

Solving Linear Programs with
Two Processors

R.E. Bixby

and

Mark Schwabacher

November, 1989

TR89-16

Solving Linear Programs with Two Processors

Robert Bixby Mark Schwabacher

December 1, 1989

Abstract

A linear programming problem can be solved in parallel using two processors. One processor performs iterations of the main algorithm, while the other processor continuously refactorizes the basis matrix. Using this method we have been able to solve linear programs in as little as 73% of the time needed with one processor.

1 Introduction

We wish to solve a linear program (LP) of the form:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & \ell \leq x \leq u \end{array}$$

Here A is an $m \times n$ matrix, c , x , ℓ , and u are $n \times 1$ vectors, and b is an $m \times 1$ vector. This problem has m constraints and n bounded variables. We assume that we are given B , a square submatrix of A , and x^* , an assignment of the x_i 's to their upper and lower bounds, such that the system is feasible. We use the notation c_B and x_B to denote the elements of c and x that correspond to the columns of A which are included in B .

We solve this problem using a state-of-the-art implementation of an algorithm called the Revised Simplex Method. It is an iterative method, in which each iteration can be summarized by the following five steps:

Step 1: Solve $y^T B = cB^T$

Step 2: Choose entering variable x_j such that $y^T A_j < c_j$ and $x_j^* < u_j$, or $y^T A_j > c_j$ and $x_j^* < \ell_j$. If none exists, current x^* is optimal; stop.

Step 3: Solve $Bd = A_j$.

Step 4: Define $x_j(t) = x_j^* + t$, $x_B(t) = x_B^* - td$ (if $yA_j < c_j$)
 $x_j(t) = x_j^* - t$, $x_B(t) = x_B^* + td$ (if $yA_j > c_j$)

if $\ell_j \leq x_j(t) \leq u_j$, $\ell_B \leq x_B(t) \leq u_B$ satisfied for all positive t , problem is unbounded; stop. Otherwise set t at maximum value allowed by constraints. If upper bound on t imposed by $\ell_B \leq x_B(t) \leq u_B$ is stronger than the upper bound imposed by $\ell_j \leq x_j(t) \leq u_j$, then choose the leaving variable x_i such that the upper bound imposed on t by $\ell_i \leq x_i(t) \leq u_i$ alone is as strict as the upper bound imposed by $\ell_B \leq x_B(t) \leq u_B$.

Step 5: Replace x_j^* by $x_j(t)$, x_B^* by $x_B(t)$. If x_j has just switched from one of its bounds to the other, go to step 2. Otherwise replace x_i by x_j , B_i by A_i . (See Chvatal, 1983)

A large portion of the time needed to perform an iteration is the time needed to solve the two systems of linear equations in steps 1 and 3. Solving such systems is an $O(n^3)$ operation, and might normally be carried out as follows: We compute a factorization of the following form: $PBQ = LU$. Here B is the basis matrix, P and Q are permutation matrices, L is a lower-triangular matrix, and U is an upper-triangular matrix. P arises from partial pivoting, and Q arises from the need to preserve sparsity. We use this factorization to solve the systems in steps 1 and 3, above. To save time, we use a product form update of the factorization from iteration to iteration, rather than computing a new factorization at each iteration. We notice that in each iteration k the matrix B_k differs from the corresponding matrix B_{k-1} from the previous iteration in only one column. Hence $B_k = B_{k-1}E_k$, where E_k is the identity matrix with the one of its columns replaced by d . Thus any B_k can be expressed in terms of an initial matrix B_0 by $B_k = B_0E_1E_2\dots E_k$. The

systems $yB_k = cB$ and $B_k d = a$ can then be solved as $yB_0 E_1 \dots E_k = c_B$ and $B_0 E_1 \dots E_k d = a$. We compute a triangular factorization of B_0 of the form $B_0 = LU$. Hence $B_k = LU E_1 E_2 \dots E_k$. Now the system $yB_k = c_B$ may be solved by solving $yLU E_1 \dots E_k = c_B$. $B_k d = a$ can be solved as $LU E_1 \dots E_k d = a$.

The matrices E_1, E_2, \dots, E_k are stored in what we call an eta file. As this file grows larger, the amount of time needed to solve these systems increases, and the numerical difficulties caused by rounding errors are also increased. Hence, after some number of iterations, we re-compute the LU factorization, at which point the eta file can be eliminated, and a new one started. (See Chvatal, 1983)

Using this method, solving the linear systems and occasionally re-computing the factorization still takes a large portion of the time needed to solve a linear program. By running some tests, we found that Cplex (a linear optimizer which is further described below) spends about 20% of its time re-computing the factorization, and about 40% of its time solving the linear systems. This fact led to the following idea: a linear program can be solved using two processors. One processor (the optimizer) performs iterations of the revised simplex method, while the other processor (the factorizer) continuously computes LU factorizations of the basis matrix. Each time a factorization is completed, the optimizer begins to use the new factorization, and the eta file is cleared of all columns, except for the columns that have been generated since the factorizer began to factorize the newly completed factorization.

Note that this method of parallelization depends on the fact that we are using a product-form update as opposed to an LU update. It would obviously not work with a method such as the Bartels-Golub method in which the update affects the LU factorization itself.

The benefits of this method of parallelization are twofold: First, the optimizer no longer has to spend time re-computing the factorization. Second, the average length of the eta file is likely to be shorter, so the optimizer will spend less time computing solutions to the linear systems, and will be less likely to have numerical difficulties.

Note that this method is a very coarse-grain form of parallelism. In some parallel applications, the processors need to communicate with each other after every iteration of a small loop, and the communications overhead can

reduce or even outweigh the benefits of using an extra processor. In this particular method, the two processors need to communicate with each other only once after each factorization. Since a factorization is a time-consuming operation, the communication does not occur very often, and there is little communications overhead.

2 Experimental Results

To test this method, we modified a program called Cplex¹ to run on two processors, using one processor for the optimization, and one for the factorization, as described above. We used a Sequent Symmetry system, running the DYNIX version 3.0.12 operating system (a parallel variant of UNIX), and the Sequent C compiler, with the Sequent Parallel Programming Library. We obtained speed increases on all the problems we tested except for some very small problems in which the overhead exceeded the benefits of parallelism.

The times for some problems are listed in Table 1. The columns labeled "time1" and "iter1" refer to the amount of time (in seconds) and number of iterations it took to compute the optimal solution, running Cplex with one processor on the above mentioned Sequent system. In these runs, Cplex used a refactorization frequency of 20. That is, the factorization was recomputed once every twenty iterations. The table also lists the time (time2), number of iterations (iter2), number of refactorizations performed (refact), and average refactorization frequency (iter2/refact), for the two-processor version, which uses the method we have described above. The last column shows the percentage of the single-processor time used by the two-processor version. That is, a statistic of 50% would imply that the two-processor version ran twice as fast as the single-processor version.

The table shows that using two processors reduces the average refactorization frequency from twenty to about four for most problems. This fact implies that the average length of the eta file is usually a little bit shorter. Note that the average length of the eta file will now be one and one half times the average refactorization frequency, whereas with one processor it was half

¹Cplex is a trademark of Cplex Optimization Inc., the owner of Cplex. The optimization routines in Cplex were written by Robert Bixby.

the refactorization frequency. The reason for this is as follows: With one processor, the length of the eta file varies linearly from zero to the refactorization frequency, so the average is half the refactorization frequency. With two processors, the minimum length of the eta file will be equal to the average refactorization frequency, since this is how many columns will remain in the file after a factorization has been completed. The maximum length of the eta file will be twice the refactorization frequency, since this is the number of columns that will be in the eta file immediately before a refactorization is completed. Hence the average length will be the average of the minimum and the maximum, which is one and one half times the average refactorization frequency. For most of the problems we tested, the average length of the eta file is still less than ten. Therefore, there is a slight increase in speed resulting from the shorter eta file, in addition to the increase resulting from the fact that the optimizer does not have to re-compute the factorizations.

For the problems listed in the table, the two-processor version took from 73% to 92% of the time used by the one-processor version. (This ratio, $\text{time}_2/\text{time}_1$, is listed in the last column.) Note that it would be unrealistic to expect a twofold speed increase using this type of parallelism on two processors, since we are assigning a specific task to each processor, rather than dividing a problem in half and assigning half to each processor. We would only achieve a twofold speed increase if the factorization and optimization each used exactly 50% of the total processor time, and there were no communications costs. There is no reason to believe that factorization and optimization use exactly the same amount of time; in fact, the ratio between the two varies considerably from problem to problem. Typically, less time is spent on factorization than on optimization, so it is in a sense wasteful to dedicate one of the two processors to factorization. Because the average length of the eta file tends to be approximately the same with one or two processors, we should expect a typical speedup of 20%, which is the amount of time typically spent on factorization in the one processor version. For some of the problems we tested, we did not achieve this speedup. Some possible explanations for this failure include the time spent synchronizing the two processors, and the fact that the average length of the eta file was sometimes longer in the two-processor version.

Since this technique will typically only produce a 20% speed increase by using a second processor, one might dismiss it as interesting but impractical.

There are, however, several cases in which the technique we have described could be very useful. First, if there is a situation in which there is a processor which is not being used at all, this technique can be employed to make use of it. For example, when solving some very large linear programs on supercomputers, so much memory is required that only one process can be run at a time. If the machine has two (or more) processors, the extra processor(s) might otherwise remain idle. Second, this technique could be applied in a heterogeneous parallel environment. The optimization task would be assigned to a more powerful processor, and the factorization task would be assigned to a less powerful processor. Third, this technique could be applied as part of a more fully parallel implementation of a linear program solving system, in which the optimization process is itself parallelized. For example, it might be efficient to assign four processors to the optimization, and one to the factorization.

Reference. Chvatal, Vasek, *Linear Programming*, W.H. Freeman and Company, New York, 1983.

Table 1

problem	(seconds)		iter1	iter2	refact	iter2/	time2/
	time1	time2				refact	time1
bandm	25.0	22.0	374	374	82	4.56	0.88
brandy	12.0	10.9	241	249	57	4.37	0.91
ganges	83.9	76.9	708	719	102	7.05	0.92
grow7	24.1	20.3	334	322	44	7.32	0.84
pilot	20315	14908	13968	15059	1034	14.56	0.73
pilot4	275	249	1742	1893	218	8.68	0.91
pilotwe	1237	961	5956	5689	922	6.17	0.78
sc205	6.41	5.56	177	165	44	3.75	0.87
scagr25	32.3	27.1	541	507	119	4.26	0.84
scsd1	6.39	5.71	290	268	78	3.44	0.89
scsd8	170	155	2116	2165	442	4.90	0.91