Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-10711 Berlin-Wilmersdorf, Germany

Robert E. Bixby
Alexander Martin

Parallelizing the Dual Simplex Method

Preprint SC-95-45 (December 1995)

Parallelizing the Dual Simplex Method

Robert E. Bixby[1]
Rice University and
CPLEX Optimization, Inc.
Houston, Texas, USA
bixby@rice.edu

Alexander Martin[2]
Konrad-Zuse-Zentrum
Berlin, Germany
martin@zib-berlin.de

Abstract

We study the parallelization of the steepest-edge version of the dual simplex algorithm. Three different parallel implementations are examined, each of which is derived from the CPLEX dual simplex implementation. One alternative uses PVM, one general-purpose System V shared-memory constructs, and one the PowerC extension of C on a Silicon Graphics multi-processor. These versions were tested on different parallel platforms, including heterogeneous workstation clusters, Sun S20-502, Silicon Graphics multi-processors, and an IBM SP2. We report on our computational experience.

1. Introduction

We investigate parallelizing the CPLEX[3] implementation of the dual simplex algorithm. We have chosen the dual over the primal for two reasons. First, the default primal algorithm typically uses some form of "partial pricing," thus removing a significant opportunity for parallelism. Second, we envision the primary application of this work to "reoptimization" in integer programming applications. There the dual is the natural algorithm, even for many very large, difficult models where, say, barrier algorithms [LuRo95] potentially provide better performance when solving from scratch. In addition, integer programming applications, particularly those that employ "column-generation," sometimes offer the opportunity to improve the underlying formulation by increasing the number of variables, thus improving the potential for parallelism.

2. Dual Simplex Algorithms

Consider a *linear program (LP)* in the following *standard form*:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array} \tag{1}$$

[1]Work partially supported by NSF grant CCR-9407142 to Rice University
[2]Work partially supported by the Center for Research in Parallel Computing, Rice University
[3]CPLEX is a registered trademark of CPLEX Optimization, Inc.

where $c \in \mathbf{R}^n$, $b \in \mathbf{R}^m$ and $A \in \mathbf{R}^{m \times n}$. Note that most practical LPs have nontrivial bounds on at least some variables; however, for purposes of this discussion it will suffice to consider problems in the form (1).

The dual of (1) is

$$\text{(2)} \qquad \begin{array}{ll} \max & b^T \pi \\ \text{s.t.} & A^T \pi \leq c \end{array}$$

Adding slacks yields

$$\text{(3)} \qquad \begin{array}{ll} \max & b^T \pi \\ \text{s.t.} & A^T \pi + d = c \end{array}$$

A *basis* for (1) is an ordered subset $B = (B_1, \ldots, B_m)$ of $\{1, \ldots, n\}$ such that $|B| = m$ and $\mathbf{B} = A_B$ is nonsingular. B is *dual feasible* if $c_N - A_N^T \mathbf{B}^{-T} c_B \geq 0$, where $N = \{1, \ldots, n\} \backslash B$.

Algorithm A generic iteration of the **standard dual simplex algorithm** for (1).

Input: A dual feasible basis B, $\bar{d}_N = c_N - A_N^T \mathbf{B}^{-T} c_B$ and $\bar{x}_B = \mathbf{B}^{-1} b$.

Step 1. If $\bar{x}_B \geq 0$, B is *optimal*–Stop; otherwise, let $i = \text{argmin}\{\bar{x}_{B_k} : k = 1, \ldots, m\}$. d_{B_i} is the *entering variable.*

Step 2. Solve $\mathbf{B}^T z = e_i$, were $e_i \in \mathbf{R}^m$ is the i^{th} unit vector. Compute $\alpha_N = -A_N^T z$.

Step 3. (*Ratio Test*) If $\alpha_N \leq 0$, (1) is *infeasible*–Stop; otherwise, let $j = \text{argmin}\{\bar{d}_k / \alpha_k : \alpha_k > 0, k \in N\}$. d_j is the *leaving variable.*

Step 4. Solve $\mathbf{B} y = A_j$.

Step 5. Set $B_i = j$. Update $\bar{x}_B$ (using y) and $\bar{d}_N$ (using z).

Remarks:

1. For all dual simplex algorithms, the efficient computation of $z^T A_N$ is crucial. This computation is implemented by storing A_N row-wise so that zero elements in z need be examined only once.

2. To improve stability, the ratio test (Step 3) is applied in several passes, using an idea of Harris [Ha73]. First, the ratios

$$r_k = \begin{cases} \bar{d}_k / \alpha_k & \text{if } \alpha_k > 0 \text{ and} \\ +\infty & \text{otherwise,} \end{cases}$$

are computed for each $k \in N$. Using these ratios, we compute

$$t = \min\{r_k + \epsilon / \alpha_k : k \in N\}$$

where $\epsilon > 0$ is the *optimality tolerance*, by default 10^{-6}. Finally, we compute the actual leaving variable using the formula

$$j = \text{argmax}\{\alpha_k : r_k \leq t\}.$$

Note that since $\epsilon > 0$, it is possible for some of the d_k to be negative, and hence that r_j is negative. In that case, depending upon the magnitude of r_j, we may *shift* c_j to some value at least $c_j + |d_j|$, and then repeat the calculation of t and j employing the new r_j. (See [GiMuSaWr89] for a discussion of the approach that suggested this *shifting*. The details of how these shifts are removed are beyond the scope of this discussion.)

3. In order to solve the two linear systems in the above algorithm (see Steps 2 and 4), we keep an updated LU-factorization of $\mathbf{B}$, using the so-called Forrest-Tomlin update [FoTo72]. For most models, a new factorization is computed once every 100 iterations. These computations may be considered part of step 5.

Steepest Edge

There are three different dual algorithms implemented in CPLEX: The standard algorithm, described above, and two *steepest-edge* variants. The default algorithm is steepest-edge.

Several steepest-edge alternatives are proposed in [FoGo72]. These algorithms replace the rule for selecting the index of the entering variable d_{B_i} by

$$i = \text{argmin}\{\bar{x}_{B_k}/\eta_k : k = 1, \ldots, m\},$$

where the η_k are the *steepest-edge norms.* The two alternatives implemented in CPLEX correspond to the choices

$$\text{(SE1)} \quad \eta_k = \sqrt{(e_k^T\mathbf{B}^{-1})(e_k^T\mathbf{B}^{-1})^T}, \text{ and}$$

$$\text{(SE2)} \quad \eta_k = \sqrt{(e_k^T\mathbf{B}^{-1})(e_k^T\mathbf{B}^{-1})^T + (e_k^T\mathbf{B}^{-1}A_N)(e_k^T\mathbf{B}^{-1}A_N)^T + 1}.$$

While it is too expensive to explicitly compute all η_k at each iteration, there are efficient update formulas. Letting $\{\eta_1, ..., \eta_m\}$ be the values of the norms at the start of an iteration, the values at the start of the next iteration for (SE1), $\bar{\eta}_k$, are given by the formula

$$\text{(SE1 norm update)} \quad \bar{\eta}_k^2 = \eta_k^2 - 2(\frac{y_k}{y_i})e_k^T\mathbf{B}^{-1}z + (\frac{y_k}{y_i})^2 z^T z \quad (k \neq i),$$

where y and z are as in the statement of the standard dual simplex algorithm. Note that the implementation of this formula requires the solution of one extra linear system per iteration, the one used to compute $\mathbf{B}^{-1}z$. As suggested in [FoGo72], this second "FTRAN" can be solved simultaneously with the linear system in Step 4, thus requiring only a single traversal of the updated LU-factorization of $\mathbf{B}$. Similar remarks apply to (SE2), for which the corresponding update formula is:

$$\text{(SE2 norm update)} \quad \bar{\eta}_k^2 = \eta_k^2 - 2(\frac{y_k}{y_i})e_k^T\mathbf{B}^{-1}w + (\frac{y_k}{y_i})^2(z^Tz + \alpha_N^T\alpha_N) \quad (k \neq i),$$

where $w = z + A_N\alpha_N$. Note that whereas the (SE1) update requires solving only one additional linear system with right-hand side z, updating the (SE2) norms requires the computation of w, and the additional linear system corresponding to computing $\mathbf{B}^{-1}w$.

The default dual in CPLEX uses the (SE1) norms with the approximate starting values $\eta_k = 1$ for all k. This choice corresponds to the assumption that most variables in the initial basis will be slacks or artificials.

Summary

In the sections that follow we discuss three different parallel implementations of the (SE1) variant of the standard dual simplex method: One using PVM, one using general-purpose System V shared-memory constructs, and one using the PowerC extension of C on an Silicon Graphics multi-processor. In section 3, we begin by outlining the basic plan for the PVM and "System V" approaches. Each of these requires some explicit form of data distribution. The PowerC version requires no such data distribution.

To illustrate where the primary opportunities for parallelism exist, and set the stage for the ensuing sections, we close this section with three profiles for runs on an SGI Power Challenge using the sequential version of CPLEX. The problem characteristics for the problems selected here are given in Table 12. In giving these profiles, we make use of the following designations, classifying the various parts of the algorithm:

Designation	Description
Enter	Step 1.
BTRAN	Solution of $\mathbf{B}^Tz = e_i$ (Step 2).
Pricing	Computation of $\alpha_N = -A_N^Tz$ (Step 2).
Ratio	Computation of the r_k and initial t (Step 3).
Pivot	Computation of j, shifting, subsequent t's (Step 3).
FTRAN	Solutions of $\mathbf{B}y = A_j$ and $\mathbf{B}w = z$.
Factor	Factorization and factorization update (Step 5).
Update-d	Update of $\bar{d}_N$.
Update-x	Update of $\bar{x}_B$ and $\bar{\eta}$.
Misc	All other work.

Thus $15.3 + 5.3 + 2.3 + 1.1 = 24.0\%$ of the work can be parallelized for *pilots* and 97.3% of the work for *aa300000* (see the first paragraph of the next section).

Algorithmic step	% of total computation time			
	pilots	cre_b	roadnet	aa300000
Enter	2.1	5.5	0.2	0.1
BTRAN	15.0	11.5	1.5	0.5
Pricing	15.3	33.1	57.2	65.4
Ratio	5.3	15.6	22.7	20.4
Pivot	2.3	3.9	6.9	4.4
FTRAN	31.2	20.5	3.3	1.1
Factor	20.3	3.7	1.2	0.4
Update-d	1.1	3.1	5.2	7.4
Update-x	2.5	0.6	0.6	0.2
Misc	4.9	2.5	1.2	0.1
Total	100.0	100.0	100.0	100.0

Table 1: CPLEX profiles.

3. Outline of the Data Distributed Implementation

In this section we discuss our data distributed implementations of the (SE1) version of the standard dual simplex method. The parallel model we use is master/slave with one master and (potentially) several slaves. We call the master the *boss* and the slaves *workers*. The *boss* keeps the basis, and each processor, including the *boss*, gets a subset of columns. Each column must belong to exactly one processor. All computations directly related to the basis are done sequentially, by the *boss*. The other steps can be executed in parallel: Pricing, Ratio, Pivot, and Update-d.

Table 2 outlines a typical dual simplex iteration. The steps that do not appear in bold face were described in the previous section. The first new step is the communication of the z vector, **Com**(z), from the *boss* to the *workers*. For the infeasibility test (see Step 3 of the dual simplex algorithm) the *workers* inform the *boss* in **Com**(α) whether their part of α_N satisfies $\alpha_N \leq 0$. The steps **Com**(t), Pivot, and **Com**(j) must then be performed iteratively until the pivot has been accepted. **Com**(j) consists of several parts. After each *worker* has sent its pivot element, the *boss* makes a choice and informs the "winning" *worker* that the entering column should be sent. The information in **Com(update)** includes the leaving variable and data for updating the reduced costs. This information is collected at several different points within the sequential code.

In view of the profile statistics given in the previous section, and the fact that Enter, BTRAN, FTRAN and Factor will all be executed on a single processor (the *boss*), it is plain that we cannot expect significant performance improvements unless the ratio of variables to constraints in a given LP is large. Indeed, our first thought was not only to enforce this requirement, but to concentrate on problems for which the total memory requirements were so large that they exceeded the memory available on a

	Boss	*Worker*
Enter	$*$	
BTRAN	$*$	
Com(z)	$\xrightarrow{z}$	
Pricing	$*$	$*$
Ratio	$*$	$*$
Com(α)	$\xleftarrow{\alpha}$	
Com(t)	$\xleftrightarrow{t}$	
Pivot	$*$	$*$
Com(j)	$\xleftrightarrow{j}$	
FTRAN	$*$	
Factor	$*$	
Com(update)	$\xrightarrow{update}$	
Update-d	$*$	$*$
Update-x	$*$	

Table 2: The arrows in this table indicate where communication between the *boss* and the *workers* must occur, with directions indicating the direction of data flow. An asterisk marks where a task is performed.

single processor. Thus, we began by considering possibly heterogeneous networks of workstations connected by a local area network. As communication software we used PVM.

4. PVM

PVM (Parallel Virtual Machine) is a general purpose software package that permits a network of heterogeneous Unix computers to be used as a single distributed-memory parallel computer, called a virtual machine. PVM provides tools to automatically initiate tasks on a virtual machine and allows tasks to communicate and synchronize [4].

Our first implementation was in one-to-one correspondence with the sequential code. Thus, the *boss* immediately sent a request to the *workers* whenever some particular information was needed. Where possible, the *boss* then performed the same operations on its set of columns, thereafter gathering the answers from the *workers*. Assuming that the first selected pivot was accepted, this approach led to from 6 to 10 communication steps per iteration, depending on whether the entering and/or leaving column belonged to the *workers*. The data was partitioned in our initial implementation by distributing the columns equally among the processors.

[4]PVM is public domain and accessible over anonymous ftp via netlib2.cs.utk.edu. For details on PVM, see the PVM man pages. In our implementation we used PVM Version 3.3.7.

Table 3 shows the results of our initial tests, carried out on the NETLIB problems.[5] Results for larger problems are presented later. The *boss* was run on a SUN S20-TX61 and the one *worker* on a SUN 4/10-41. The two workstations were connected by a 10 Mb/s (megabits per second) Ethernet. The sequential code was run on the SUN S20-TX61. The times, measured in wallclock seconds, do not include reading and presolving.

Model	Sequential		2 processors	
	Time	Iterations	Time	Iterations
NETLIB	3877.8	130962	12784.8	137435

Table 3: First results on local area network.

Note that the parallel version was approximately 3.3 times slower than the sequential version! Most, but not all of this excess time was due to communication costs, which suggested the following improvements.

1. In Com(j) each *worker* sends not only the pivot element but simultaneously the corresponding column. This modification saves one communication step, since the *boss* no longer needs to inform the "winning" *worker* to send a column.

2. The information for the infeasibility test Com(α) can be sent in Com(j). In case infeasibility is detected, the pivot computation is wasted work, but such occurrences are rare.

3. The pivot selection strategy was changed to reduce the number of communication steps. Each processor determines its own t and performs the Ratio Test independently of the other processors. The *workers* then send their selected pivots and t values to the *boss*, which makes the final selection. This procedure reduces the number of communication steps in Com(t) and Com(j) from $3 \cdot$ (number of rejected pivots +1) to 3. The further application of 1. reduces the number to 2.

[5]NETLIB problems: afiro, sc50b, sc50a, sc105 , kb2, adlittle, scagr7, stocfor1, blend, sc205, recipe, share2b, vtpbase, lotfi, share1b, boeing2, scorpion, bore3d, scagr25, sctap1, capri, brandy, israel, finnis, gfrdpnc, scsd1, etamacro, agg, bandm, e226, scfxm1, grow7, standata, scrs8, beaconfd, boeing1, shell, standmps, stair, degen2, agg2, agg3, scsd6, ship04s, seba, tuff, forplan, bnl1, pilot4, scfxm2, grow15, perold, fffff800, ship04l, sctap2, ganges, ship08s, sierra, scfxm3, ship12s, grow22, stocfor2, scsd8, sctap3, pilotwe, maros, fit1p, 25fv47, czprob, ship08l, pilotnov, nesm, fit1d, bnl2, pilotja, ship12l, cycle, 80bau3b, degen3, truss, greenbea, greenbeb, d2q06c, woodw, pilots, fit2p, stocfor3, wood1p, pilot87, fit2d, dfl001. Size statistics for non-NETLIB problems employed in our testing are given in Table 12, ordered by the sequential solution times given in Table 11. For the most part these models were collected from proprietary models available to the first author through CPLEX Optimization, Inc.. With the exception of aa6, all models with names of the form 'aa$K$', where K is an integer, are K-variable initial segments of the 12,753,312 variable "American Airlines Challenge Model" described in [BiGrLuMaSh92]. All solution times given in this paper are real (wallclock) times in seconds, unless otherwise noted, and are for the reduced models obtained by applying the default CPLEX presolve procedures.

4. All relevant information for the *workers*' update is already available before FTRAN. Note that the *workers* need only know the entering and leaving column and the result from the Ratio Test in order to update the reduced costs. Thus, only one communication step after Pivot is needed for the update.

5. PVM offers different settings to accelerate message passing for homogeneous networks. We make use of these options where applicable.

6. Load balancing was (potentially) improved as follows: Instead of distributing columns based simply upon the number of columns, we distributed the matrix nonzeros in as nearly equal numbers as possible over all processors.

Table 4 shows the results on the NETLIB problems after implementing the above improvements. For a typical simplex iteration, the number of communication steps was reduced to three: the *boss* sends z, the *workers* send their pivots and corresponding columns, and the *boss* sends information for the update.

Example	2 processors	
	Time	Iterations
NETLIB	7736.5	142447

Table 4: Improved results on local area network.

Based upon Table 4, the implementation of 1.-6. improves computational times by a factor of 1.6, even though increasing the number of iterations slightly. However, the performance of the parallel code is still significantly worse than that of the sequential code. One reason is certainly the nature of the NETLIB problems. Most are either very small or have a small number of columns relative to the number of rows. Table 5 gives corresponding results for a test set where the ratio of columns to rows was more favorable.

Example	Sequential		2 processors	
	Time	Iterations	Time	Iterations
0321.4	9170.1	21481	7192.0	20178
cre_b	614.5	11121	836.1	13219
nw16	120.7	313	83.1	313
osa030	645.8	2927	515.4	3231
roadnet	864.7	4578	609.6	4644

Table 5: Larger models on a local area network.

The results are significantly better. With the exception of *cre_b*, the parallel times are between 20% (for *osa030*) and 37% (for *nw16*) faster, though, again largely due to communication costs, still not close to equaling linear speedup. Our measurements indicated that communication costs amounted to between 30% (for *osa030*) and 40% (for *cre_b*) of the total time. Since communication was taking place over Ethernet, we decided to test our code on two additional parallel machines where communication did not use Ethernet, a SUN S20-502 with 160 MB of RAM memory and an IBM SP2 with eight processors (each a 66 MHz thin-node with 128 MB of RAM). The nodes of the SP2 were interconnected by a high speed network running in TCP/IP mode.

Example	Sequential		2 processors	
	Time	Iterations	Time	Iterations
NETLIB	4621.2	130962	6931.1	142447
0321.4	9518.3	21481	8261.1	20178
cre_b	650.5	11121	769.4	13219
nw16	99.6	313	78.4	313
osa030	556.3	2927	502.1	3231
roadnet	801.0	4578	652.5	4644

Table 6: Larger models on SUN S20-502.

The results on the SUN S20-502 were unexpectedly bad, worse than those using Ethernet. We will come to possible reasons for this behavior later. The results on the SP2 were much better (with the exception of *cre_b*) and seem to confirm our conclusions concerning the limitations of Ethernet.

Example	Sequential		2 processors		4 processors	
	Time	Iterations	Time	Iterations	Time	Iterations
NETLIB	2140.9	130054	5026.9	143348	not run	not run
0321.4	5153.7	24474	3624.6	26094	2379.7	21954
cre_b	390.2	11669	399.8	11669	458.9	10915
nw16	94.0	412	50.4	412	30.4	412
osa030	321.3	2804	191.8	2804	152.7	2836
roadnet	407.3	4354	235.5	4335	182.4	4349

Table 7: Larger models on SP2.

To summarize, there seems little hope of achieving good parallel performance on a general set of test problems using a distributed-memory model. That result is not unexpected. However, the distributed memory code is not without applications. as illustrated by the final table of this section.

The two examples in Table 8 did not fit onto a single node of the machine being used, so we could not compare the numbers to sequential times. However, the CPU-time spent on the *boss* was 9332.9 sec. (90.5% of the real time) for *aa6000000* and 52.5

Example	Time	Iterations
aa6000000	10315.8	10588
us01	59.4	249

Table 8: Large airline models on SP2 using all 8 nodes.

sec. (= 88.5% of the real time) for *us01*. Time measurements for the smaller examples in Table 7 confirm that about 10% went for communication.

In closing this section, we note that one of the biggest limitations of PVM is directly related to its portability. The generality of PVM means that transmitted data usually must be passed through different interfaces and thereby often packed, unpacked, encoded, decoded, etc. For multiprocessors like the SUN S20-502 or the Power Challenge (see section 5), this work is unnecessary.

4. Shared Memory/Semaphores

Based upon our results using PVM we decided to investigate the use of general-purpose, UNIX System V shared-memory constructs. We restricted our choice to System V mainly because it provides high portability. Possible candidates for inter-process communication (IPC) on a single computer system are *pipes*, *FIFOs*, *message queues*, and *shared memory* in conjunction with *semaphores* (for an excellent description of these methods see [St90]). We looked at the performance of these four types of IPC by sending data of different sizes between two processors. It turned out that the shared memory/semaphore version was the fastest (see also [St90], page 683). *Shared Memory* allows two or more processes to share a certain memory segment. The access to such a shared memory segment is synchronized and controlled by *semaphores*. There are different system calls available that create, open, give access, modify or remove shared memory segments and semaphores. For a description of these functions, see the man pages of Unix System V or [St90].

We implemented our shared memory version in the following way: We have one shared memory segment for sending data from the *boss* to the *workers*. This segment can be viewed as a buffer of appropriate size. All the data to be sent to the *workers* is copied into this buffer by the *boss* and read by the *workers*. The *workers* use the first four bytes to determine the type of the message. The access to the buffer is controlled by semaphores. In addition, we have one shared memory segment for each *worker* to send messages to the *boss*. These segments are used in the same manner as the "sending buffer" of the *boss*.

The shared memory version differs from the PVM version in the following respects:

1. The *workers* do not send the pivot column immediately together with the pivot element, i.e., improvement 1. on page 7 is removed: There might be several pivot elements sent (and thus columns) per iteration, depending upon numerical

considerations. This behavior could result in overflow in the shared memory buffer. On the other hand, informing a *worker* to send a column is relatively inexpensive using semaphores.

2. We changed the pivot selection strategy (see 3. on page 7) back to that of the sequential code, mainly because we wanted to have the same pivot selection strategy for an easier comparison of the results and because the additional communication steps are not time-consuming using shared memory and semaphores.

3. We saved some data copies by creating another shared memory segment for the vector z. Thus, in Com(z) the *workers* are notified of the availability of the new vector by a change of the appropriate semaphore value.

Table 9 shows the results of the shared memory version on the SUN S20-502.

Example	2 processors	
	Time	Iterations
NETLIB	5593.3	141486
0321.4	7958.2	20465
cre_b	604.9	13219
nw16	82.2	313
osa030	545.1	3231
roadnet	711.2	4644

Table 9: Shared memory version on SUN S20-502.

The results on the SUN S20-502 are again not satisfactory. For the NETLIB problems the times are better than those using PVM, but are still far inferior to the CPLEX sequential times. For the larger models the numbers are even worse. Two contributors to these negative results are the following:

1. The semaphore approach is probably not the right way to exploit shared memory for the fine-grained parallelization necessary in the dual simplex method. It is true that there are other communication primitives available that might be faster. However, as this work was being done, there did not seem to be any better approach available that was portable. We will come to this point again in the next section.

2. There is a serious memory bottleneck in the SUN S20-502 architecture. Because the data bus is rather small, processes running in parallel interfere with each other when accessing memory. Looking at the SPEC results for the single processor and 2-processor models (see [Sun]) we have

	SUN S20-50	SUN S20-502
SPECrate_int92	1708	3029
SPECrate_fp92	1879	3159

This means that up to about 19% is lost even under ideal circumstances. For memory intensive codes like CPLEX, the numbers are even worse. For the NETLIB problems, we ran CPLEX alone and twice in parallel on the SUN S20-502:

CPLEX (alone)	CPLEX (twice in parallel)
4621.2 sec.	6584.4 sec.
	6624.7 sec.

This degradation was about 40%! Clearly the SUN S20-502 has serious limitations in parallel applications[6].

The Silicon Graphics Power Challenge multi-processors are examples of machines that do not suffer from this limitation. Table 10 summarizes our tests running the System V semaphore implementation on a two-processor, 75 Mhz Silicon Graphs R8000 multi-processor.

We note that the five larger models (*0321.4, cre_b, nw16, osa030*, and *roadnet*) achieve reasonable, though with one exception not linear speedups, ranging from 22% for *cre_b* to 105% for *nw16*. One reason that better speedups are not obtained is that a significant fraction of the communication costs is independent of problem size – indeed, all steps to the point that the *worker* sends an entering column. As a consequence, examples with low-cost iterations cannot be expected to achieve significant speedups. This phenomenon is illustrated by *aa25000, sfsu4, nopert, cre_b, mctaq, usfs2, food, aa6, ra1, pilots*, and especially the NETLIB problems (including *fit2d*), where on average at most 0.03 seconds are needed per iteration, running sequentially. All other examples where, in addition, the number of iterations of the sequential and parallel codes are roughly equal, give approximately the desired speedup. The "aa" examples behave particularly well: The numbers of iterations are constant, individual iterations are expensive, the fraction of work that can be parallelized is near 100% (see Table 1).

Finally, note that (*mctaq, sfsu2, sfsu3, finland*, and *imp1*), fail to follow any particular trend, primarily because the number of iterations for the parallel and sequential codes differ drastically. That such differences arise was unexpected, since the pivot selection strategy in both codes is the same, as is the starting point. However, since the basis is managed by the *boss* we distribute only the initial nonbasic columns among the processors, resulting in a possible column reordering. With this reordering, different columns can be chosen in the Pricing step, leading to different solution paths. Note, however, that in terms of time per iteration, the five listed models do achieve close to linear speedups.

---

[6]Sun Microsystems gave us the opportunity to test some of these examples under an optimal environment on their machines. On the SUN S20-502 we got the same results as on our machine, whereas on a SUN S20-712 the degradation was at most 20%. These better results are mainly due to the 1 MB external cache each of the two processors of a SUN S20-712 has. The extra cache helps in avoiding bottlenecks on the data bus.

Example	Sequential		2 processors		Speedup
	Time	Iterations	Time	Iterations	
NETLIB	2004.4	133299	2361.7	138837	0.8
aa25000	7.9	546	7.1	546	1.1
aa6	22.7	2679	26.2	2679	0.9
w1.dual	27.2	67	13.5	67	2.0
aa50000	34.1	916	23.2	916	1.5
nw16	109.2	403	53.3	403	2.0
ra1	51.1	3091	46.2	3091	1.1
pilots	71.2	4211	82.2	4437	0.9
aa75000	105.1	1419	60.8	1419	1.7
fit2d	131.7	6366	97.0	6959	1.4
sfsu4	71.5	2256	66.4	2414	1.1
us01	782.5	278	350.8	278	2.2
usfs2	241.0	8356	268.5	7614	0.9
aa100000	257.2	2133	128.8	2133	2.0
osa030	354.8	2943	192.2	2833	1.8
roadnet	378.9	4405	213.9	4608	1.8
cre_b	337.8	10654	275.3	10654	1.2
nopert	424.1	26648	249.9	24185	1.7
continent	771.6	16586	558.8	16570	1.4
food	653.5	21433	598.4	21328	1.1
mctaq	531.4	28714	683.1	41460	0.8
0341.4	564.8	8225	394.8	8225	1.4
sfsu3	779.2	4055	804.0	9436	1.0
aa200000	1262.4	4090	632.2	4090	2.0
finland	1654.1	24356	1560.7	31416	1.0
osa060	2182.7	5787	1074.5	5801	2.0
sfsu2	1818.2	12025	1828.0	23200	1.0
aa300000	2724.0	5513	1339.5	5513	2.0
amax	3122.5	8276	1923.9	9780	1.6
aa400000	4068.9	5931	1964.7	5931	2.1
0321.4	4406.2	20677	2681.2	20662	1.6
aa500000	6081.8	6747	2878.1	6747	2.1
imp1	8252.9	38421	3231.4	30036	2.6
aa600000	7619.0	6890	3599.5	6890	2.1
tm	8154.3	74857	5478.7	71657	1.5
aa700000	9746.5	7440	4536.4	7440	2.1
aa800000	11216.1	7456	5172.8	7456	2.2
aa900000	13130.8	7590	6028.9	7590	2.2
aa1000000	15266.6	7902	7030.5	7902	2.2

Table 10: Run times using semaphores on 75 Mhz Silicon Graphics R8000.

5. PowerC

We describe a thread-based parallel implementation of the dual steepest-edge algorithm on an SGI Power Challenge using the SGI PowerC extension of the C programming language [SGI]. We note that the work described in this section was carried out at a somewhat later date than that in previous sections, and hence that the initial sequential version of CPLEX was somewhat different. As the tables will show, this version not exhibited improved performance when parallelized, but was significantly faster running sequentially.

In our work we use only a small subset of the compiler directives provided by the PowerC extension: `#pragma parallel`, `#pragma byvalue`, `#pragma local`, `#pragma shared`, `#pragma pfor`, and `#pragma synchronize`. The `parallel` pragma is used to define a parallel *region*. The remaining pragmas are employed inside such a region.

Defining a parallel region is analogous to defining a C function. The `byvalue`, `local`, and `shared` directives specify the argument list for that function, with each directive specifying the obvious types – for example, `shared` specifies pointers that will be shared by all threads. Exactly one `#pragma synchronize` is used in our implementation (it could be easily avoided by introducing another parallel region). All of the actual parallelism is invoked by the loop-level directive `pfor`.

The key parallel computation is the Pricing step. If this step were carried out in the straightforward way, it's parallelization would also be straightforward, employing the following sort of loop (inside a parallel region):

```
#pragma pfor iterate (i = 0; nrows; 1)
for (j = 0; j < ncols; j++) {
  compute a sparse inner product for column j;
}
```

where `ncols` denotes the number of columns and `nrows` the number of rows. However, as noted earlier, CPLEX does not carry out the Pricing step column-wise. In order to exploit sparsity in z (see Step 2), the part of the constraint matrix corresponding to the nonbasic variables at any iteration is stored in a sparse data structure by row, and this data structure is updated at each iteration by deleting the *entering variable* (which is "leaving" the nonbasic set) and inserting the *leaving variable*.

Given that A_N is stored by row, the computation of $z^T A_N$ could be parallelized as follows:

```
#pragma pfor iterate (i = 0; nrows; 1)
for (i = 0; i < nrows; i++) {
   d_N += z[i] * (ith row of A_N);
}
```

where the inner computation itself is a loop computation. The difficulty with this approach is that it creates *false sharing*. In particular, the individual entries in d_N

will be written to by all threads, causing this data to be constantly moved among the processor caches. One obvious way to avoid this difficulty is to create separate target arrays d_{N_p}, one for each thread, with the actual update of d_N carried out as a sequential computation following the computation of the d_{N_p}. However, a much better approach is to directly partition N into subsets, one for each thread. To do so required restructuring a basic CPLEX data structure and the routines that accessed it. Once that was done, the implementation of the parallel pricing was straightforward.

Where K is a multiple of the number of processors, let

$$0 = n_0 \leq n_1 \leq n_2 \leq \ldots \leq n_K = \text{ncols},$$

and let $P_k = \{n_k, \ldots, n_{k+1} - 1\}$ for $k = 0, \ldots, K-1$. The n_k are chosen so that the numbers of nonzeros in A_{P_k} are as nearly equal as possible. For a given set of nonbasic indices N, the corresponding partition is then defined by $N_k = N \cap P_k$. Using this partition, the parallel pricing loop takes the form

```
#pragma pfor iterate (k = 0; K; 1)
for (k = 0; k < K; k++) {
   for (i = 0; i < nrows; i++) {
      d_N_k += z[i] * (ith row of A_N_k);
   }
}
```

In initial testing of the above partitioning, an interesting phenomenon was discovered, related at least in part to the cache behavior of the R8000. Consider the model *aa400000*. Running the sequential code with no partitioning yielded a timing of 2864.1 seconds while the initial PowerC version on two processors using $K = 2$ ran in 1300.4 seconds, a speedup considerably greater than 2.0. Setting $K = 2$ in the sequential code yielded a run time of 2549.4, much closer to what one would expect. After considerable testing, we thus chose to set K – in both the sequential and parallel instances – to be the smallest multiple of the number of processors that satisfies $K \geq ncols/(50\ nrows)$. Thus, for *aa400000* and two processors, K was 8, the smallest multiple of 2 greater than $259924/(50 \cdot 837)$. We note that this change also seems to have benefitted other platforms. The dual solution time for *fit2d* on a 133 Mhz Pentium PC was 204.5 seconds with $K = 1$ and 183.7 with the new setting of $K = 9$.[7]

We now comment on the remaining steps that were parallelized in the dual algorithm: Enter, Ratio, Pivot, Update-d, and the update of the row-wise representation of A_N. Update-x could also have been parallelized, but was not after initial testing indicated that doing so was at best of marginal value, and in some cases actually degraded performance. The total effort consumed by this step was simply too small to justify the overhead for executing a parallel region.

[7] Dual is not the way to solve *fit2d*, especially not on a PC. The solution time using simplex primal was 18.6 seconds and using the barrier algorithm 15.4 seconds.

Ratio and Pivot: For these computations we use the same partition of N used in the Pricing step. Note that the dual algorithm allows these two steps to be performed without any intervening computations. As it turned out, in the CPLEX sequential implementation, before the current work was carried out, there were several relatively inexpensive, minor computations that were interspersed between the two major steps. Since entering and leaving parallel regions does incur some fixed costs, it seemed important to be able to do the Pricing and Ratio steps inside a single region; moreover, with some reorganization within each of these computations, it was possible to carry out the "major part" of each step without introducing synchronization points. Thus, the essential form of the computation as implemented was the following:

```
#pragma pfor iterate (k = 0; K; 1)
for (k = 0; k < K; k++) {
   for (i = 0; i < nrows; i++) {
      d_N_k += z[i] * (ith rows of A_N_k);
   }
   ratio test int N_k;
}
```

The reorganization of computations for these two steps, as well as other reorganizations to facilitate the parallel computation were carried out so that they also applied when the dual was executed sequentially, thus preserving code unity.

Enter: Since this computation is easy to describe in essentially complete detail, we use it as an illustration of the precise syntax for the PowerC directives:

```
#pragma parallel
#pragma byvalue (nrows)
#pragma local (i_min, min, i)
#pragma shared (x_B, norm, i_min_array)
{
   i_min = -1;
   min   = 0.0;
   #pragma pfor iterate (i = 0; nrows; 1)
   for (i = 0; i < nrows; i++) {
      if ( x_B[i] < min * norm[i] ) {
         min   = x_B[i] / norm[i];
         i_min = i;
      }
   }
   i_min_array[mpc_my_threadnum ()] = i_min;
}
i_min = -1;
min   = 0.0;
for (i = 0; i < mpc_numthreads (); i++) {
   if ( i_min_array[i] != -1 ) {
```

```
        if ( x_B[i_min_array[i]] < min * norm[i_min_array[i]] ) {
           min   = x_B[i_min_array[i]] / norm[i_min_array[i]];
           i_min = i_min_array[i];
        }
     }
  }
```

The PowerC function `mpc_my_threadnum()` returns the index of the thread being executed, an integer from 0 to $K-1$, where K is the total number of threads. The function `mpc_numthreads()` returns K.

A_N **update:** The insertion of new columns is a constant-time operation. However, the deletion operation can be quite expensive. It was parallelized in a straightforward manner.

Finally we remark on one important computation that was not parallelized. As discussed earlier, the dual steepest-edge algorithms all require the solution of one additional FTRAN per iteration. The result is that two ostensibly "independent" solves are performed using the same basis factorization. These solves are typically quite expensive, and it would seem clear that they should be carried out in parallel (on two processors). However, in the sequential code these two solves have been combined into a single traversal of the factorization structures. That combination, when carefully implemented, results in some reduction in the actual number of computations as well as a very effective use of cache. As a result, all our attempts to separate the computations and perform them in parallel resulted in a degradation in performance.

Computational Results

The computational results for the PowerC parallel dual are given in Table 11. Tests were carried out on a 4-processor 75 Mhz R8000. (There was insufficient memory to run *aa6000000*.)

Comparing the results in Table 11 to the profiles in Table 1, we see that *pilots* – as expected, because of the large fraction of intervening non-parallel work – did not achieve ideal performance; on the other hand, *cre_b* came very close to the ideal speedup and *aa300000* exceeded ideal speedup by a considerable margin.

There are unfortunately several, as yet unexplained anomalies in our results. These mainly show up on larger models. In several instances superlinear speedups are achieved. Examples are *aa200000* and *imp1*, with 4-processor speedups exceeding factors of 5. On the other hand, other models that would seem even more amenable to parallelism, principally the four largest "aa" models, achieve speedups considerably smaller than 4 on 4 processors. At this writing, the authors can offer no better explanation than that these anomalies are do to R8000 cache and memory bus properties.

Example	Iterations	Run time (no. of processors)				Speedups		
		1	2	3	4	2	3	4
NETLIB	136369	1310.2	1216.2	1151.3	1123.6	1.1	1.1	1.2
aa25000	552	3.7	2.9	2.4	2.1	1.3	1.5	1.8
aa6	2509	12.1	10.4	9.5	9.1	1.2	1.3	1.3
w1.dual	67	16.5	9.7	7.2	5.9	1.7	2.3	2.8
aa50000	1038	20.0	11.5	8.5	7.1	1.7	2.4	2.8
nw16	256	21.9	10.6	6.7	5.2	2.1	3.3	4.2
ra1	3018	26.4	20.4	17.9	16.6	1.3	1.5	1.6
pilots	4196	44.2	42.2	40.9	40.5	1.0	1.1	1.1
aa75000	1360	45.8	21.2	15.3	12.6	2.2	3.0	3.6
fit2d	5724	49.2	29.3	21.5	17.3	1.7	2.3	2.9
sfsu4	3071	56.7	35.8	27.6	23.6	1.6	2.1	2.4
us01	245	108.9	57.2	39.2	30.0	1.9	2.8	3.6
usfs2	7962	114.4	83.9	72.2	65.5	1.4	1.6	1.8
aa100000	2280	153.1	64.3	43.3	32.9	2.4	3.5	4.7
osa030	2831	154.4	67.8	46.3	37.3	2.3	3.3	4.1
roadnet	3921	164.5	75.5	51.5	42.3	2.2	3.2	3.9
cre_b	11136	168.5	124.9	107.9	100.5	1.3	1.6	1.7
nopert	27315	197.4	135.9	113.9	99.6	1.5	1.7	2.0
continent	12499	236.7	163.9	141.5	128.9	1.4	1.6	1.8
food	21257	311.3	259.5	238.5	223.9	1.2	1.3	1.4
mctaq	30525	317.0	219.2	177.9	153.0	1.4	1.8	2.1
0341.4	9190	341.5	205.7	168.3	146.9	1.7	2.0	2.3
sfsu3	3692	413.4	201.5	130.8	99.3	2.1	3.2	4.2
aa200000	3732	675.4	318.4	189.8	128.1	2.1	3.6	5.3
finland	29497	1086.8	691.4	580.3	526.4	1.6	1.9	2.1
osa060	5753	1197.8	548.1	328.0	241.0	2.2	3.7	5.0
sfsu2	16286	1724.5	1060.3	761.9	609.3	1.6	2.3	2.8
aa300000	5865	1743.1	876.4	557.7	381.7	2.0	3.1	4.6
amax	9784	2093.8	1151.7	795.2	625.3	1.8	2.6	3.4
aa400000	6271	2473.0	1286.5	855.4	629.0	1.9	2.9	3.9
0321.4	19602	2703.6	1599.4	1218.5	1034.7	1.7	2.2	2.6
aa500000	6765	3349.5	1713.6	1165.4	879.9	2.0	2.9	3.8
imp1	29297	3424.5	1423.0	868.0	651.5	2.4	3.9	5.3
aa600000	6668	3904.9	2019.1	1393.0	1054.9	1.9	2.8	3.7
tm	70260	4230.5	2633.7	2232.8	1997.5	1.6	1.9	2.1
aa700000	7162	4951.4	2542.8	1760.0	1361.5	1.9	2.8	3.6
aa800000	7473	5763.1	3000.6	2084.1	1616.1	1.9	2.8	3.6
aa900000	8166	7242.4	3738.3	2606.4	2020.2	1.9	2.8	3.6
aa1000000	7703	7413.5	3851.2	2687.5	2089.4	1.9	2.8	3.6

Table 11: PowerC run times on 1 to 4 processors.

Example	Original			Presolved		
	Rows	Columns	Nonzeros	Rows	Columns	Nonzeros
aa25000	837	25000	192313	837	17937	140044
aa6	541	4486	25445	532	4316	24553
w1.dual	42	415953	3526288	22	140433	1223824
aa50000	837	50000	380535	837	35331	276038
nw16	139	148633	1501820	139	138951	1397070
ra1	823	8904	72965	780	8902	70181
pilots	1441	3652	43167	1275	3243	40467
aa75000	837	75000	576229	837	52544	415820
fit2d	25	10500	129018	25	10450	128564
sfsu4	2217	33148	437095	1368	24457	180067
us01	145	1053137	13636541	87	370626	3333071
usfs2	1484	13822	158612	1166	12260	132531
aa100000	837	100000	770645	837	68428	544654
osa030	4350	100024	600144	4279	96119	262872
roadnet	463	42183	394187	462	41178	383857
cre_b	9648	72447	256095	5229	31723	107169
nopert	1119	16336	50749	1119	16336	50749
continent	10377	57253	198214	6841	45771	158025
food	27349	97710	288421	10544	69004	216325
mctaq	1129	16336	52692	1129	16336	52692
0341.4	658	46508	384286	658	27267	264239
sfsu3	1973	60859	2111658	1873	60716	2056445
aa200000	837	200000	1535412	837	134556	1075761
finland	56794	139121	658616	5372	61505	249100
osa060	10280	232966	1397796	10209	224125	584253
sfsu2	4246	55293	984777	3196	53428	783198
aa300000	837	300000	2314117	837	197764	1595300
amax	5160	150000	6735560	5084	150000	3237088
aa400000	837	400000	3115729	837	259924	2126937
0321.4	1202	71201	818258	1202	50559	656073
aa500000	837	500000	3889641	837	320228	2624731
imp1	4089	121871	602491	1587	112201	577607
aa600000	837	600000	4707661	837	378983	3138105
tm	28420	164024	505253	17379	139529	354697
aa700000	837	700000	5525946	837	434352	3620867
aa800000	837	800000	6309846	837	493476	4112683
aa900000	837	900000	7089709	837	548681	4575788
aa1000000	837	1000000	7887318	837	604371	5051196
aa6000000	837	6000000	46972327	837	2806468	23966705

Table 12: Problem statistics.

Conclusions

We described three different approaches to implementing parallel dual simplex algorithms. The first of these, using distributed memory and PVM, gave acceptable speedups only for models where the ratio of rows to columns was very large. It seemed most applicable to situations involving very large models with memory requirements too large for available single processors.

We examined two shared memory implementations. The first of these used System V constructs, and, not surprisingly, produced better results than the PVM implementation, but, in many ways, not significantly better. Finally, we constructed a thread-based, shared-memory implementation using the Silicon Graphics PowerC extension of the C programming language. This implementation was far simpler than the previous two, and produced quite good results for a wide range of models. It seems likely that this thread-based approach can also be used to produced equally simple and useful parallel dual simplex implementation on other multi-processors with memory buses having adequate bandwidth.

Finally, we note that primal steepest-edge as well as other "full-pricing" alternatives in the primal simplex algorithm, are also good candidates for parallelization.

References

[BiGrLuMaSh92] R. E. Bixby, J. W. Gregory, I. J. Lustig, R. E. Marsten, and D. F. Shanno, 1992. Very Large-Scale Linear Programming: A Case Study in Combining Interior Point and Simplex Methods. *Operations Research 40*, 885–897.

[FoGo72] J. J. Forrest and D. Goldfarb, 1992. Steepest-Edge Simplex Algorithms for Linear Programming. *Mathematical Programming 57*, 341–374.

[FoTo72] J. J. H. Forrest and J. A. Tomlin, 1972. Updating Triangular Factors of the Basis to Maintain Sparsity in the Product-Form Simplex Method. *Mathematical Programming 2*, 263–278.

[GiMuSaWr89] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright, 1989. A Practical Anti-Cycling Procedure for Linearly Constrained Optimization, *Mathematical Programming 45*, 437–474.

[Ha73] P. M. J. Harris, 1973. Pivot Selection Methods of the Devex LP Code. *Mathematical Programming 5*, 1–28.

[LuMaSh94] I. J. Lustig, R. E. Marsten, and D. F. Shanno, 1994. Interior Point Methods for Linear Programming: Computational State of the Art. *ORSA Journal on Computing 6*, 1–14.

[LuRo95] I. J. Lustig, E. Rothberg, 1995. Gigaflops in Linear Programming. To appear in *OR Letters*.

[SGI] *Power C User's Guide*, Silicon Graphics, Inc.

[St90] W. R. Stevens, *Unix Network Programming*, PTR Prentice-Hall, New Jersey, 1990.

[Sun] Sun Microsystems, *WWW Page: http://www.sun.com/smi/bang/ss20.spec.html.*