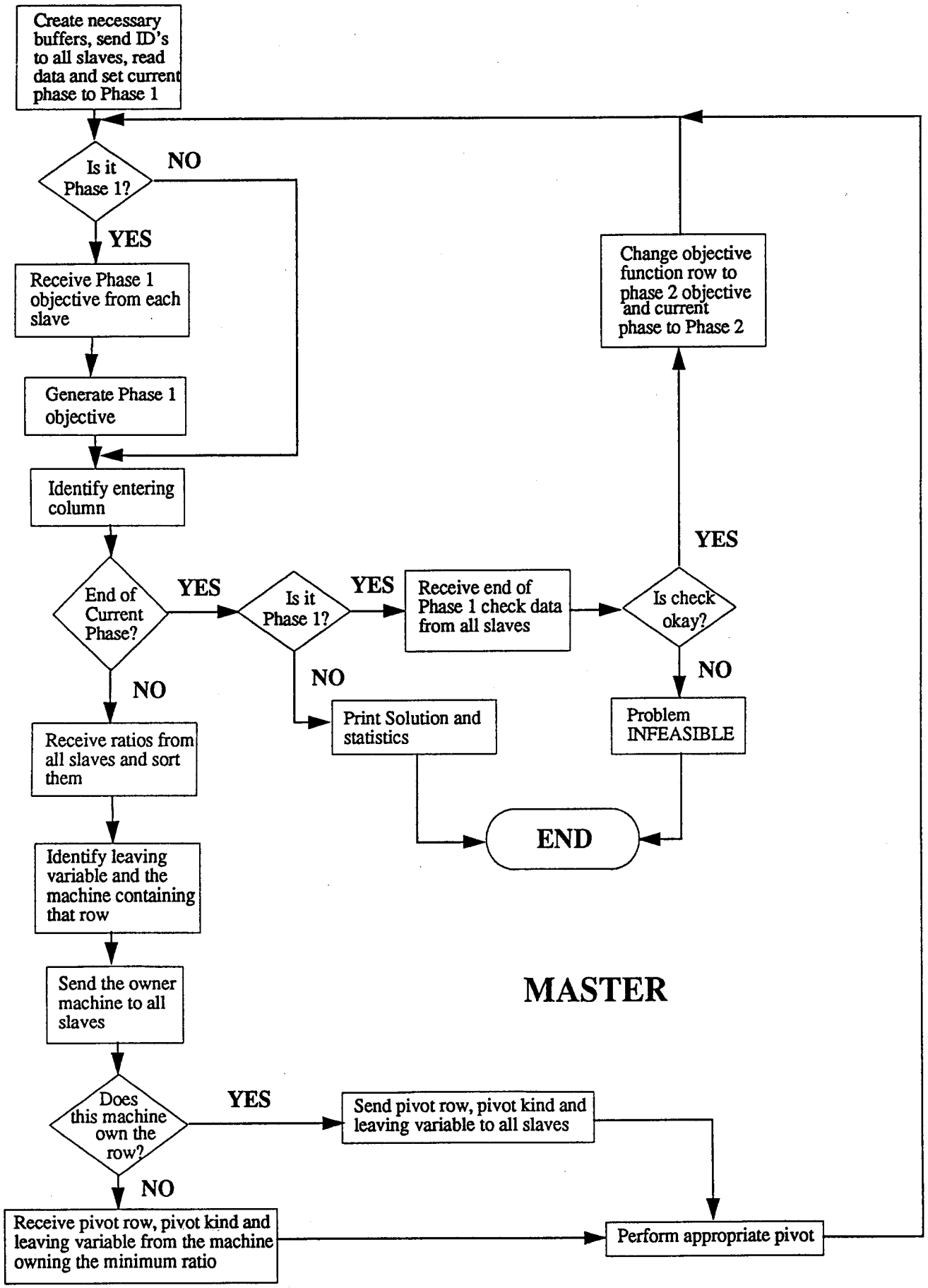

Figure 2: Flow chart depicting the activities of the master machine

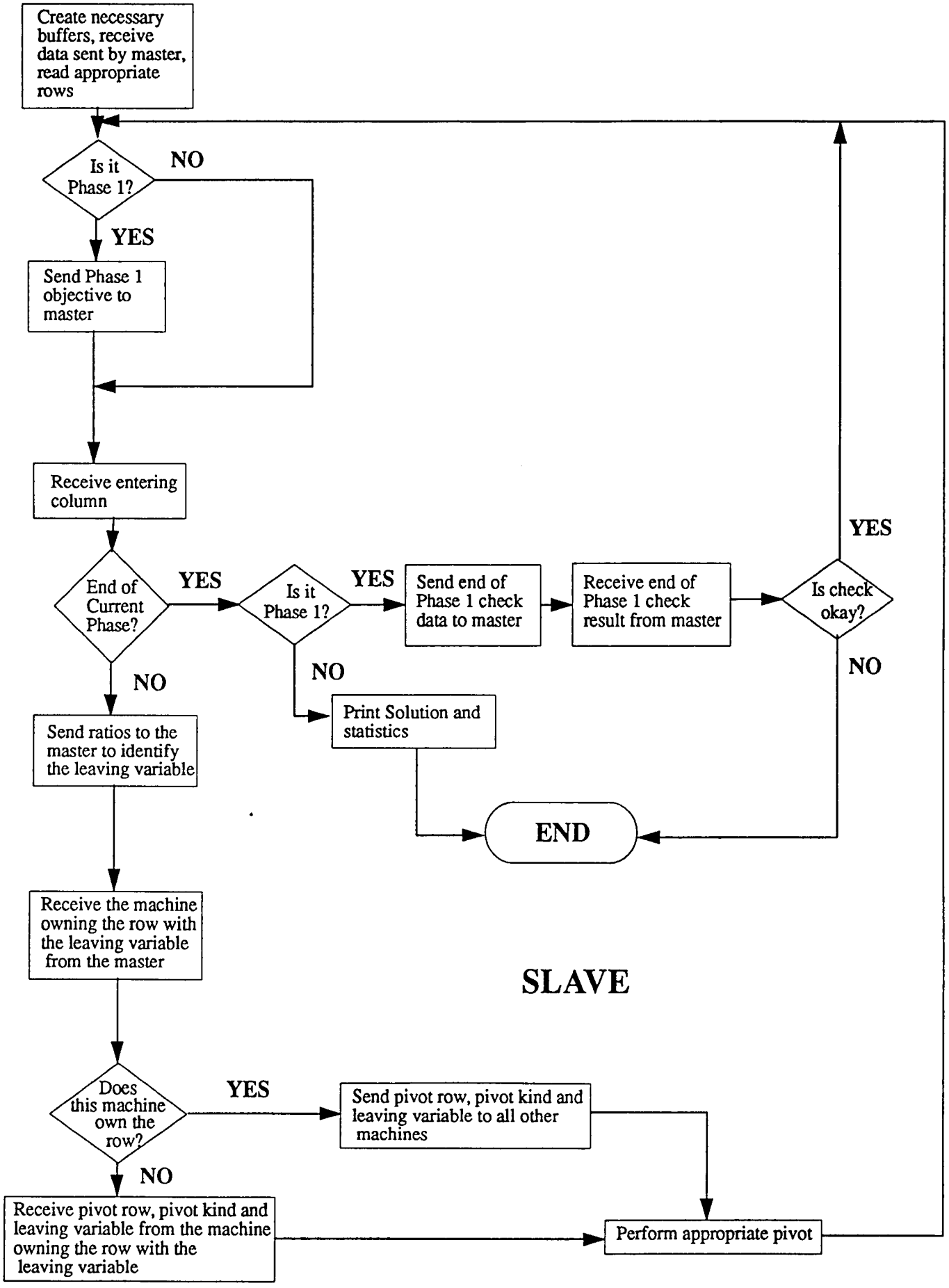

Figure 3: Flow chart depicting the activities of the slave machines

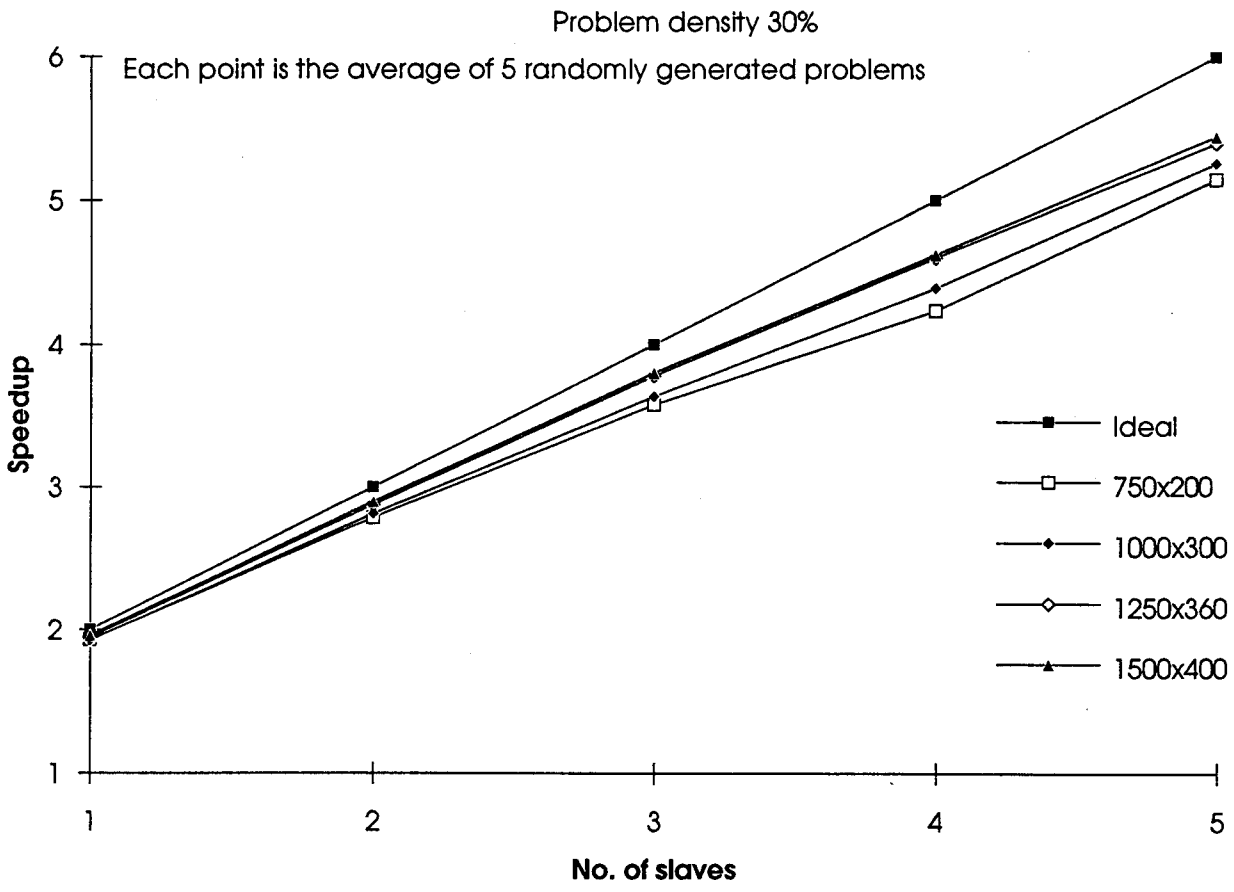

Figure 4: Problem size vs. Speedup with respect to serial version

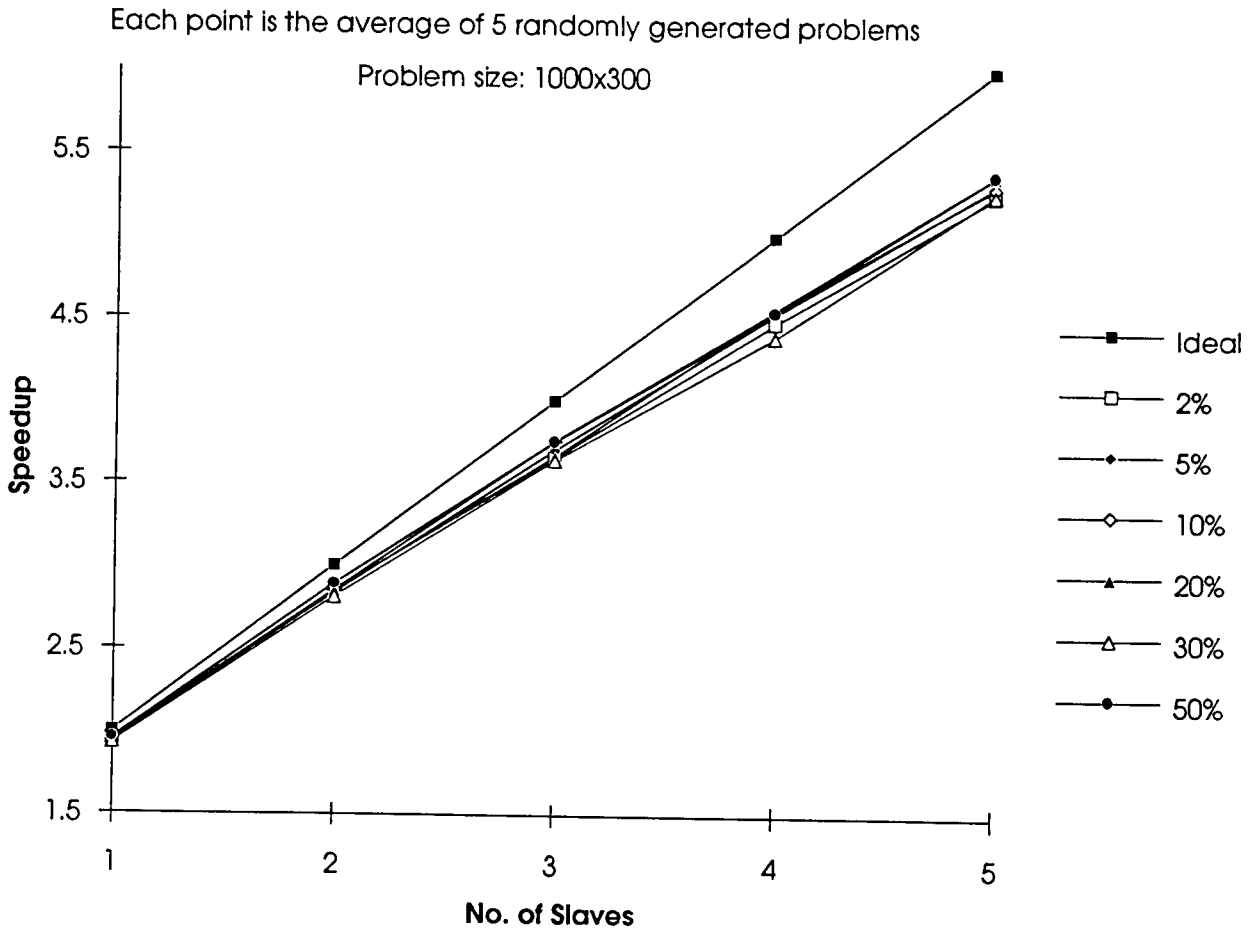

Figure 5: Problem density vs. Speedup with respect to serial version

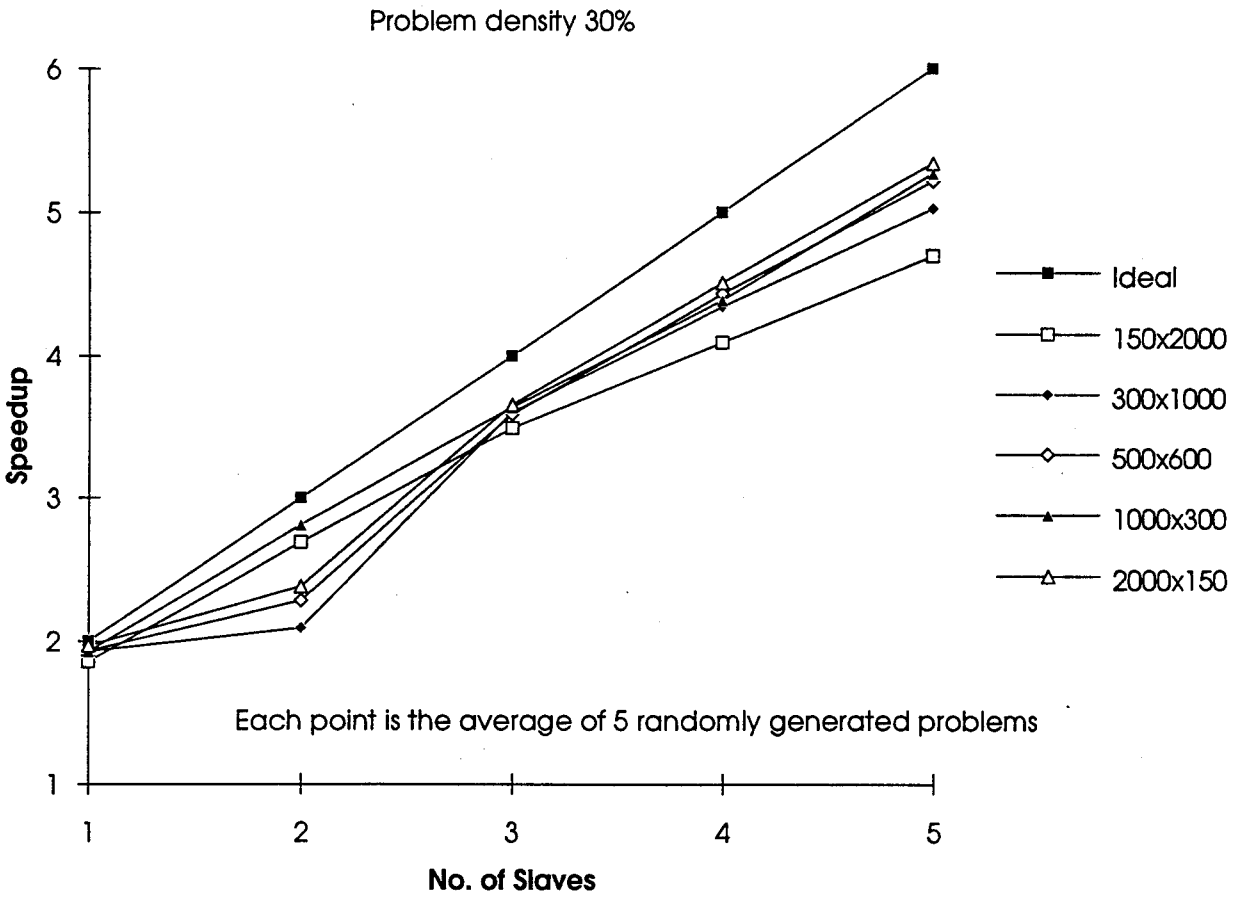

Figure 6: Problem Shape vs. Speedup with respect to serial version

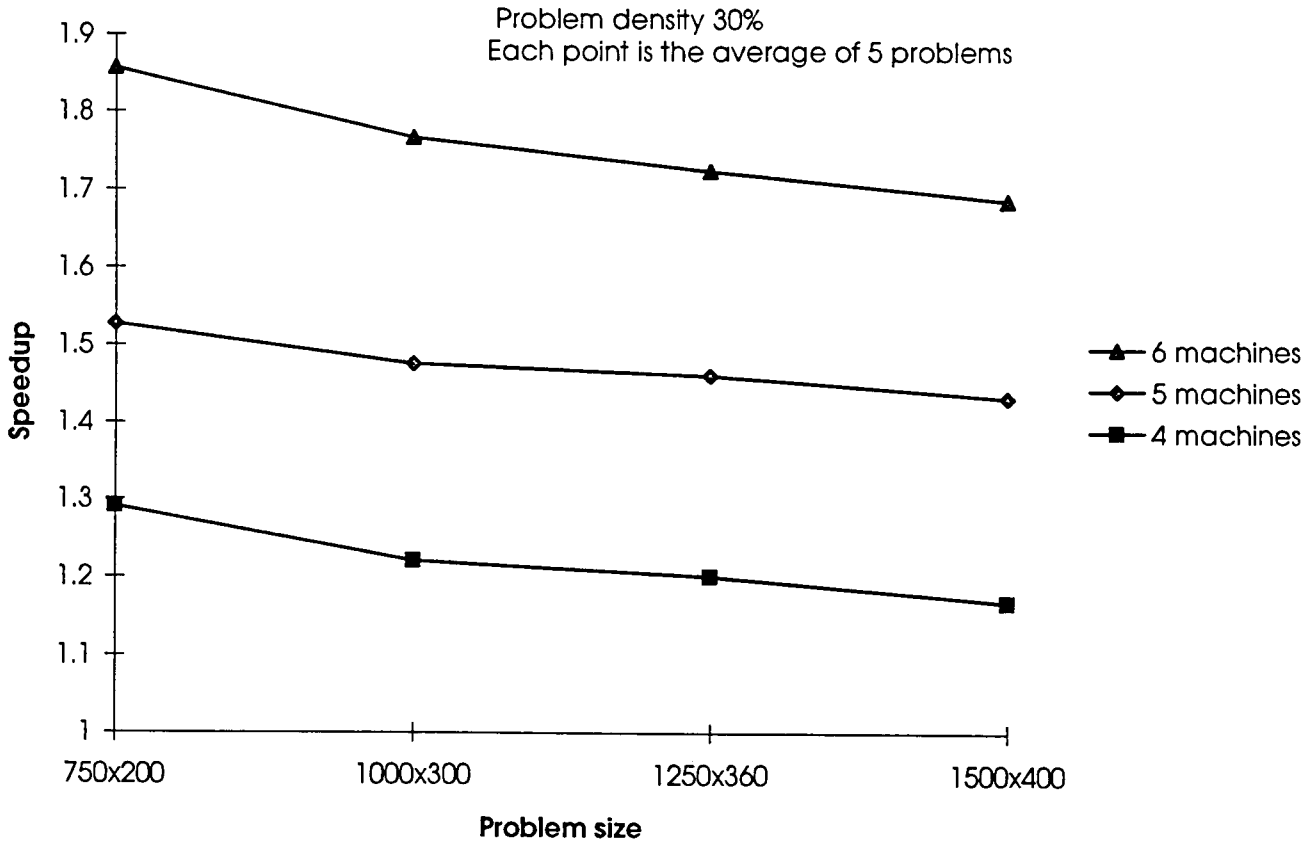

Figure 7: Problem size vs. Speedup with respect to CPLEX 2.1

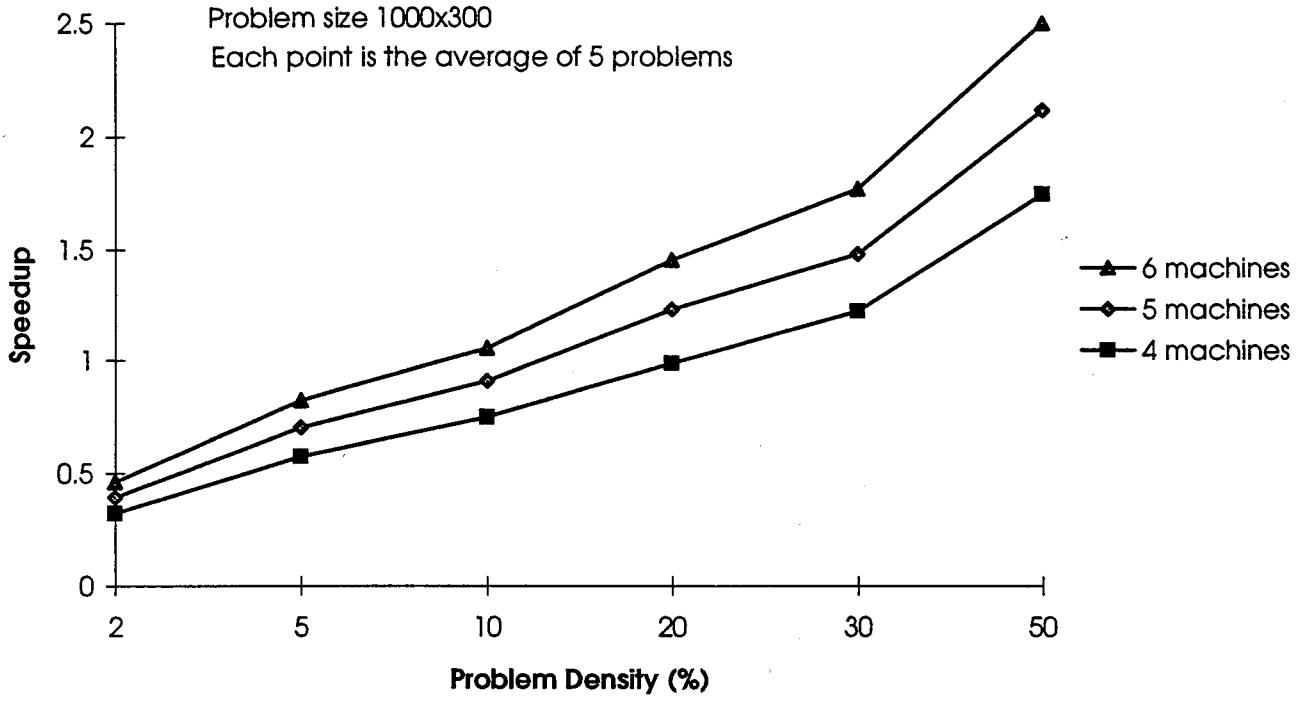

Figure 8: Problem density vs. Speedup with respect to CPLEX 2.1

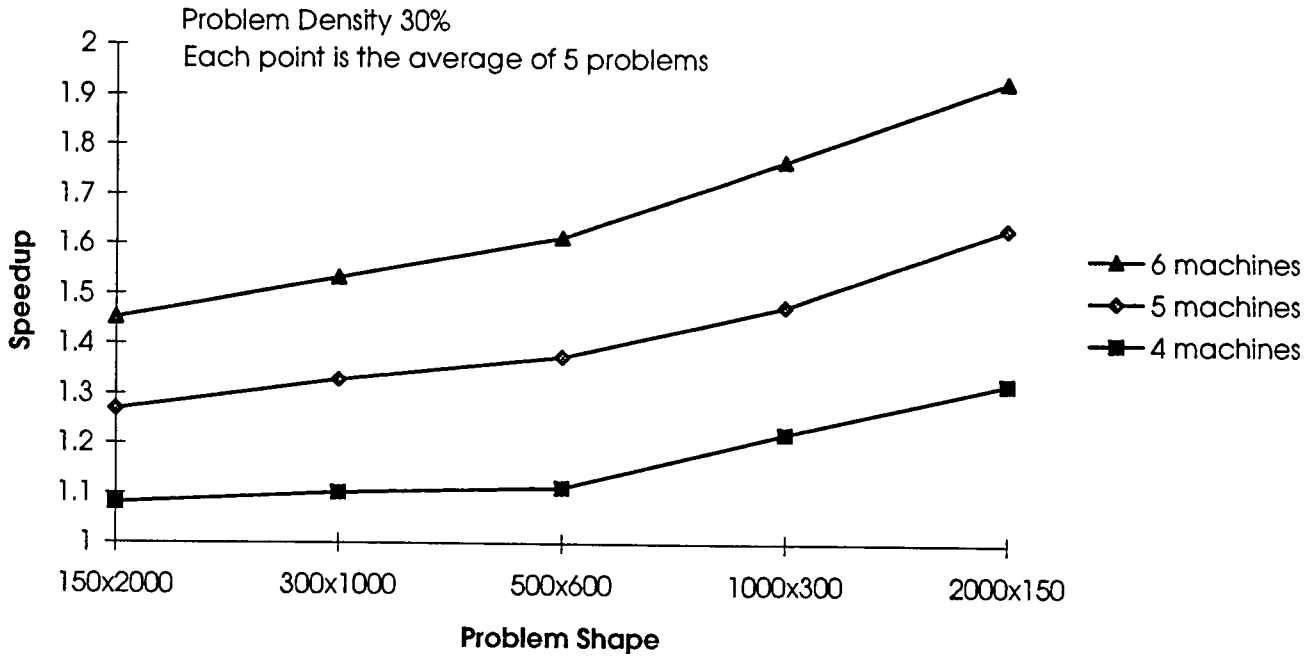

Figure 9: Problem size vs. Speedup with respect to CPLEX 2.1