Parallel Algorithms for Decomposed Linear Programs

Youfeng Wu
T.G. Lewis
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

88-70-5

Parallel Algorithms for Decomposed Linear Programs

Youfeng Wu
Ted G. Lewis
Computer Science Dept.
Oregon State University
Corvalis, OR. 97331
(503)-754-3273
lewis@mist.cs.orst.edu

November 1988

Parallel Algorithms for Decomposed Linear Programs

abstract

New parallel algorithms for solving the decomposed linear programs are developed. Direct parallelization of the sequential algorithm results in very limited performance improvement using multiple processors. By redesigning the algorithm, we achieved more than 2*P times performance improvement over the sequential algorithm, where P is the number of processors used in parallel computation. Furthermore, a particular variation of the sequential algorithm runs more than 2 times faster than the original sequential algorithm. The new parallel algorithm linearly speedups the new sequential algorithm.

1. Introduction

People have been looking for fast Linear Program solvers for a long time because linear programs model many real world applications and solving linear programs is computationally intensive ([DANTZIG-63], [BEN-68], [CHARNES-80], [GROTSCHL-81]). New sequential linear program solvers such as Karmarkar's algorithm ([KARMARKAR-84]) reduce the worst-case time complexity to a polynomial bound. But results of recent computational study ([GILL-85]) cast doubt on Karmarkar's claim that his algorithm will replace the simplex algorithm.

An alternate approach to solving computationally difficult linear programs is to devise parallel solutions that run on fast parallel machines ([WYPIOR-77], [FINKEL-87], [THOMPSON-87], [PANG-87], [CHOI-88]).

In [WU-88a], we parallelized the revised two-phase simplex algorithm with linear performance improvement in terms of number of processors used. Here, we study parallel algorithms for solving decomposed linear programs.

Direct parallelization of the sequential algorithm often results in very limited performance improvement using multiple processors because a sequential algorithm is designed without parallel consideration in mind. When we were parallelizing the decomposed simplex algorithm, we found that, without changing the algorithm itself, the sequential decomposed simplex algorithm can be improved by only half of the number of processors used. But by redesigning the algorithm, we achieved more than 2*P times performance improvement over the sequential algorithm, where P is the number of processors used in parallel computation. Furthermore, a particular variation of the sequential algorithm runs more than 2 times faster than the original sequential algorithm. The new parallel algorithm linearly speedups the new sequential algorithm.

2. Background

A Linear Program (LP) is a system that finds vector x which

$$\begin{array}{ll} \text{minimizes} & z = \mathbf{c}^T x, \\ \text{subject to} & Ax = b,\ x \geq 0, \end{array}$$

where A is an m by n matrix (n > m), **c** is an n element cost vector, **b** is a vector of length m, and **x** is an unknown vector of length n. The superscript ^{T} denotes vector transposition. The equation $Ax = b$ stands for m constraints on the unknowns. An example of an LP is:

$$\begin{array}{ll} \text{Find} & (x_1, x_2, x_3) \text{ that} \\ \text{minimizes} & z = 2x_1 + x_2 + x_3 \\ \text{subject to} & 2x_1 + x_2 - 3x_3 = 0 \\ & x_1 + x_2 + x_3 = 1 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

For this example, $\mathbf{c}^T = (2, 1, 1)$, $\mathbf{b}^T = (0, 1)$, $m = 2$, $n = 3$, and

$$A = \begin{bmatrix} 2 & 1 & -3 \\ 1 & 1 & 1 \end{bmatrix}$$

Geometrically, the constraints $Ax = b$ and $x \geq 0$ define a convex polyhedron of dimension m in an n dimensional space. The polyhedron of the above LP is shown in Figure 1.

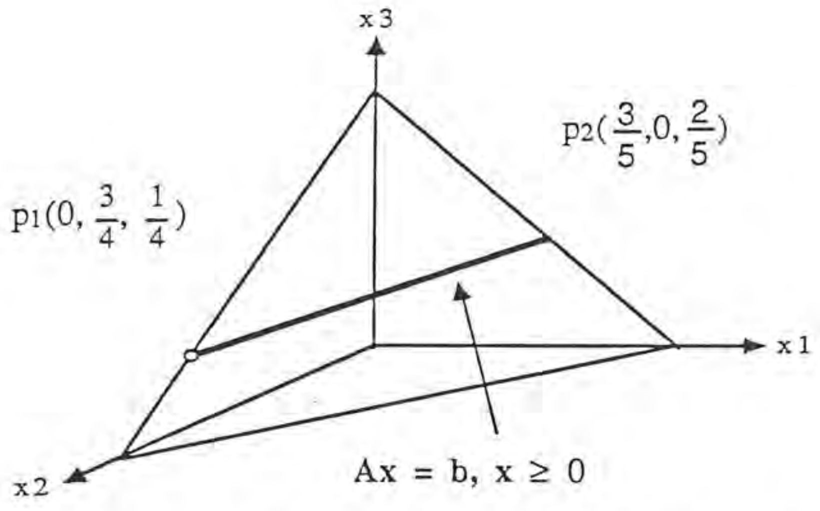

x3
$p_2(\frac{3}{5}, 0, \frac{2}{5})$
$p_1(0, \frac{3}{4}, \frac{1}{4})$
x1
x2
$Ax = b, x \geq 0$

Figure 1. Convex Polyhedron in 3-space Showing Region of Feasible Solutions to Sample Problem.

The end points p_1 and p_2 in Figure 1 are the extreme points of the system. An extreme point is a solution that has no more than m non-zero components. It can be shown that if an LP has a minimal solution, then one of its extreme points must be a minimal solution. In the above example, p_1 is the minimal solution.

The simplex algorithm [DANTZIG-63] solves an LP by starting from an extreme point and repeatedly going to the next adjacent extreme point that decreases the z value, until it goes to an extreme point where the z value can not be further decreased.

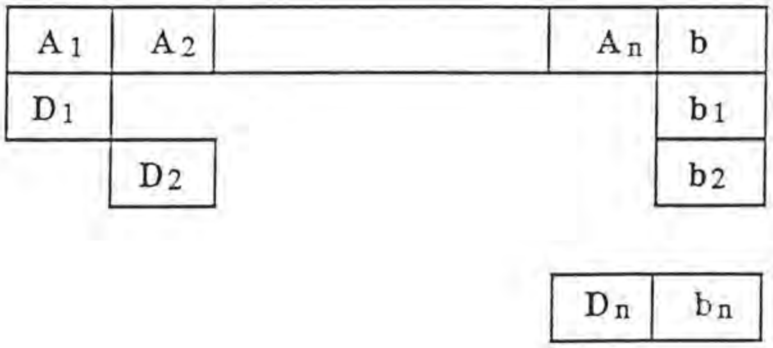


Figure 2. Pattern of a Decomposed Linear Program.

The decomposed linear programs are a special class of LPs in which the coefficient array A contains all zeros except in the first few rows and along diagonal blocks according to the pattern shown in Figure 2, where for j=1..n, A_j is an m by n_j matrix; D_j is an m_j by n_j matrix; b_j is an m_j vector; and b is an m vector.

An example of a decomposed LP is:

Find (x_1, x_2, x_3, x_4) that

$$\begin{array}{ll} \text{minimizes} & z = 2x_1 + x_2 + x_3 + 5x_4 \\ \text{subject to} & 2x_1 + x_2 - 3x_3 + x_4 = 0 \\ & x_1 + x_2 = 1 \\ & 3x_1 - x_2 = 0 \\ & x_3 + 5x_4 = 1 \end{array}$$

In this example,

$$A_1 = (2, 1), \qquad A_2 = (-3, 1)$$

$$D_1 = \begin{bmatrix} 1 & 1 \\ 3 & -1 \end{bmatrix}, \qquad D_2 = (1, 5)$$

$$b = (0),\ b_1 = (1, 0)_T\ b_2 = (1).$$

The decomposition principle of Dantzig and Wolfe [DANTZIG-60, 61] is an elegant method for solving decomposed LPs. According to this principle, an input decomposed linear program is treated as the central program, and the diagonal blocks are treated as the coefficient matrices of sublinear programs. Each central iteration first determines the solutions of the sublinear programs and then uses the solutions to determine its own pivot operation. Although there are many sequential implementations of the principle ([ADLER-73], [BEALE-65], [KUTCHER-73], [HO-81]), there are very few discussions on the parallelization of the algorithm ([WYPIER-77]).

Performance Evaluation Metrics

There are two ways that a program can be parallelized: 1) implicit parallelization ([KUCK-72, 76, 81, 84]), in which the parallel algorithm is the same as the sequential algorithm except that certain statements of the sequential algorithm are allowed to be executed in parallel; 2) explicit parallelization, in which the parallel algorithm employs a different approach to the problem than the sequential algorithm.

Implicit parallelization is limited because the original program was designed with sequential semantics in mind. Only very few programs can be implicitly parallelized with nearly linear speedup, and in most cases an efficiency of 10% is considered quite satisfactory ([LEE-85]), as stated in [WOLFE-87], "users rarely achieve the peak speed of the machine unless they are willing to rewrite their programs."

Explicit parallelization requires that a programmer redesign the sequential algorithm to exploit parallelism in both the problem and the underlying parallel machine. However, explicit parallelization is a difficult job because the programmer has to think in terms of multiple threads of program execution and take care of many possible interactions among the parallel processes.

For a parallel algorithm (PA) obtained through implicit parallelization from a sequential algorithm (SA), the performance improvement of PA over SA can be measured by the speedup of PA over SA and the goodness of PA can be measured using the efficiency of PA. Formally, if SA takes T_s time units to execute and PA takes $T_p(i)$ time units to execute using i parallel processors, then the speedup of PA over SA is defined as:

$$S_p(i) = T_s/T_p(i)$$

and the efficiency of the parallel program using i processors is defined as:

$$E_p(i) = T_p(1)/(i*T_p(i)).$$

We note that $S_p(i) \leq i$ in the implicit parallelization case. This may not be true in the case of explicit parallelization. Assume SA is parallelized explicitly through algorithm changes to PA. Then PA can be executed sequentially by using only one processor. Clearly, we can rewrite PA as another sequential program (SA') using sequential constructs as if the program is executed by only one processor. Now suppose PA is "obtained" from SA' through implicit parallelization, and we compute the speedup of PA over SA' using i processors. The speedup of PA over SA' will be $\leq i$. However, SA' can run many times faster than SA. If SA' runs F (> 1) times faster than SA, then the speedup of PA over SA can be as high as F*i.

For both implicit and explicit parallelization, $E_p(i) \leq 1$. The efficiency of a parallel program is a variable of the number of processors used and is independent of which sequential program it corresponds to.

3. Computational Procedure for Decomposed Simplex Algorithm

Based on Dantzig-Wolfe's decomposition principle, a decomposed simplex algorithm for solving decomposed linear programs can be described as follows (see [WU-88c]):

Input. A_i, $m*n_i$ matrices and D_i, m_i*n_i matrices, $i = 1, \dots n$; b_0, an m vectors; b_i, m_i vectors, $i = 1, \dots n$; c_i, n_i vectors, $i = 1, \dots n$.

Initialization. Assume e be a vector of all 1's.

The inverse of initial base

$$U = (u_1, u_2, \dots u_{m+n+2}) = B^{-1} = \begin{bmatrix} I_{m+n+1} & 0 \\ -e & 1 \end{bmatrix};$$

the initial base feasible solution

$$s = (s_1, s_2, \dots, s_{m+n+2}) = (b_0, 1, \dots 1, 0, n+\sum_{i=1}^{m} b_{0_i});$$

the central left hand side vector $b = (b_0, 1, \dots 1, 0, 0)$;

the initial subsolutions $ex = (ex_1, ex_2, \dots, ex_{m+n}) = (0, 0, \dots, 0)$

and the corresponding indices of the subproblems that lead to the subsolutions $\mathbf{w} = (w_1, w_2, ..., w_{m+n}) = (0, 0, ... ,0)$, meaning that the initial subsolutions are not solutions of any subproblems (sub-problems range from 1 to n); phase = 1; $q = m+n+2$.

Iteration.

Step 1. If $s_q = 0$ and phase = 1, then set phase = 2, $q = m+n+1$, and redo step 1. If $s_q < 0$ or phase = 2, then

a) calculate $\mathbf{c}_j = (\mathbf{u}_{q,1..m}A_j + \mathbf{u}_{q,m+n+1}\mathbf{c}_j)$, for $j = 1,...,n$.

b) using the two-phase revised simplex algorithm to solve sublinear problems S_j, for $j=1, ..., n$,

$$\begin{array}{lll} S_j: & \text{minimize} & \mathbf{c}_j\mathbf{x}_j, \\ & \text{subject to} & A_j\mathbf{x}_j=\mathbf{b}_j \text{ and } \mathbf{x}_j\geq 0 \end{array}$$

for optimal solutions or extreme homogeneous solutions (if S_j is unbound) $\mathbf{x}_j$, $j = 1, .., n$. If any of the subproblems is infeasible, the original problem is infeasible, stop.

c) If $\mathbf{x}_j$ is an optimal solution of S_j, make $\mathbf{a}_j = (A_j\mathbf{x}_j,0,...,0, 1,0,...,0,\mathbf{c}_j\mathbf{x}_j,0)$, otherwise, make $\mathbf{a}_j = (A_j\mathbf{x}_j,0,...,0, 0,0,...,0,\mathbf{c}_j\mathbf{x}_j,0)$.

d) For $j = 1, ... , n$, calculate $\delta_j = \mathbf{u}_q * \mathbf{a}_j$. If phase = 2 then for $j = 1, ... , n$, calculate $\lambda_j = \mathbf{u}_{m+n+2} * \mathbf{a}_j$ and if $\lambda_j \neq 0$ then set $\delta_j = 0$.

Step 2. Calculate $\delta_k = \min(\delta_j \mid j = 1, ..., n)$. If $\delta_k \geq 0$ and phase = 1, then the original LP is infeasible, stop. If $\delta_k \geq 0$ and phase = 2, then s_q is the optimal solution and $-s_q$ is the minimal value of the original LP, exit. Otherwise, $\mathbf{a}_k$ is the new column to enter the base.

Step 3. Compute $y_i = \mathbf{u}_i * \mathbf{a}_k$, $i = 1, ... , q$.

Step 4. If all $y_i \leq 0$ and phase =1, then the original LP is infeasible, stop. If all $y_i \leq 0$ and phase =2, then the original LP is unbounded, stop. Otherwise, calculate

$$\theta = \frac{s_t}{y_t} = \min_{1 \leq i \leq m+n \ \& \ y_i > 0} \left[\frac{s_i}{y_i} \right]$$

and a_t is the column to be removed from the base.

Step 5. Calculate the new values of the variables in the base solution:

$w_t = k$, $s_k = \theta$

$s_i = s_i - \theta y_i \quad (i \neq k,\ i = 1,..,m+n+2)$

$ex_k = x_k$,

and update U, the inverse of the base:

$u_{ij} = u_{ij} - y_i * u_{tj}/y_t \quad (i \neq t,\ i = 1,..,m+n+2, j=1,..,m+n+2)$

$u_{tj} = u_{tj}/y_t$.

Output. The optimal objective value is $-s_q$, and the optimal feasible solution (may not be basic) is $x = (x_1, \dots, x_n)$, where x_j is obtained from:

$$x_j = \sum_{\overset{n}{\underset{i=1}{\forall}} w_i = j} s_i * ex_i$$

4. Parallelizing Decomposed Simplex Algorithm

The kernel of the procedure is the iteration of the steps 1 to 5. The data dependency graph of the iteration is shown in Figure 3, from which the parallelism can be easily seen as each iteration (the central iteration) requires the solutions from the subLPs, which can be solved independently. In addition, the calculation of $y_1, \dots, y_q$ and $u_1, \dots, u_q$ can be done in parallel.

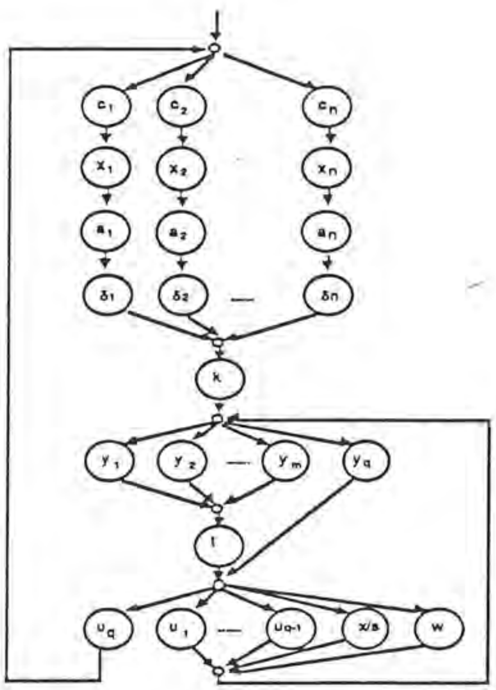
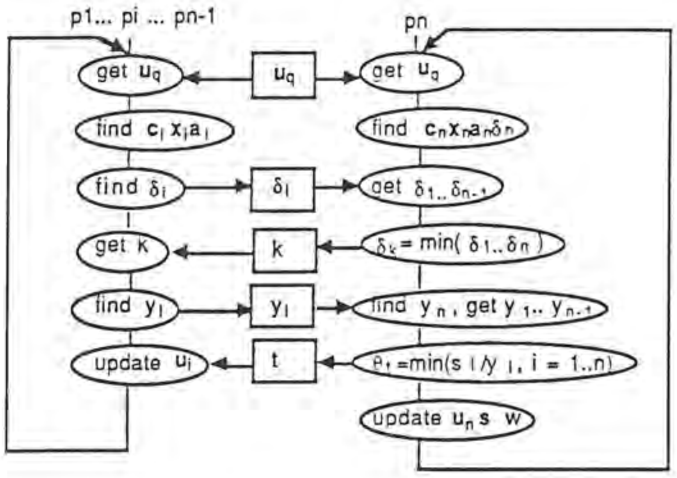


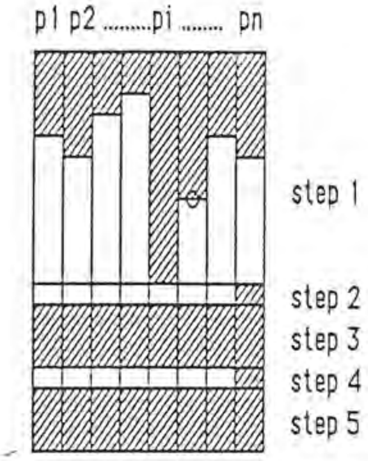
Figure 3. Data Dependency Graph of The Decomposed Simplex Algorithm.

A straightforward parallelization is to invoke the subLP solvers in parallel and continue the central iteration when all of the subLP solvers finish. In this algorithm (call it SF algorithm), n processes, p_1, p_2, ... p_n, are used and pi is assigned to solve the subproblem i in step 1, as in Figure 4 (a). When all of the subproblems find their solutions, the subprocesses send δ_i's to one of the processes (say p_n) and this process determines $\delta_k=\min(\delta_i)$. After k is determined, it is broadcasted to all of the other processes, and the process p_i calculates y_i (i=1,...m). Then, y_i's are sent to p_n. P_n determines $\theta_t=\min(s_i/y_i)$ and broadcasts t to all of the other processes. Finally, the process p_i updates u_i (i=1,...q), and the next iteration begins.

A timing chart of the algorithm is sketched in Figure 4 (b), where p_i represents process i, i=1,...n, and the circle indicates the point in time when the optimal x_k is produced.


p1... pi ... pn-1
pn
get u_q
u_q
get u_q
find $c_i x_i a_i$
find $c_n x_n a_n \delta_n$
find δ_i
δ_i
get $\delta_1 .. \delta_{n-1}$
get k
k
$\delta_k = \min(\delta_1 .. \delta_n)$
find y_i
y_i
find y_n, get $y_1 .. y_{n-1}$
update u_i
t
$\theta_t = \min(s_i / y_i,\ i = 1..n)$
update u_n s w

(a)



(b)

Figure 4. Straightforward Parallelization Algorithm.

The straightforward algorithm seems to exploit the inherent parallelism fairly well. However, step 2 cannot proceed until all of the subproblems finish. As indicated by [KUNG-76], the efficiency of this kind of synchronous algorithm is heavily affected by the structures of the subproblems. A synchronous algorithm performs best when the subproblems are of equal size and take the same amount of time to finish. Even with the assumption that all subproblems have the same number of constraints and same number of variables, the subproblems will take a very different number of iterations to finish, depending on the starting bases and the cost functions. It is even possible for one subproblem to find its solution in one iteration, while another takes exponential number of iterations ([KLEE-76]). Each central iteration has to wait until the slowest subproblem finishes.

The algorithm has been implemented on the Sequent/Balance shared memory machine. We ran the algorithm using 8 processors on randomly generated decomposed LPs of 3 to 20 subproblems. Figure 5. plots the speedup.

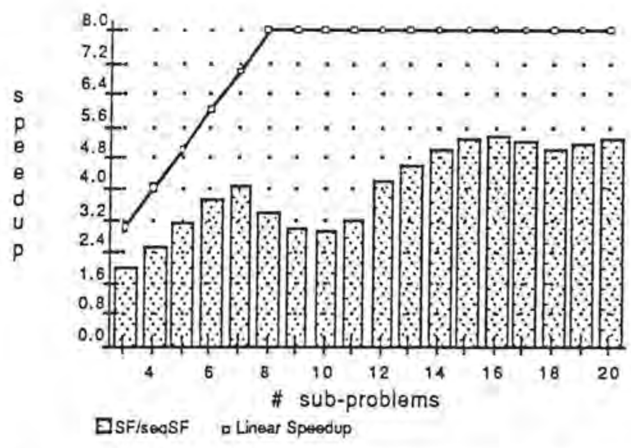


Figure 5. Speedup of Straightforward Algorithm.

From Figure 5, we see that the speedup is much less than the number of processors used. The speedups is at most 5 times using 8 processors (with an efficiency of less than 65%).

To elaborate on this <u>inefficiency</u> further, we cite the experimental data of [LINDBERG-84], which shows that, to solve the assignment problem of d dimensions (2d-1 constraints) using the standard Simplex algorithm, the number of simplex iterations follows the normal distribution with mean $\mu_d = 1.10d^{1.57}$, and standard derivation $\sigma_d = 0.33d^{1.51}$. According to this distribution, an assignment problem of d dimensions needs in the average of $1.10d^{1.57}$ iterations to solve. Assume the number of iterations for N assignment problems of d dimensions are $T_1, T_2, ..., T_N$, respectively. Solving them sequentially requires a total of $T_s = T_1 + T_2 + ... + T_N = N*\mu_d$ iterations, and solving them in parallel using N processors needs $T_p = \max(T_1, T_2, ..., T_N)$ iterations. The efficiency using N processors is:

$$E(N) = \frac{T_s}{N * T_p} = \frac{N * \mu_d}{N * T_p} = \frac{\mu_d}{T_p}$$

To evaluate E(N), instead of determining the mean value of Tp analytically, we used simulation to estimate E(N). The expected efficiency for N = 10, d = 10 is 65.3%. This low efficiency matches our experimental result well.

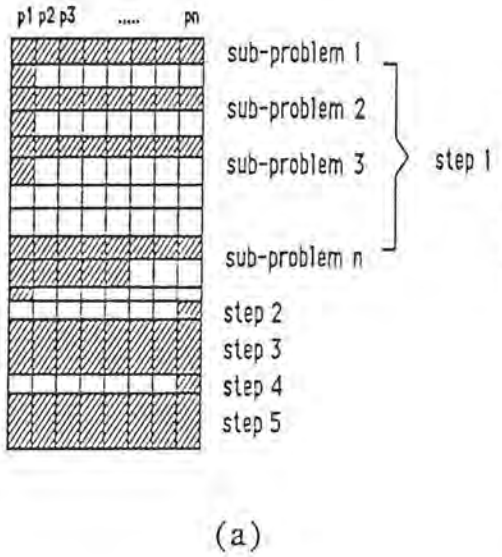
Because the big variance (σ_d) on the number of simplex iterations does not allow all subproblems to find their minimal solutions at the same time, new approaches or algorithm changes are needed to achieve better performance.

Parallelizing Subproblem Solvers.

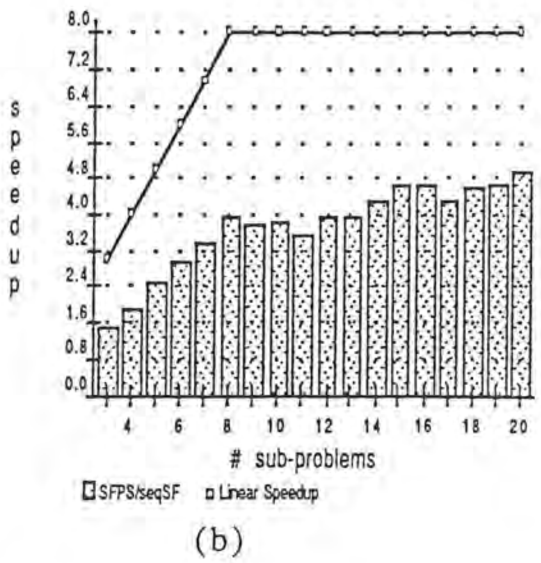
An alternative way to parallelize the decomposed simplex algorithm is to parallelize individual subproblem solvers (see [WU-88a]). In this algorithm, step 1) can be solved as follows:

. Perform step 1 a) in parallel;
. Solve the n subproblems in b) one after the other in sequence and each subproblem is solved by multiple processors in parallel;
. Execute step 1 c) and d) in parallel.

In this algorithm, no subproblem needs to wait for the other subproblems to finish. A timing chart for the algorithm is shown in Figure 6 (a). Figure 6 (b) plots the speedup of the algorithm using 8 processors on randomly generated decomposed LPs of 3 to 20 subproblems.



(a)



(b)

Figure 6. Subsolver Parallelizing Algorithm.

The performance of the algorithm is even worse than the straightforward algorithm. The reasons for the inefficiency are that 1) the subproblems are usually of relatively small size compared to the original problem and we know from [WU-88a] the performance drops when the input data size decreases; 2) in this algorithm, each subiteration needs a fork-join of processes (an invocation of the PAR construct, which costs CPU time), while the straightforward algorithm requires a fork-join only for each central iteration; 3) When the sizes of the subproblems are not the multiple of the number of processors, lots of last round effects are encountered, leaving several processors idle at the end of solving each subproblem. From these we conclude that parallelizing the subsolvers is not an appropriate approach to improve performance.

These two parallelizing approaches have one thing in common, that is, they both extract the parallelism in the sequential algorithm without modification to the algorithm itself. Better performance may result if we adapt the algorithm to parallel execution.

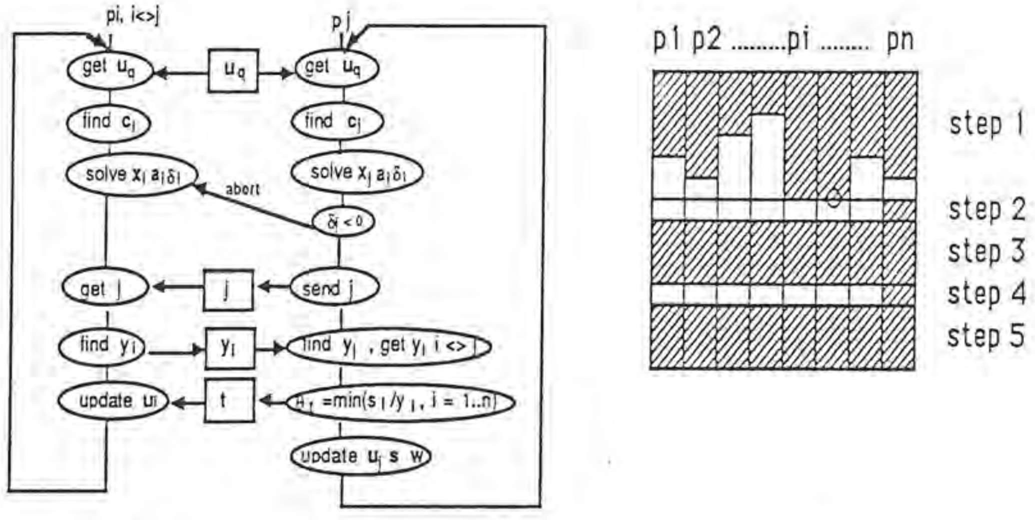
5. Parallel Algorithms for Decomposed Linear Programs

We note that in step 1, finding the best x_k among all of the solutions of the subproblems after waiting for all subproblems to finish is equivalent to moving from the current extreme point to an adjacent extreme point such that the objective function is improved by the greatest amount. Statistics shows that moving to the best adjacent extreme point performs only moderately better than moving to any of the adjacent extreme points which improves the objective function ([DANTZIG-63]). Thus, we can use any solution x_k that makes $\delta_k < 0$. In this way, there is less chance that a fast subproblem waits for a slow subproblem. We have several ways to implement this strategy.

5.1. First Finished First (FFF) Algorithm

Instead of finding the best solution among the subsolutions of all of the subproblems that make $\delta_j < 0$, we use the solution of the first finished subproblem that satisfies the condition.

In this method, every time a subproblem Sj finds its optimal solution x_j, it checks to see whether or not this solution makes $\delta_j = u_q a_j < 0$. When its solution satisfies the condition, it proceeds to step 2 and signals all other subprocesses to stop searching. When step 5 finishes, a new cycle starts. The modified algorithm is illustrated in Figure 7 (a), and the timing chart of the algorithm is sketched is Figure 7 (b) (where the circle indicates when the qualified solution x_k is found).



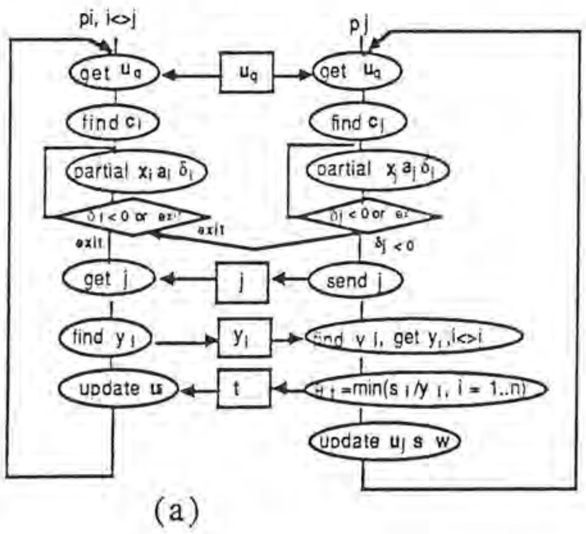
(a) (b)

Figure 7. Parallel FFF Algorithm.

5.2. Tightly Synchronous (TS) Algorithm

We further notice that the minimal solution of one subproblem may not be as good as the non-minimal solutions of the other subproblems. For example, it is possible that a subproblem that takes a very long time to find a minimal solution may have already found a non-minimal solution that is better than the minimal solutions of the other subproblems. When this happens, the FFF algorithm will ignore these good solutions.

In the TS algorithm described here, instead of determining optimal solutions, each subproblem sends its current solution (not necessarily optimal) to the central process after some number of iterations. The central process selects from the n solutions the one that makes $\delta_j = \mathbf{u}_q a_j$ negative if one exists, or else repeatedly invokes the subproblems. If we assume that the subproblems are the same size, then all subproblems will take equal time to finish a single iteration, and this algorithm can synchronize all subproblems after they perform an equal amount of computation. The TS algorithm is shown in Figure 8 (a). The timing of the algorithm can be sketched as in Figure 8 (b) (where the circle indicates when the qualified solution $\mathbf{x}_k$ is found).



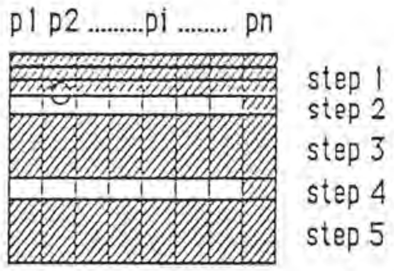


Figure 8. Parallel TS Algorithm.

5.3. Lookahead First Finish First (FFFL) Algorithm

The consideration that leads to the lookahead algorithms is the observation that the optimal solution of a subproblem constantly changes as the objective vector of the subLPs changes. A non-minimal solution for one objective vector may be optimal or very close to optimal for another objective vector. This suggests that, during steps 2 to 5 of the central iteration, the subproblems should not wait for the next iteration to start. Instead, the subprocesses can keep optimizing on the current objective functions, and when the next iteration starts, the solutions of some subproblems may already be good enough to satisfy the condition $\delta_j < 0$. So step 2 of the next iteration of the central problem can start immediately.

The lookahead algorithms use n processes (the subprocesses) for the n subproblems. An n+1'th process (the central process) controls the central problem. These processes all share a global value $\mathbf{u}_q$, and each subprocess S_j maintains the values δ_j and x_j which can be are accessed by the central process.

Each subprocess calculates its own objective vector from the global $\mathbf{u}_q$ that is updated by the central process. When it finds a solution with $\delta_j < 0$, it recommends the solution (x_j and δ_j) to the central process. The central process periodically checks whether or not any of the δ_j is negative, and once it finds such a δ_j, it starts steps 2 to 5. Meanwhile, the subproblems continue optimizing on the current objective vectors. When the central process finishes updating $\mathbf{u}_q$, the subproblems update their objective vectors.

The Lookahead FFF algorithm is based on the FFF algorithm. When a subproblem finds its optimal solution that makes $\delta_j < 0$, it will not signal the other subproblems to stop, so the other subproblems continue running as the central problem proceeds. The algorithm is shown in Figure 9 (a). The timing of the algorithm is shown in Figure 9 (b).

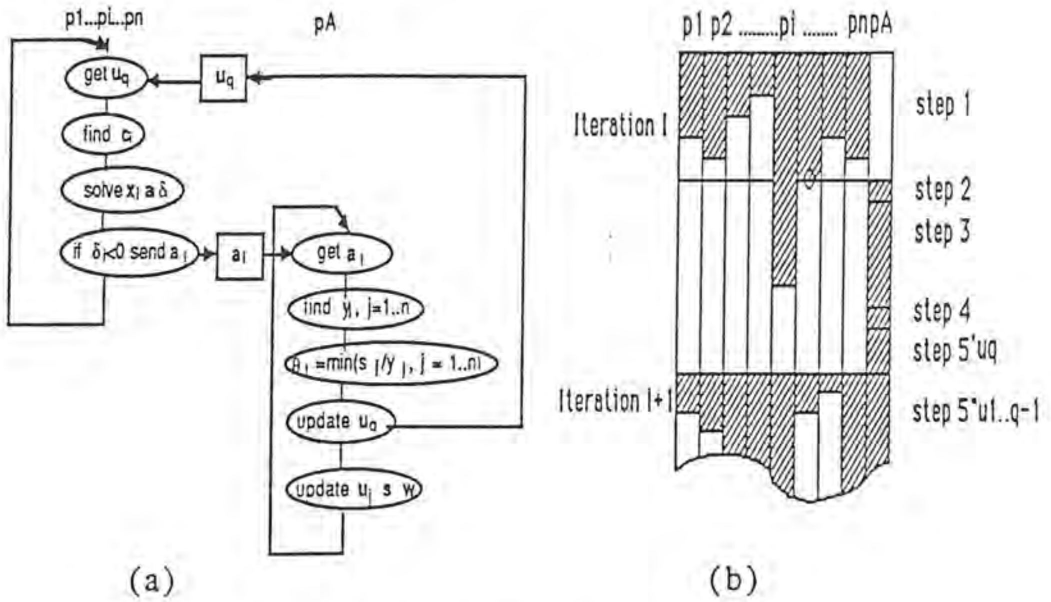
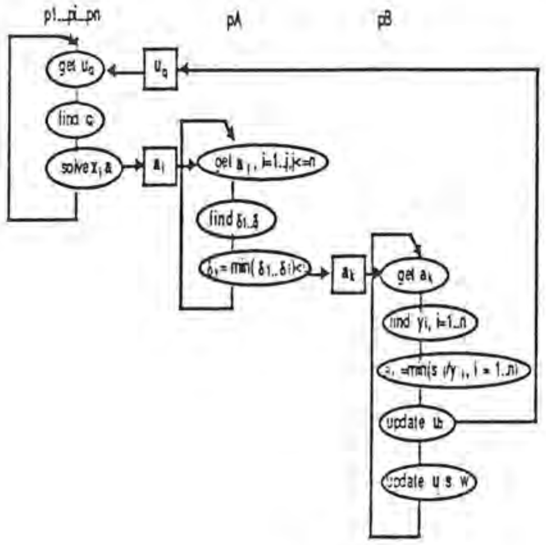


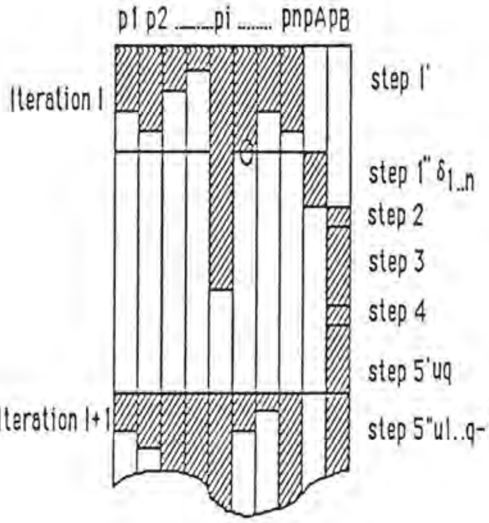
Figure 9. Lookahead FFF Algorithm.

5.4. Wypior's Approach

Wypior's approach is a variation of the lookahead FFF algorithm. In this approach, n processes, p1, p2 ,..., pn, are assigned to solve the n subproblems, and these processes continually perform steps 1 a) to c), send the result a_j's to another process p_A, and wait for the u_q before performing the next iteration. Process p_A continually collects a_j's and performs steps 1 d) and 2, and if it finds an a_k that makes $\delta_k < 0$, it sends the a_k to yet another process p_B. Process p_B continually asks for a_k from p_A and performs steps 3, 4, and 5. This situation can be described in Figure 10 (a). The timing of the algorithm is shown in Figure 10 (b).



(a)



(b)

Figure 10. Wypior's Algorithm.

The reason for using process P_A is that passing the solutions from p_i, i=1,...,n, to P_B may take some non-trivial time, especially in a message passing system. Using P_A, the collection of the solutions from p_1,...,p_n can be overlapped with the update of **u** done by P_B.

One drawback of this method is that P_A can be the bottleneck of the algorithm, as all of δ_j's are calculated in P_A sequentially without overlapping with any other processes and each δ_j needs an inner product operation. An improvement is to let each subproblem solver calculate δ_j and send δ_j and $\mathbf{a}_j$ to P_A. In this way, P_A only needs to determine $\delta_k = \min(\delta_j)$ and send $\mathbf{a}_k$ to P_B. Even with this modification, we see that among all of the vector $\mathbf{a}_j$'s sent to P_A, only $\mathbf{a}_k$ is used in later computation. We can further modify the algorithm so that each process p_j only sends δ_j to P_A, and when P_A finds δ_k, it sends k to process p_k, asking for the vector $\mathbf{a}_k$; and p_k directly sends $\mathbf{a}_k$ to P_B. With these two modifications, only very few data are shared among p_j's and P_A and P_B. With the decreased data sharing, however, the consideration that leads to the necessity of P_A is no longer valid, and we can merge P_A to P_B, thus resulting in the lookahead FFF algorithm. For this reason, we consider Wypior's algorithm less efficient than the lookahead FFF algorithm and will not evaluate Wypior's algorithm further.

5.5. Lookahead Tightly Synchronous (TSL) Algorithm

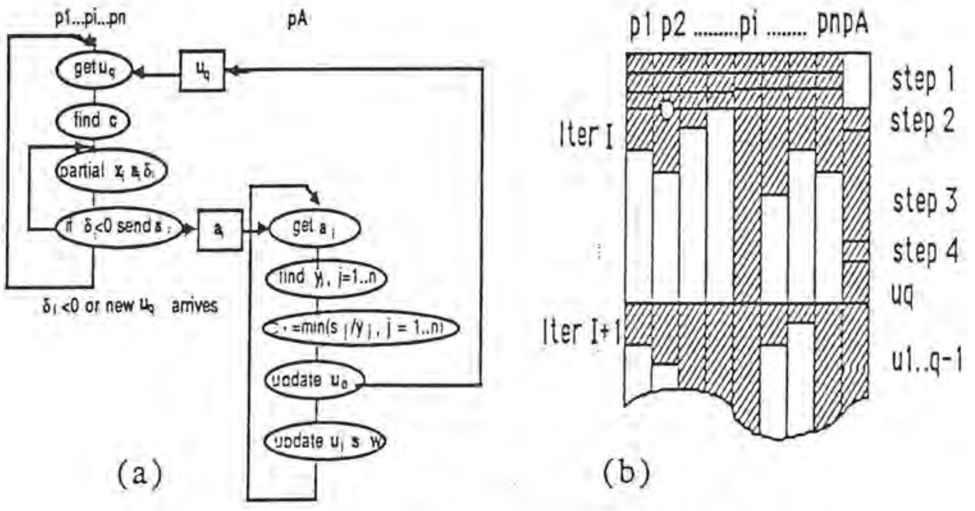


Figure 11. Lookahead TS Algorithm.

Another lookahead algorithm is based on the tightly synchronized algorithm. In this algorithm, each subproblem checks to see whether or not its current solution makes $\delta_j < 0$ after a constant number of iterations. When one such solution is found, the subproblems continue running as the central problem proceeds. The algorithm is shown in Figure 11 (a). The timing of the algorithm is shown in Figure 11 (b).

5.7. Performance Comparison

Experiment is performd on the parallel algorithms and the sequential algorithm on the Sequent/Balance machine using 8 processors. The input LPs consist of n=3 to 20 subproblems, each with m=2+2n/3 constraints and v=2m variables. For each fixed triple (n,m,v), 5 different cases are run and averaged. Figure 12. plots the speedup of the parallel algorithms over the sequential algorithm.

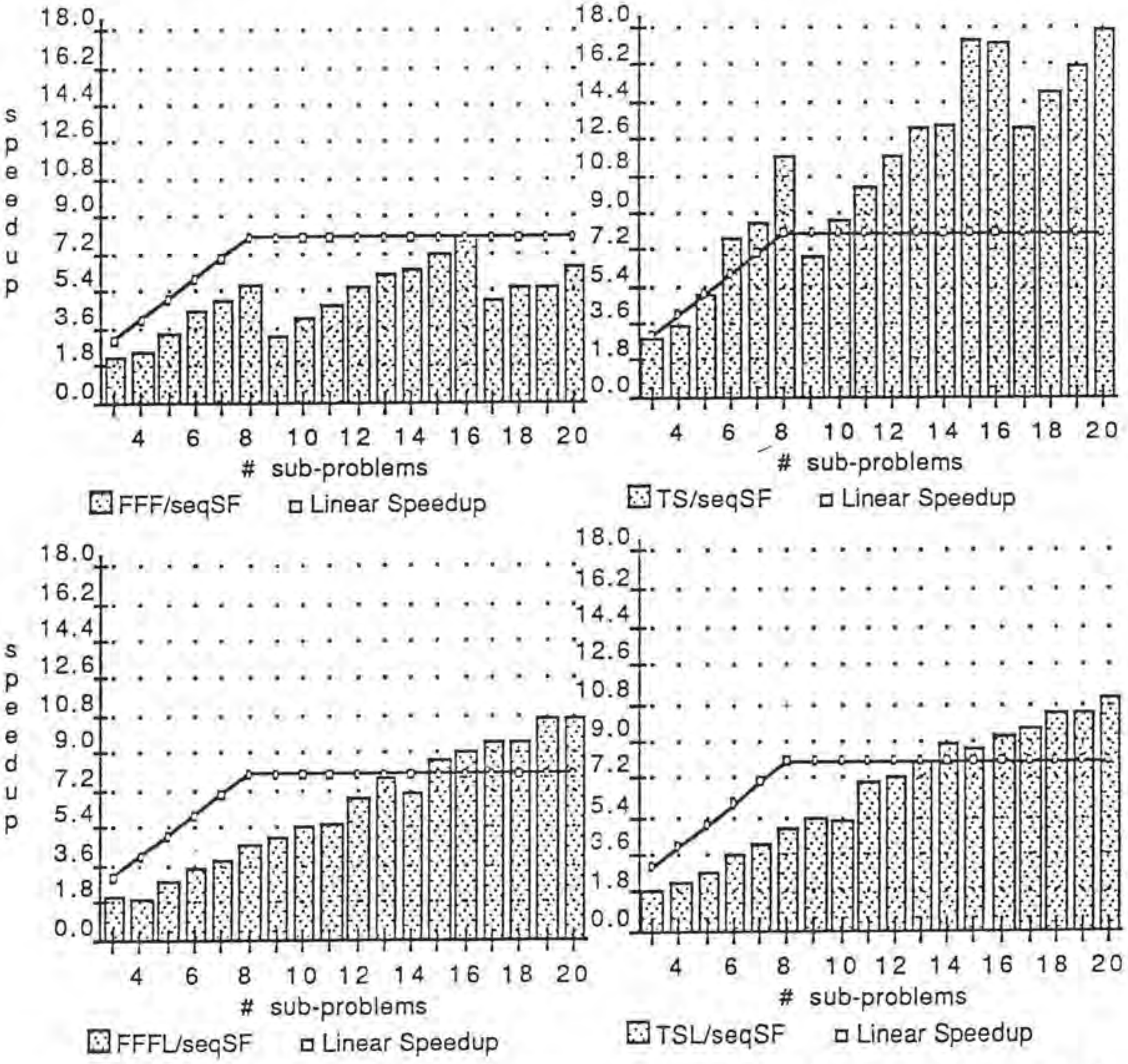
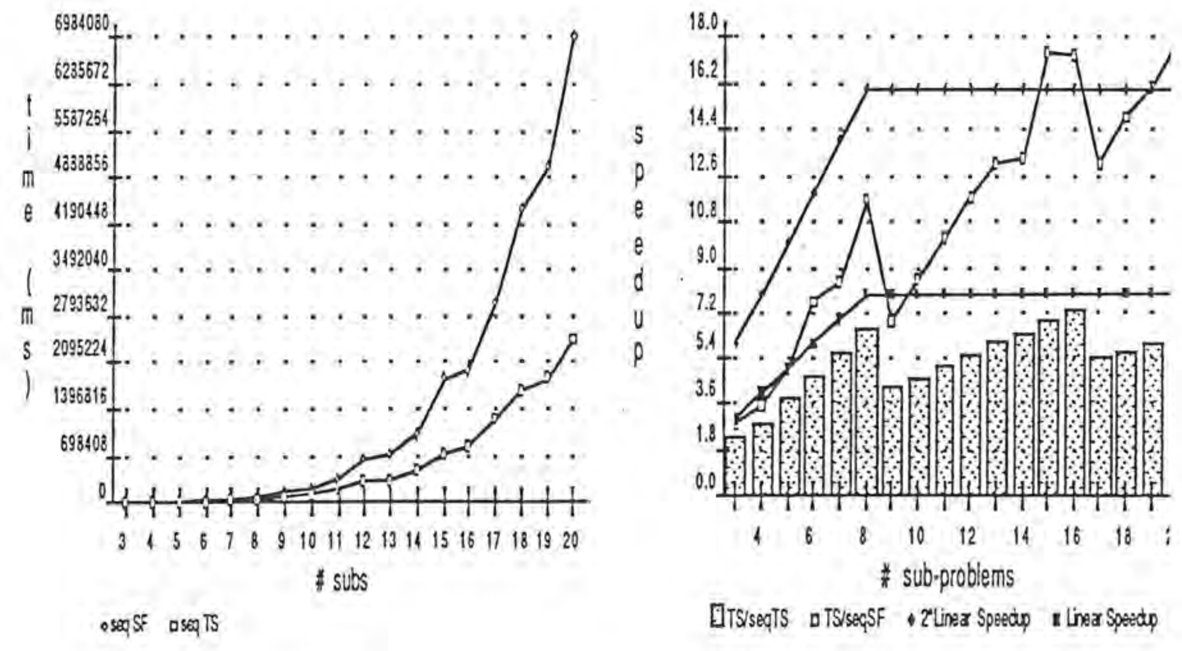


Figure 12. Speedup of Parallel Algorithms Over the Sequential Algorithm.

The parallel TS algorithm has a peak speedup of more than twice the number of processors used (see Figure 12). The parallel FFF algorithm has a speedup a little less than the linear speedup. The Lookahead algorithms (FFFL and TSL) have nearly linear speedup in the number of processors used. All of the algorithms here perform much better than the parallel SF algorithm, which has only half of linear speedup.

5.8. Fast Sequential Algorithm

In the above, we showed that the parallel TS algorithm achieves more than 2*P speedup over the sequential algorithm. This implies that the TS criterion can reduce the execution time of the sequential algorithm by half as well. In order to see whether or not this is the case, we implemented a particular version of the sequential algorithm which uses the TS criterion. Figure 13 (a) plots the execution times of the two sequential algorithms, and Figure 13 (b) plots the speedup of the TS algorithm over the two sequential algorithms.



(a) (b)

Figure 13. Comparison of SF/TS Sequential Algorithms.

From Figure 13 (a), the sequential algorithm using TS criterion runs twice as fast as the normal sequential algorithm. Figure 13 (b) shows that the parallel TS algorithm has nearly linear speedup over the sequential TS algorithm and nearly twice the linear speedup over the sequential SF algorithm.

5.9. Performance vs. Number of Processors

In order to observe the behavior of the parallel TS algorithm when the number of processors changes, we run the TS parallel algorithm using 3 to 8 processors. The 5 input LPs are fixed to have 16 subproblems, each with 12 constraints and 24 variables. Figure 14 plots the corresponding speedup.

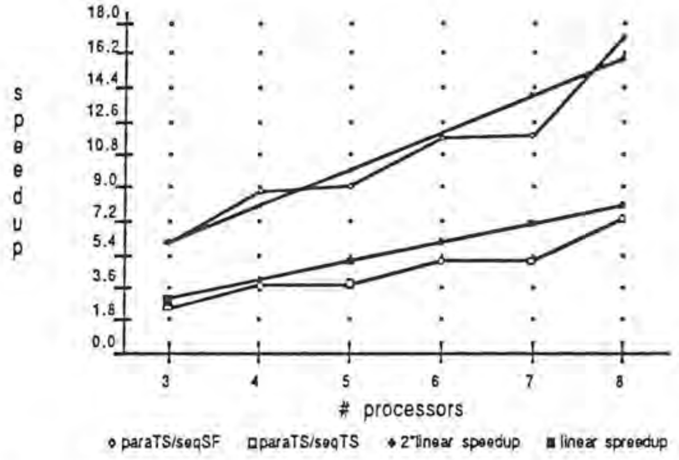


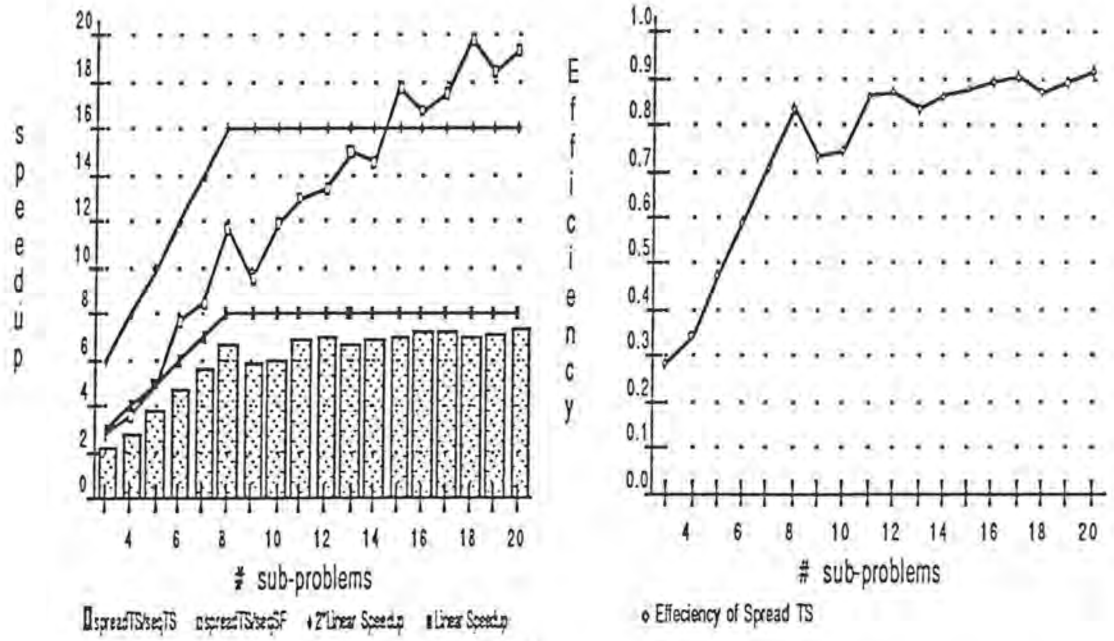
Figure 14. Speedup of Parallel TS Over Sequential TS and SF When Number of Processors Changes From Three to Eight.

From Figure 14, we see that, as the number of processors increases from 3 to 8, the parallel TS algorithm has nearly linear speedup over the sequential TS algorithm, and the parallel TS algorithm has improved the performance of the normal sequential algorithm by twice the linear speedup.

5.10. Removing Last Round Effects

In Figure 13 (b), a drop-off in speedup occurs when the number of subproblems goes from P to P + 1, where P is the number of processors used. This because when the N subproblems are executed by P processors in parallel, they are solved in $\lceil N/P \rceil$ rounds, and in the last round only N MOD P subproblems are solved by N MOD P processors and the remaining processors are left idle.

We also observe last round effects in Figure 14. For the input LPs of 16 subproblems, the parallel TS algorithm has the best performance when the number of processors used is a divisor of 16. For example, when 4 (a divisor of 16) processors are used, the parallel TS algorithm has a speedup of 3.72 over the sequential TS algorithm. But when the number of processors increases from 4 to 7, the speedup only increases from 3.72 to 5.0. When the number of processors changes from 7 to 8 (a divisor of 16), the speedup jumps from 5.0 to 7.28.



(a) (b)

Figure 15. Balanced Performance of Parallel TS Algorithm.

This is the typical processor load balance problem ([COFFMAN-76]). The performance drop-off can be prevented using the Loop

Spreading technique described in [WU-88b]. Figure 15 shows the balanced speedup of the parallel TS algorithm over sequential TS algorithm and the efficiency of the parallel TS algorithm when loop spreading is used.

Comparing the results in Figure 15 (a) to those in Figure 13 (b), we see that the performance of the parallel TS algorithm is quite stable, showing an efficiency of around 90%, without drop-off when the number of subproblems changes.

Figure 16 shows the balanced speedup of the parallel TS algorithm with loop spreading over sequential SF and TS algorithms when the number of processors changes from 3 to 8. From Figure 16 we see that the performance of the parallel TS algorithm is always more than two times the linear speedup when compared to the sequential algorithm and is very close to linear speedup over the fast sequential algorithm.

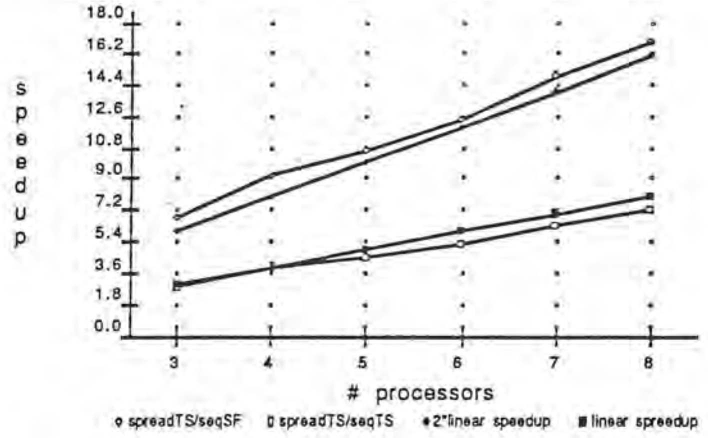
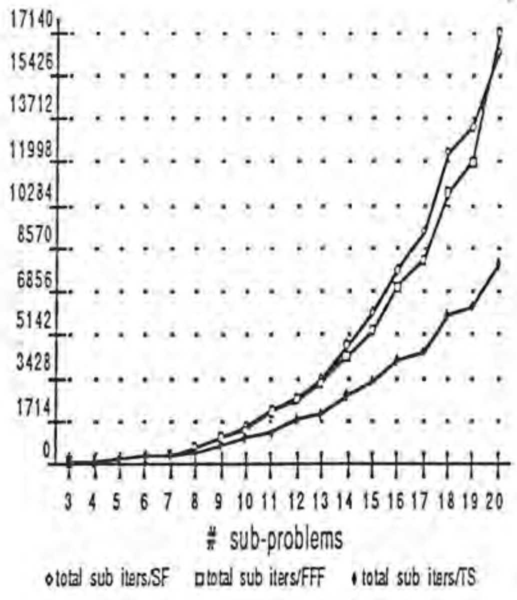


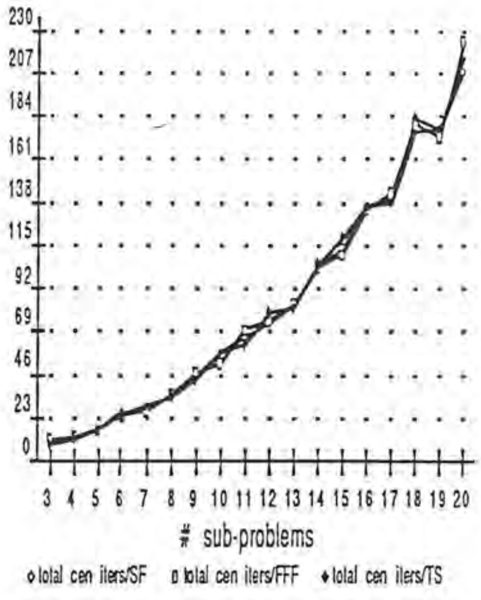
Figure 16. Speedup of Spread TS over sequential TS and SF When Number of Processors Changes From Three to Eight.

5.11. Why TS algorithm is Good

The evolution from SF to FFF and then to the TS algorithm aims at speeding up the step 1 of the decomposed simplex algorithm. We call this optimization **local optimization**. The side effect of this optimization is that the time saving in step 1 might increase the execution time of the step 1 in the next central iteration. Also, it might increase the total number of central iterations (and also the total number of subiterations) required to solve the given LP. We call the minimization of the total number of sub/central iterations the **global optimization.**

To see the relative goodness of the three algorithms in local and global optimization, we collected the number of subiterations in central iterations and the total number of central iterations for each of the algorithms, shown in Figure 17 (a) and Figure 17 (b).





(a) (b)

Figure 17. Numbers of Subiterations and Total Numbers of Central Iterations.

From Figure 17 (a), we see that the TS algorithm has only about half of the total number subiterations used by the SF algorithm, and the SF and FFF algorithms have similar total numbers of subiterations. From Figure 17 (b), we further see that all three algorithms have similar numbers of central iterations. It is clear that algorithm TS is not only the best in local optimization but also the best in global optimization. This conforms our experimental results.

6. Conclusions

Direct parallelization of the sequential algorithm yields very limited performance improvement using multiple processors. For example, without changing the algorithm itself, the performance of the sequential decomposed simplex algorithm can be improved by only half the number of processors used.

We discovered four new ways to parallelize the decomposed simplex algorithm. The parallel TS algorithm can achieve more than 2*P times performance improvement over the sequential algorithm using P processors. Furthermore, sequential execution of the TS algorithm runs more than 2 times faster than the original sequential algorithm.

7. References

[ALDER-73] Alder, I. and A. ülkücü, "On the number of iterations in Dantzig-Wolfe Decomposition." in: D.M. Himmelblan, ed., Decomposition of Large Scale Problems. (North-Holland, Amsterdan, 1973) pp 181-187.

[BEALE-65] Beale, E., P. Huges, and R. Small, "Experiences in Using a Decomposition Program," Computer Journal 8 (1965) 13-15.

[COURTOIS-71] Courtois, P.J., F. Heymans, and D.L. Parnas, "A Concurrent Control With Readers and Writers," CACM, Vol. 14, No. 10, October 1971, pp667-668.

[DANTZIG-60] Dantzig, G. B., and P. Wolfe, "The Decomposition Principle for Linear Programs," Operations Research 8, 1960, pp. 101-111.

[DANTZIG-61] Dantzig, G. B., and P. Wolfe, "The Decomposition Algorithm for Linear Programs," Econometrica, 29, 1961, pp. 767-778.

[DANTZIG-63] Dantzig, G. B., Linear Programming and Extensions, Princeton University Press, Princeton, NJ (1963).

[GILL-85] Gill, P., W. Murray, M. Saunders, J. Tomlin and M. Wright, "A Note on Interior-point Methods for Linear Programming," MPS Committee on Algorithms Newsletter 13, 13-18 (1985).

[FINKEL-87] Finkel, Raphael A., "Large-grain Parallelism -- Three Case Studies," The Characteristics of Parallel Algorithms, Leah H. Jamieson (ed), The MIT Press, 1987.

[HO-78] Ho, J.K., "Implementation and Application of a Nested Decomposition Algorithm," in: W.W. White, ed, Computer and Mathematic Programming (National Bureau of Standards, 1978) pp67-76.

[HO-80] Ho, J.K., and E. Loute, "A Comparative Study of Two Methods for Staircase Linear Programs," ACM Trans on Math. Software 6 (1980) 17-30.

[HO-81] Ho, J.K., and E. Loute, "An Advanced Implementation of the Dantzig-Wolfe Decomposition Algorithm for Linear Programming," Mathematical Programming 20 (May 1981) 303 326.

[KARKARMAR-84] Karkarmar, N., "A New Polynomial Time Algorithm for Linear Programming," Proceedings of the 16th Annual ACM Symposium on the Theory of Computing, 302-311 (1984).

[KLEE-76] Klee, V. and G.J. Minty, "How Good is the Simplex Algorithm?", in O. Shisha, ed., Inequalities III (Academic Press, New York, 1972).

[KUCK-72] D. J. Kuck, Y. Muraoka, S. C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resultant Speedup," IEEE Trans Comp. vol. C-21, no. 12, Dec. 1972, pp. 1293-1310.

[KUCK-76] D. J. Kuck, "Parallel Processing of Ordinary Programs," in Advances in Computers, vol. 15, Rubinoff and Yovits, eds, Academic Press, New York, 1976, pp. 119-179.

[KUCK-81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proc. 8th ACM Symp. Principles Programming Languages, Jan. 1981, pp. 207-218.

[KUCK-84] D.J. Kuck, A.H. Sameh, R. Cytron, A.V. Veidenbaum, C.D. Polychronopoulos, G. Lee, T. McDaniel, B.R. Leasure, C. Beckman, J.R.B. Davies, and C.P Kruskal, "The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance," 1984 ICPP, Aug. 1984, pp. 129-138.

[KUNG-76] Kung, H. T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," in Algorithms and Complexity, Academic Press, 1976, pp. 153-200.

[KUNZI-68] Künzi, P. Hans, H.G. Tzschach, and C.A. Zehnder, Numerical Methods of Mathematical Optimization with ALGOL and FORTRAN programs. Academic Press, New York and London, 1968.

[LEE-85] Lee, Gyungho, Clyde P. Kruskal, and David J. Kuck, An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors, IEEE Trans. on Computers, Vol. c-34, No. 10, October 1985.

[LINDBERG-84] Lindberg, P. O. and Snjolfur Olafsson, "On the Length of Simplex Paths: the Assignment Case," Mathematical Programming 30 (1984) 243-260.

[ORCHARD-54] Orchard-Hays, Wm., "Background, Development, and Extensions of the Revised Simplex Method," RAND report (RM) 1433, 1954.

[OSTERHAUG-86] Osterhaug, Anita, Guide to Parallel Programming on Sequent Computer Systems, 1986.

[SYSLO-83] Syslo, Maciej M., Deo, Narsingh, and Kowalik, Janusz S., "Discrete Optimization Algorithms--with PASCAL Programs," Prentice-Hall, 1983.

[THAKKAR-85] Thakkar, S. P. Gifford, and G. Fielland, "Balance: A Shared Memory Multiprocessor System," Proc. Int'l Conf. Supercomputing, Institute for Supercomputing, St. Peterburg. Fla., pp. 93-101.

[WOLFE-87] Wolfe, M.J. and Utpal Banerjee, "Data Dependence for Parallelism Detection," Int'l Journal of Parallel Programming, Vol. 15, No. 2, April, 1987.

[WU-88a] Wu, Youfeng and Ted G. Lewis, "Performance of Parallel Simplex Algorithms On a Shared Memory Machine," Technical Report, Dept. of Computer Science, Oregon State University, 1988.

[WU-88b] Wu, Youfeng and Ted G. Lewis, "Parallel Processor Load Balance Through Loop Spreading," Technical Report, Dept. of Computer Science, Oregon State University, 1988.

[WU-88c] Wu, Youfeng,"Parallel Simplex Algorithms and Loop Spreading," Ph. D. Thesis, Dept. of Computer Science, Oregon State University, 1988.

[WYPIOR-77] Wypior, Peter, "A Parallel Simplex Algorithm," Parallel Computers-Parallel Mathematics, M. Feilmeier (ed), Proceedings of the IMACS (AICA)-GI, Symposium, March 14-16, 1977, Technical University of Murich. (North-holland) p235-237.