ASYNPLEX, an asynchronous parallel revised simplex algorithm

J.A.J. Hall K.I.M. McKinnon

February 1998

MS 95-050b

Supported by EPSRC research grant GR/J08942

Presented at APMOD95 Brunel University 3rd April 1995

Department of Mathematics and Statistics
University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ
Tel. (33) 131 650 5075 E-Mail : jajhall@maths.ed.ac.uk, ken@maths.ed.ac.uk

ASYNPLEX, an asynchronous parallel revised simplex algorithm

J. A. J. Hall K. I. M. McKinnon

27^{th} February 1998

Abstract

This paper describes ASYNPLEX, an asynchronous variant of the revised simplex method which is suitable for parallel implementation on a shared memory multiprocessor or MIMD computer with fast inter-processor communication. The method overlaps simplex iterations on different processors. Candidates to enter the basis are tentatively selected using reduced costs which may be out of date. Later the up-to-date reduced costs of the tentative candidates are calculated and candidates are either discarded or accepted to enter the basis. The implementation of this algorithm on a Cray T3D is described and results demonstrating significant speed-up are presented.

1 Introduction

Linear programming (LP) is a widely applicable technique both in its own right and as a sub-problem in the solution of other optimization problems. The revised simplex method and the barrier method are the two efficient methods for general LP problems on serial machines. There have been successful parallel implementations of the barrier method but as yet little progress has been reported on parallel methods based on the revised simplex algorithm. In contexts where families of related LP problems have to be solved, such as in integer programming and decomposition methods, the revised simplex method is usually the more efficient method, so there is strong motivation to devise a parallel version of this method. If this is to be of value then it should be significantly faster than current serial simplex solvers.

The particular approach to exploiting parallelism which is considered in this paper is to overlap simplex iterations performed by a number of *iteration processes*, with an additional *invert process* devoted to calculating a factored inverse of simplex basis matrices. Candidates for variables to enter the basis are tentatively selected using the most recently available reduced costs, and the true reduced costs for these columns are calculated cheaply later before finally deciding whether or not they will enter the basis. The effectiveness of the method relies on there being some persistence in the values of the reduced costs in the course of a small number of iterations. The algorithm is a variant of the simplex method and as such follows a single path on the surface of the feasible region. This requires the coordination of the basis change decisions among all the processors.

A detailed description of the algorithm ASYNPLEX is given in Section 2. Its implementation on a Cray T3D is described in Section 3. Computational results presented in Section 4 demonstrate significant speed-up for a representative set of four test problems from the Netlib set [5]. The potential for implementing the algorithm on a shared memory multiprocessor is discussed in Section 5 and further conclusions are offered in Section 6.

1.1 Background

The two main variants of the simplex method are the standard simplex method and the revised simplex method. Although early versions of the revised simplex method used an explicit form of the inverse, this was quickly replaced by methods based on a factored form of the inverse. Most important LP problems are large (some with millions of variables and constraints) and sparse (the coefficient matrix has an average of 5–10 non-zeros per column). An important point to note is that for large sparse problems the standard simplex and the explicit inverse form of the revised simplex are completely uncompetitive in speed compared with the revised simplex method when a factored form of the inverse is used.

There have been several studies [3, 14, 17] which have implemented either the standard form of the simplex or the revised simplex with the inverse of the basis matrix stored as a full matrix. Both methods parallelise well but, as noted above, are so bad for large sparse problems, that the results are a lot slower than a good serial implementation.

For the revised simplex method with a factored inverse, attention hitherto has been given to parallelising the individual computational components of the algorithm. Pfefferkorn and Tomlin [15] discuss how all the major operations might be parallelised on an ILLIAC IV, although no implementation was attempted. However, in reports of practical

implementations [12, 16], exploitation of parallelism has been limited to just the PRICE operation and/or overlapping the refactorization of the inverse with simplex iterations, so little or no speed-up was obtained.

An investigation of the extent to which the simplex method can exploit a vector processor was made by Forrest and Tomlin [4]. They report a speed-up of between 1 and 5 for their first twelve problems, and a speed-up of 12 for a further problem.

1.2 The revised simplex method

A linear programming problem has the form

$$\begin{array}{rl} \text{maximize} & f = \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} & \boldsymbol{x} \geq \boldsymbol{0} \\ & A\boldsymbol{x} = \boldsymbol{b} \\ \text{where} & \boldsymbol{x} \in I\!R^n \quad \text{and} \quad \boldsymbol{b} \in I\!R^m. \end{array}$$

At any stage in the simplex method the variables are partitioned into two sets, basic variables $\boldsymbol{x}_B$ and nonbasic variables $\boldsymbol{x}_N$. The set of basic variables is referred to as the basis. If the problem is partitioned correspondingly then the objective function is $f = \boldsymbol{c}_B^T \boldsymbol{x}_B + \boldsymbol{c}_N^T \boldsymbol{x}_N$, the constraints are $B\boldsymbol{x}_B + N\boldsymbol{x}_N = \boldsymbol{b}$ and the basis matrix B is nonsingular. Each basic variable is identified with a particular row of the constraint matrix A, and each nonbasic variable is identified with a particular column of the matrix N. The major computational steps of the revised simplex method are illustrated in Figure 1.

`BTRAN`: Form $\boldsymbol{\pi}^T = \boldsymbol{c}_B^T B^{-1}$.
`PRICE`: Calculate the *reduced costs* $\hat{\boldsymbol{c}}_N^T = \boldsymbol{c}_N^T - \boldsymbol{\pi}^T N$.
`CHUZC`: Scan $\hat{\boldsymbol{c}}_N$ for a variable q with a negative reduced cost.
If no such candidate exists then exit. (Basis is optimal.)
`FTRAN`: Form $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$, where $\boldsymbol{a}_q$ is column q of A.
`CHUZR`: Scan the ratios $\hat{b}_i/\hat{a}_{iq}$ for the row p of a good candidate to leave the basis, where $\hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$. Let $\alpha = \hat{b}_p/\hat{a}_{pq}$.
`UPRHS`: Update right hand side using $\hat{\boldsymbol{b}} := \hat{\boldsymbol{b}} + \alpha\hat{\boldsymbol{a}}_q$.
`If` {growth in factors} `then`
  `INVERT`: Find a factored inverse of B.
`else`
  `UPDATE`: Update the inverse of B corresponding to the basis change.
`endif`

Figure 1: A major iteration of the revised simplex method

At the beginning of an iteration of the revised simplex method it is assumed that a factored inverse of the basis matrix B is available: elementary row or column matrices $M_1, M_2, \ldots, M_r$ are known such that $B^{-1} = M_1 M_2 \ldots M_r$. The first operation is the calculation of the dual variables $\boldsymbol{\pi}^T = \boldsymbol{c}_B^T B^{-1}$ by passing backwards through the factors of B^{-1}, an operation known as BTRAN. This is followed by the PRICE operation, which is a sparse matrix-vector product which yields the reduced costs of the nonbasic variables. These reduced costs are scanned for a negative value in the operation known as CHUZC. Although any variable, q say, with a negative reduced cost can be chosen as the variable to enter the basis, the rule used for selecting among the variables with negative reduced cost can have a major impact on the number of iterations required to solve the problem. The original rule was the *Dantzig* criterion, which selects the variable with the reduced cost. For many problems it is more efficient to use an exact or approximate steepest edge criterion. Exact steepest edge is described by Goldfarb and Reid in [8] and *Devex* approximate steepest edge is described by Harris in [11]. However a discussion of these latter techniques in the parallel context is beyond the scope of this paper.

In order to determine the basic variable which would be replaced by the variable q entering the basis it is necessary to calculate the column of the standard simplex tableau corresponding to q. This *pivotal column* $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$, where $\boldsymbol{a}_q$ is column q of the constraint matrix A, is formed by passing forward through the factors of B^{-1}, an operation known as FTRAN. The row corresponding to the leaving variable is determined by the CHUZR operation which scans the ratios $\hat{b}_i/\hat{a}_{iq}$, where $\hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$ is the vector of current values of the basic variables. Traditionally the leaving variable is the one corresponding to the smallest non-negative ratio, since this ensures that no variable exceeds its bounds after the basis change . However it is preferable for numerical stability to select the row with the largest value of $\hat{a}_{iq}$ among those which lead to bound violations which are sufficiently small. Such 'thick pencil' techniques are described by Harris in [11] and by Gill *et al* in [7].

Once a basis change has occurred, the inverse of the current basis matrix is normally updated rather than recalculated. Various methods are possible for the UPDATE operation. Let the basis invert produced by INVERT be denoted by B_0^{-1} and the inverse k basis changes later by B_k^{-1}. The simplest form of update is the product form update [2], which represents B_k^{-1} as

$$S_k^{-1} \ldots S_1^{-1} B_0^{-1}. \tag{1}$$

The matrix S_j is an elementary column matrix whose only non unit column is simply derived from the vector $\hat{\boldsymbol{a}}_q$ for the variable to enter the basis in

that iteration and is referred to as an **eta vector**. The Bartels-Golub and Forrest-Tomlin updates modify the factors of B_0^{-1} in order to reduce the size of the factored inverse of B_k. Update procedures based on the use of a Schur complement are described by Gill *et al* in [6].

1.3 Parallelising the revised simplex method with a factored inverse

For the conventional revised simplex method as illustrated in Figure 1, each of the major computational steps has to be completed before the following step can start. In the revised simplex method on a serial machine, INVERT takes typically 10% of the time, so there is very limited scope for speed-up in using one process to perform INVERT and one other to perform the rest of the algorithm in parallel. This approach was taken in one of the experiments reported by Ho and Sundarraj [12], but no worthwhile speed-up was obtained.

It follows that any real exploitation of parallelism in the method as given in Figure 1 is limited to the parallelisation of the of the individual computational steps. For sparse problems, it is easy to parallelise PRICE, CHUZC and CHUZR. Shu and Wu [16], and Ho and Sundarraj [12] report experiments in which the PRICE operation is parallelised but achieve little or no speed-up over their serial implementation. The parallelisation of PRICE within the dual simplex algorithm has formed the basis of recent work by Bixby and Martin [1].

In contrast to the simple techniques required to parallelise PRICE, CHUZC and CHUZR, the parallelism which may be exploited within BTRAN, FTRAN and INVERT is limited, very fine grained and hard to achieve. This was identified by Pfefferkorn and Tomlin [15]. This suggests that algorithms which are genuinely parallel variants of the revised simplex method should be considered. Subsequent to the ASYNPLEX algorithm presented in this paper, experiments with alternative parallel algorithms have been reported by Wunderling [18] (fully parallel for only two processors) and by Hall and McKinnon [10].

2 ASYNPLEX, an asynchronous parallel algorithm

Since the parallelisation of all individual steps of the revised simplex method is limited and very hard to achieve, it is important to consider how the

method itself can be modified to allow the maximum degree of independence between the computational steps in different iterations. However, it is also essential that any algorithm performs INVERT in parallel with simplex iterations, otherwise INVERT will then become the dominant step and limit the possible speed-up.

The ASYNPLEX algorithm performs serial simplex iterations but overlaps them to the maximum extent possible using a technique described in Section 2.1. The algorithm also performs INVERT in parallel with simplex iterations. This means that the basis matrix whose factored inverse is formed is out-of-date when INVERT is completed. The issue of bringing the new factored inverse up-to-date with minimal overhead, and the consequences for numerical stability are discussed in Section 2.2.

Further algorithmic refinements which ensure that the overlapping simplex iterations determine a single path on the surface of the feasible region and duplicated work is reduced to a minimum are discussed in Section 2.3. The ASYNPLEX algorithm is also presented formally as pseudocode suitable for implementation on a distributed memory machine. The minor modifications to the algorithm for a shared memory implementation are discussed briefly.

Note that although the sequence of computational steps is performed differently, ASYNPLEX may be viewed as a variant of the revised simplex method since it corresponds to a particular column selection rule. Since the variables are only allowed to enter the basis if they have negative reduced cost, ASYNPLEX inherits the termination properties of the simplex method.

2.1 Overlapping simplex iterations

In the simplex method, there are usually several variables with a negative reduced cost, and a valid simplex iteration will occur if any of these is chosen to enter the basis. It is common, especially in very sparse problems, for variables with negative reduced costs in one iteration to have negative reduced costs for a number of subsequent iterations. It is this *candidate persistence* which is exploited by the ASYNPLEX algorithm to allow simplex iterations to be overlapped.

Rather than waiting for the reduced costs for the current basis to be calculated, an attractive candidate is selected from the most up-to-date reduced costs yet formed and the FTRAN operation is started, allowing it to overlap with calculations from previous iterations. Provided the up-to-date reduced cost is calculated and found to be negative before that variable is finally allowed to enter the basis, a valid simplex iteration will occur.

Once the updated pivotal column $\hat{\boldsymbol{a}}_q$ for the tentative candidate q has

been calculated, the updated reduced cost can be calculated as follows. By combining the BTRAN and PRICE steps (see Figure 1) it is seen that the reduced costs may be expressed as $\hat{\boldsymbol{c}}_N^T = \boldsymbol{c}_N^T - \boldsymbol{c}_B^T(B^{-1}N)$. Since $\hat{\boldsymbol{a}}_q$ is the column of $B^{-1}N$ corresponding to variable q, the reduced cost for variable q is given by

$$\hat{c_q} = c_q - \boldsymbol{c}_B^T \hat{\boldsymbol{a}}_q.$$

This operation involves a single inner product within which any sparsity is readily exploited.

If $\hat{c_q} \geq 0$, the variable q no longer has a negative reduced cost and is not chosen to enter the basis. In this case the work done in performing the FTRAN has been wasted. Also, even when q is accepted to enter the basis, its true reduced cost is likely to be poorer than the best available using up-to-date information, and this is likely to lead to an increase in the number of iterations taken to solve the problem. As a result, it is important to minimise the extent to which the reduced costs are out-of-date. This is achieved by performing the BTRAN and PRICE operations after every basis change. With this strategy it is natural to use the same process to do this as was used to perform the CHUZR leading to that basis change, as this process is guaranteed to be free at that point.

2.2 Overlapping INVERT

As observed above, since the particular basis which is reinverted is generally unimportant, it follows that INVERT can be performed in parallel to, and independent of, any simplex iterations. Since each UPDATE operation generally increases the cost of using the factored inverse in subsequent FTRAN and BTRAN operations, it follows that if the cost of accommodating a new factored inverse is relatively small and sufficient processors are available that one can be dedicated to INVERT, then there is no upper limit on the desirable reinversion frequency.

When INVERT is overlapped with simplex iterations, the major issues which need to be addressed are how to bring the factored inverse up-to-date with respect to basis changes which have been determined since the start of INVERT, how efficiently this may be achieved and the impact on numerical stability.

Recovering an up-to-date factored inverse after INVERT

The issue of recovering an up-to-date factored inverse following reinversion is discussed by Hall and McKinnon in [9] where it is concluded that the use of Bartels-Golub or Forrest-Tomlin updates is inappropriate. With these

update methods it is expensive to incorporate the basis changes which have occurred during INVERT and the updates alter the original factors, so it is more difficult to share the factored invert between different processes. However, when product form or Schur complement update procedures are used, the inverse can be brought up-to-date in an efficient manner. The product form update is particularly simple and it is this approach which is used in ASYNPLEX.

The procedure used in ASYNPLEX to update the factors produced by a reinversion to correspond to the current basis is as follows. Suppose that the INVERT which produced the factors currently incorporated into the basis factors started at basis r, that a new INVERT started at basis s and becomes available for use when basis t is current. Let the factors after the INVERTs be denoted by B_r^{-1} and B_s^{-1}, and the product form update corresponding to the basis changes from basis r to s and from s to t be denoted by $U_{s\leftarrow r}^{-1}$ and $U_{t\leftarrow s}^{-1}$ respectively. Before the factors from the new INVERT are used, the factors for basis t have the form $U_{t\leftarrow s}^{-1}U_{s\leftarrow r}^{-1}B_r^{-1}$. It is normal in implementations of the revised simplex method to allow the reinversion procedure to permute (implicitly) the columns of the basis matrix so as to obtain sparser factors. A separate index is kept which records the variable which is solved for in each row after FTRAN. After each basis change this index is updated. After reinversion, the new factors will solve for the same variables as before but usually in a different order. Thus the the previous order can be recovered by permuting the solution produced by the new factorization. This may be expressed as a permutation P_s such that $U_{s\leftarrow r}^{-1}B_r^{-1} = P_sB_s^{-1}$ from which it follows that the current factors $U_{t\leftarrow s}^{-1}U_{s\leftarrow r}^{-1}B_r^{-1}$ are entirely equivalent to $U_{t\leftarrow s}^{-1}P_sB_s^{-1}$.

There are three ways to accommodate this permutation. It could be left where it is in the algebraic expression for the current factored inverse and applied in the middle of every BTRAN and FTRAN. This is unattractive because in both cases a full vector has to be reordered at a stage when it would not otherwise be accessed. Alternatively the permutation P_s may be applied symmetrically to either $U_{t\leftarrow s}^{-1}$ or B_s^{-1} to give

$$\begin{aligned} U_{t\leftarrow s}^{-1}P_sB_s^{-1} &= P_s(P_s^{-1}U_{t\leftarrow s}^{-1}P_s)B_s^{-1} \\ \text{or}\quad U_{t\leftarrow s}^{-1}P_sB_s^{-1} &= U_{t\leftarrow s}^{-1}(P_sB_s^{-1}P_s^{-1})P_s. \end{aligned}$$

The disadvantage of permuting the $U_{t\leftarrow s}^{-1}$ factors is that this information is distributed in the local memory (or cache) of several processors so the delay caused by this permutation affects all processors. The alternative of permuting the B_s^{-1} factors, can be done once by the INVERT process and so is preferred. This is what is implemented in ASYNPLEX. Note that the

permutation must be applied symmetrically to each of the elementary factors of B_s^{-1}, otherwise the sparsity of the representation is lost. Apart from this permutation of the new factors, each (usually sparse) right-hand-side vector $\boldsymbol{a}_q$ for FTRAN and each output $\boldsymbol{\pi}$ from BTRAN has to be permuted. However this is done as a stage where these vectors are naturally being processed so it is not a large overhead.

Permuting the factors of B_s^{-1} is achieved in two stages. First the permutation itself is determined by scattering the list of what variable is solved for in each row before INVERT and then gathering it using the corresponding list after INVERT at a total cost of $2m$ integer operations. Applying the symmetric permutation to the factored form of B_s^{-1} requires each of its constituent indices to be permuted. Since this operation does not require each of the elementary factors to be treated in a separate loop and no floating-point operations are required, the total cost of applying the permutation is significantly less than that which would be incurred by a single BTRAN applied to a full right-hand-side.

Once the new (permuted) inverse factors are available, all that is then involved in bringing the new factors up to date is to discard those current factors corresponding to basis changes which occurred before the recent INVERT was started, and attach the remaining current factors to those produced by INVERT. Provided the BTRAN and FTRAN implementation can deal with factors in separate blocks, this change can be implemented by a few changes of pointers and does not require the factors to be moved.

Impact on numerical stability

The efficiency of using the product form update comes at a cost in terms of numerical stability compared to Bartels-Golub, Forrest-Tomlin or Schur complement updates. If the basis at any stage is ill-conditioned, then factors will be inaccurate. With product form updates the factors will remain inaccurate, whereas with the other update methods it is possible for the factors to regain accuracy when the basis becomes less ill-conditioned. During reinversion, the ability to select pivots on numerical grounds means that a stable representation of the basis being factorised can be obtained. If the values of the basic variables and reduced costs are also recalculated then in an implementation where the invert does not overlap the iterations, subsequent simplex iterations may be viewed as having a numerical 'fresh start'. This does not occur in the case where the invert is overlapped as the update factors $U_{t\leftarrow s}^{-1}$ are re-used without change. These update factors result from FTRANs performed using factors from a previous inverse and, if these are inaccurate, the update factors are likely also to be inaccurate. Thus the

```
   Repeat
V1:    While  {basis changes received←I8 not yet incorporated in list}
          Update list of basic variables
       end while
       INVERT; Permute factors
V2:    Send→I1 on all iteration processes the new factored inverse
   until {Simplex algorithm terminates}
```

Figure 2: ASYNPLEX: Algorithm for the invert processor

numerical cleansing resulting from reinversion is not so effective in the parallel case. The 'thick pencil' row selection techniques described by Harris in [11] and by Gill *et al* in [7], reduce the incidence of small pivots and so are particularly valuable in the context of parallel algorithms in which INVERT and simplex iterations are performed concurrently. Our implementation of ASYNPLEX uses a modification of the EXPAND technique of Gill *et al* in which, for reasons given in the following subsection, does not further expand the feasible region each iteration.

2.3 The ASYNPLEX algorithm

The ASYNPLEX algorithm is defined and discussed here in terms of the operation of four types of process, an *iteration process*, and *invert process*, a *column selection manager process* and a *basis change manager process*. There must be $p \geq 1$ iteration processes and one each of the other process types. The iteration processes perform all the major computational steps of the method with the exception of INVERT, which is done by the invert process. Coordination among the iteration processes is achieved by the basis change and column selection manager process. The roles of these processes are discussed below. Detailed pseudo-code for the operations performed by each of the process types is given in Figures 2–5.

One goal of the implementation is to allow processes to operate in an asynchronous manner as independently of each other as possible. A result of this asynchronous execution is that different processes can be working with different bases at the same time. Each iteration process i must therefore keep its own record of the basis in use by that process, and this is done by a basis counter k_i. This is incremented whenever iteration process i changes the basis after its own CHUZR, or whenever it incorporates basis changes made by other iteration processes into its own data structures. In both cases this operation is referred to as UPDATE_BASIS and consists of updating the values of the basic variables (referred to as UPRHS in Figure 1) and the

```
    k_i = 0
    BTRAN
    PRICE
    Let q be the i^th most attractive candidate
    Repeat
I1:     If {Received←V2 a new factored inverse}
            Install new factored inverse
I2:     While  {basis changes received←I7 not yet applied}
            APPLY_BASIS_CHANGE; k_i := k_i + 1
        end while
        Permute column a_q; FTRAN
     1  Continue
I3:     While  {basis changes received←I7 are not yet applied}
            APPLY_BASIS_CHANGE; FTRAN_STEP; k_i := k_i + 1
        end while
        If {ĉ_q > 0}  then
I4:         Send→C4 a message that the candidate is unattractive
        else
I5:         Send→R1 an offer to perform CHUZR
I6:         Wait←(R2 or R3) for a reply to offer
            If {Offer accepted} then
                CHUZR
I7:             Send→(I2 or I3 or I10 on all other iteration processes)
                the basis change and pivotal column
I8:             Send→(V1 and C1) basis change
                UPDATE_BASIS; k_i := k_i + 1
                BTRAN; Permute π
                PRICE
                Choose a set of the most attractive candidates
I9:             Send→C2 the most attractive candidates
            else
I10:            Wait to receive←I7 next basis change
                goto 1
            end if
        end if
I11:    Wait to receive←(C3 or C5) a new candidate column, q
    until {Simplex algorithm terminates}
```

Figure 3: ASYNPLEX: Algorithm for iteration process i

```
    k_c = 0
    Mark all nonbasic variables as un-selected
    Repeat
C1:     If {Received←I8 basis change} then
            Mark the variables which has left the basis as unselected
C2:     else if {Received←I9:i a set of candidates
                    corresponding to basis k_i} then
            If {k_i > k_c} then
                Filter out the candidates already selected and those
                already rejected after the FTRAN at a basis ≥ k_i
                k_c = k_i
            endif
C3:         Send→I11:i the most attractive candidate to enter the basis and
            mark the candidate as selected
C4:     else if {Received←I4:i a message that its current
                    candidate is now unattractive} then
C5:         Send→I11:i the most attractive candidate to enter the basis and
            mark the candidate as selected
        endif
    until {Simplex algorithm terminates}
```

Figure 4: ASYNPLEX: Algorithm for the column selection manager

```
    k_b = 1
    Repeat
R1:     If {Received←I5:i an offer to perform CHUZR
            corresponding to basis k_i} then
            If {k_i = k_b} then
R2:             Send→I6:i an acceptance of the offer
                k_b = k_b + 1
            else
R3:             Send→I6:i a refusal of the offer
            endif
        endif
    until {Simplex algorithm terminates}
```

Figure 5: ASYNPLEX: Algorithm for the basis change manager

relatively small amount of index modification required to add the new eta to the factored inverse and change the status of the variables entering and leaving the basis.

The column selection manager

The column selection manager maintains a pool of candidates from which it supplies iteration processes when they become idle. The column selection manager keeps track of those candidates which have been sent and those which have been rejected due to having a positive reduced cost once their pivotal column is up-to-date, together with the basis number when this occurred. When the column selection manager receives a set of attractive candidates from an iteration process (after that process has performed CHUZC), so long as this set of candidates is more up-to-date than those which form the current pool, it forms a new pool. This is done by filtering out those candidates which have been sent to other iteration processes (but which have not yet entered the basis or been rejected), and those candidates which have been rejected due to being found unattractive at a more up-to-date basis.

The basis change manager

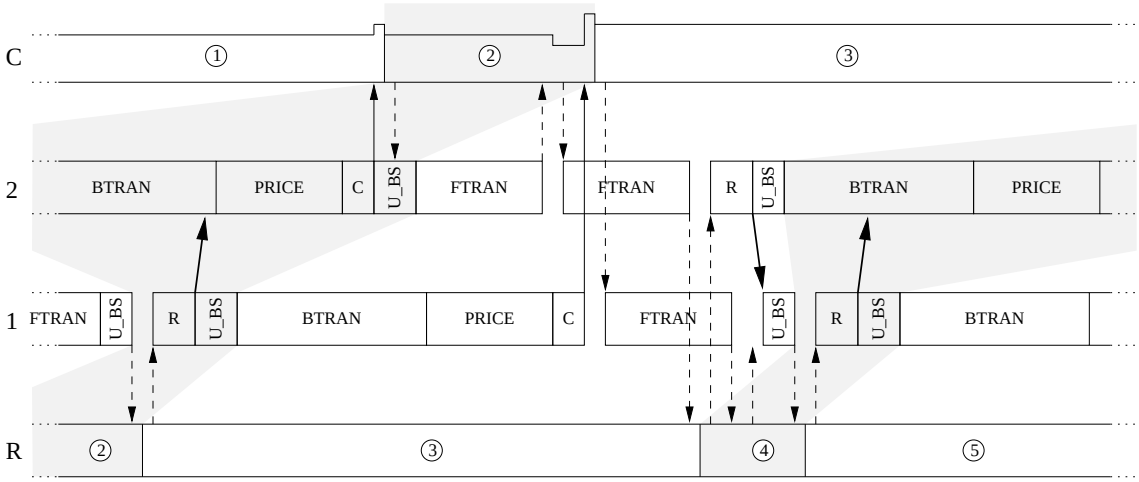
The basis change manager ensures that no two iteration processes perform CHUZR for the same basis and, hence, that the algorithm follows a single path on the surface of the feasible region. Whenever a iteration process has no more basis changes to make and is ready to start a CHUZR (which will then lead to a basis change), it sends its current basis number k_i to the basis change manager. The basis change manager maintains a record k_b of the basis number of the last basis which it has given permission be created. If $k_i < k_b$ then iteration process i has not yet applied a pending basis change and the request is refused. In this case the iteration process waits until it receives a basis change message from another process before making a further request to change the basis. If $k_i = k_b$ then iteration process i is up to date. Permission is given for it to make the next basis change and k_b is incremented by one.

Operation of the column selection and basis change manager

The action of the column selection and basis change managers in relation to typical events on two iteration processes is illustrated by the Gantt chart in Figure 6. Arrows indicate communication between processes, with dashed arrows representing the communication of purely logical information (a few

bytes), solid arrows representing the communication of a small amount of information (a few tens of bytes) and the solid arrows representing the communication of a larger amounts of information (of the order of m bytes, where m is the number of rows in the LP).

Note that this illustration is not to scale in that the time required for communication and the length of some of the shorter operations is exaggerated for reasons of clarity. An example Gantt chart for true processor activity in the T3D implementation described in Section 3 is given in Figure 7.



C: CHUZC R: CHUZR U_BS: UPDATE_BASIS

Figure 6: Column selection manager (C), basis change manager (R) and two iteration processes (not to scale)

The numbered sections on the Gantt chart indicate the (relative) basis count of each of the four processes. The uppermost bar in the Gantt chart corresponds to the column selection manager and the height of this bar represents the number of attractive candidates in the pool. When the set of candidates arrives following CHUZC on iteration process 2, the number of attractive candidates increases. Note that the size of the pool is not guaranteed to increase since when a new set of candidates arrives. This can be due to a reduction in the total number of attractive candidates (as optimality is reached) or due to the removal of candidates which have been sent to iteration processes.

Iteration process 1 begins the time slice illustrated in Figure 6 by completing FTRAN. It then updates the basis and the pivotal column with respect to the basis change determined during FTRAN as a result of a CHUZR on iteration process 2 (not shown). The candidate remains attractive so

iteration process 2 requests the right to change basis 2. Since $k_b = 2$, the basis change manager has not yet given permission for basis 3 to be formed, so this request is granted. Once iteration process 1 has performed CHUZR and communicated the pivotal column (and index of the pivotal row) to iteration process 2, it updates its own basis and starts calculating the reduced costs for basis 3.

Meanwhile, the basis update operation on process 2 corresponding to the basis change determined by iteration process 1 is being overlapped with sending the list of attractive candidates to the column selection manager and receiving an individual attractive candidate in return. The number of attractive candidates in the pool is seen to be reduced by one prior to this latter communication. Following the FTRAN operation on this candidate, it is discovered that, as a result of the change from basis 2 to basis 3, the candidate is no longer attractive. Iteration process 2 requests a new candidate from the column selection manager and is idle until the candidate arrives. Upon completion of the subsequent FTRAN, the candidate is found to be still attractive so iteration processor 2 sends a request to change basis 3. Since the basis change manager has not yet received a request to change basis 3, the request from iteration process 2 is granted and it starts CHUZR.

At this stage, both iteration processors are at almost the same stage in a simplex iteration and the necessity for a basis change manager becomes evident. Iteration process 1 brings its pivotal column up-to-date with respect to basis changes already performed *before* receiving the pivotal column and pivotal row index from iteration process 2, indicating that a further basis has occurred. Iteration process 1 requests permission to change basis 3 but this is refused since $k_b = 4$, indicating that the basis change manager has given permission for another iteration process to change basis 3. This refusal thus prevents CHUZR from being started by more than one process for the same basis. Iteration process 1 is then idle, not only for the time required for the communication to and from the basis change manager, but also for the time until the pivotal column and pivotal row index determined by iteration process 2 have arrived.

Limit on parallel performance

The need to ensure that basis changes do not happen simultaneously leads to an upper limit on the iteration frequency of ASYNPLEX. This limiting iteration frequency is the reciprocal of the minimum time between basis changes which, in turn is the (typical) sum of the time required to communicate a pivotal column and pivotal row index, the time required by UPDATE_BASIS, the time required for communication to and from the basis

change manager and the time required for CHUZR. Considerable effort must be put into implementing these four operations efficiently lest they result in a serious bottleneck in the parallel performance.

Duplicated work

The only work in ASYNPLEX which is duplicated across all iteration processes is that required by UPDATE_BASIS in Figure 3. When a non-zero step is made, the majority of the cost of UPDATE_BASIS is incurred in updating the values of the basic variables.

In many linear programming problems, a significant proportion of the iterations are degenerate. When the traditional ratio test is used, this results in a zero step for which, of course, there is no need to update the right-hand-side. However, the EXPAND technique for CHUZR of Gill *et al* reduces the possibility of cycling or stalling in the presence of degeneracy by ensuring that zero steps are not made. This is achieved by expanding the feasible region a small amount each iteration, so guaranteeing that (at least) a small positive step is made. Unfortunately this means that the right-hand-side must be updated after every basis change: a significant amount of duplicated work in the context of ASYNPLEX. Since we place less value on this anti-degeneracy procedure than the improved numerical stability properties afforded by the 'thick pencil' row selection possibilities, we choose not to expand the feasible region each iteration and so obtain a significant reduction in the number of small nonzero steps made.

3 Implementation on a Cray T3D

An implementation of ASYNPLEX has been written for the Cray T3D based at Edinburgh University. This machine has the very high ratio of communication to computation speed which is necessary to achieve speed-up when using ASYNPLEX on a distributed memory machine.

In addition to an implementation of the widely-used message-passing routines MPI and PVM, the Cray T3D has a suite of inter-processor communication routines known as SHMEM. These virtual shared memory routines allow one processor to write to, or read from, an address on any other processor with no significant impact on the speed of any computation which is being performed on that processor. In particular, the `shmem_put` routine enables data to be written to another processor with a latency of fewer than ten microseconds and at a bandwidth of 120MB/s. By contrast, messages sent using PVM or MPI have a latency of tens of microseconds

and a bandwidth of 40-60MB/s. Despite the non-portability of the SHMEM routines and the fact that they have to be used with much greater care than PVM or MPI routines, initial experiments dictated that all communications in the implementation of ASYNPLEX be performed using `shmem_put`.

The T3D processing elements are Dec Alpha chips with 300Mflop peak performance. However, this computational performance is not approached for large sparse linear programming problems because of the the indirect addressing of arrays which the solution of these problems involves.

The implementation ASYNPLEX was developed using modules of our serial code ERGOL. The INVERT module of ERGOL compromises the reduction of fill-in and numerical stability for speed. This balance of properties is important when implementing ASYNPLEX. A fast INVERT increases the frequency with which new factored bases become available and so reduces the number of update etas which must be applied, thus increasing the speed of FTRAN and BTRAN. In practice, the fill-in generated by this INVERT is not significantly greater than is obtained by procedures based on the Markowitz criterion, which preserve sparsity particularly well.

Communication

When ASYNPLEX is implemented on a distributed memory machine, there are four main communication requirements. In only two of these is the volume of communication related to the dimension of the problem. However these messages can be overlapped with computation. The other communications involve no more than a few tens of bytes but are frequent and not all can be overlapped with computation. As a result, the algorithm must be implemented using message-passing routines with very low latency.

Each factored basis must be broadcast from the invert processor to each of the iteration processors. This is by far the largest single communication but, as with INVERT itself, it can be overlapped with simplex iterations.

The second largest communication in terms of volume is the broadcast of the pivotal column from the iteration processor which determines the corresponding basis change to the other iteration processors. In our implementation, the values and row indices of the nonzero entries (determined during the pass through the pivotal column in CHUZR) are written directly to their final destination in the factored inverse of the basis on each other iteration processor. This is overlapped by computation, except when the receiving processor is idle because all previous updates have already taken place. Once the values of the basic variables have been updated (if necessary), all that is required to update the factored inverse is to calculate the reciprocal of the pivot, store the index of the row in which it occurs and

increase the eta count by one. This virtually 'free' UPDATE operation is only possible when using the product form. Thus the number of passes through all the entries in the pivotal column is reduced to a minimum. As identified above, this is important since each UPDATE_BASIS operation is duplicated over all iteration processes.

Once an iteration processor has performed BTRAN and PRICE, it chooses a set of good candidates to send to the column selection manager. In practice the number of candidates chosen is seldom more than ten so the volume of this communication is not significant. Before the processor can start FTRAN for a new candidate, it must wait until its set of good candidates has been received by the column selection manager and a new candidate has been identified and communicated back. Once again this is a communication of insignificant volume but the iteration processor may be idle for at least twice the latency period for a single communication. A similar idle period will always occur after a pivotal column has been brought up-to-date. This is due to the logical send-and-receive with the row selection manager which is required to determine whether the iteration processor can proceed with CHUZR.

4 Computational experiments

Computational experiments on the Cray T3D were performed for four representative problems from the Netlib [5] test set. These test problems are discussed in Section 4.1 and the results of the experiments are given in Section 4.2

4.1 Test problems

The names and the number of rows, columns and nonzeros in the constraint matrix for the LP problems selected for the numerical experiments are given in the following table.

Problem	Rows	Columns	Nonzeros
SHELL	536	1775	3556
SCTAP3	1480	2480	8874
25FV47	821	1571	10400
GREENBEB	2382	5405	30877

Although these problems are small by modern practical standards, they are representative of the problems in the Netlib test set and have neither a common structure nor extreme relative dimensions.

SHELL is a particularly sparse problem for which the basis matrix can always be reordered into triangular form and, even at the optimal basis, pivotal columns are 2% full on average—corresponding to just ten non-zero entries. It is also a problem for which candidate persistence is known to be good. The number of iterations required to solve the problem is not affected significantly by the column selection strategy, the Dantzig, Devex and steepest edge finding the optimal solution in 774, 708 and 715 iterations respectively (using ERGOL). As a result, column selection using out-of-date reduced costs is not expected to have a significant effect on the number of iterations required to solve the problem. SCTAP3 is a larger problem which exhibits similar low fill-in but is a little more sensitive to the choice of column selection criterion.

Although of dimensions which are comparable to those of SHELL and with only three times as many nonzeros, 25FV47 is a problem for which fill-in is significant. The factored inverse of the optimal basis has 66% more nonzeros than the matrix itself and pivotal columns at the optimal basis are 58% full. GREENBEB was chosen to represent the larger problems in the Netlib set. The factored inverse of the optimal basis has 23% more nonzeros than the matrix itself and pivotal columns at the optimal basis are 24% full.

4.2 Results

A stated aim in the Introduction was that a parallel algorithm should be significantly faster than good serial simplex solvers. Although no comparison with a commercial solver is possible on the T3D, Table 1 gives a comparison of ERGOL and OSL Version 1.2 [13] for the four test problems. The solution times given are the CPU time required on a SUN SPARCstation 5, starting from a logical basis and using the Dantzig strategy in CHUZC.

Problem	Solution time (s)		Simplex iterations		Iteration frequency (s^{-1})	
	ERGOL	OSL	ERGOL	OSL	ERGOL	OSL
SHELL	3.25	3.79	774	751	238	198
SCTAP3	12.3	21.7	1630	1677	133	77
25FV47	95.8	62.7	4922	4164	51	66
GREENBEB	358.	424.	11636	11869	33	28

Table 1: Comparison of ERGOL and OSL

The results for the same four problems using ASYNPLEX on the T3D are presented in Tables 2-5. In each case the problem was scaled and then solved

from a logical basis. The first column in each table gives the number of iteration processors, with zero corresponding to ERGOL running on one processor. The number of simplex iterations required to solve the problem is given in the second column. Column three gives the total number of candidates which prove to be unattractive when their pivotal column is brought up-to-date. The final two pairs of columns give the iteration frequency and solution time, together with the respective speed-up compared with the serial implementation. The total solution time is the maximum elapsed time on any processor. The number of iteration processors used was increased in unit steps up to six and then in steps of two until either no further speed-up in solution time could be achieved or a maximum total number of sixteen processors (including the two manager processors) was reached, this being the limit on the number of processors which could be used interactively.

Iteration processors	Simplex iterations	Unattractive candidates	Iteration frequency (s^{-1})	Speed-up	Solution time (s)	Speed-up
0	774	-	260	-	3.0	-
1	759	0	210	0.8	3.6	0.8
2	772	198	370	1.4	2.1	1.4
3	766	312	510	2.0	1.5	2.0
4	742	341	620	2.4	1.2	2.5
5	762	469	820	3.2	0.93	3.2
6	803	421	940	3.7	0.85	3.5
8	814	779	1000	4.0	0.79	3.8
10	761	601	1200	4.5	0.66	4.5
12	783	977	1300	4.9	0.62	4.8

Table 2: Results for SHELL using ASYNPLEX

The difference in the number of simplex iterations when ERGOL is used (zero iteration processors) and the case where just one processor performs simplex iterations is explained by the fact that different basis matrices are inverted, leading to different rounding and hence differences in both column selection (when reduced costs would be equal using exact arithmetic) and row selection when the vertex is degenerate.

For SHELL and SCTAP3 there is no gain in efficiency by performing INVERT in parallel with just one simplex iteration processor. However there is some speed-up for 25FV47 and GREENBEB. This is partly because INVERTs are relatively more expensive for these problems but mainly due to the significant reduction in the average time for FTRAN and BTRAN when new factored inverses are available more frequently.

Iteration processors	Simplex iterations	Unattractive candidates	Iteration frequency (s^{-1})	Speed-up	Solution time (s)	Speed-up
0	1681	-	110	-	15.8	-
1	1589	0	110	1.0	15.0	1.1
2	1949	520	190	1.8	10.4	1.5
3	2021	1005	260	2.5	7.7	2.1
4	2119	1387	320	3.0	6.6	2.4
5	2299	1410	360	3.4	6.4	2.5
6	2390	1876	400	3.7	6.0	2.6
8	2333	2418	470	4.4	5.0	3.2
10	2453	2843	500	4.7	4.9	3.2
12	2363	2771	530	4.9	4.5	3.5

Table 3: Results for SCTAP3 using ASYNPLEX

Iteration processors	Simplex iterations	Unattractive candidates	Iteration frequency (s^{-1})	Speed-up	Solution time (s)	Speed-up
0	5003	-	48	-	107	-
1	5263	0	71	1.5	74	1.4
2	6561	3247	110	2.3	62	1.7
3	7111	5542	140	3.0	51	2.1
4	8171	8825	160	3.4	52	2.1
5	8245	10710	180	3.8	46	2.3
6	7678	12082	190	4.1	40	2.7
8	9279	16760	210	4.5	44	2.4

Table 4: Results for 25FV47 using ASYNPLEX

In each case the iteration frequency increases steadily with the number of iteration processors and a speed-up of between 4.0 and 4.5 is achieved when eight processors are used. This indicates that the time per iteration on each iteration processor has increased by a factor of up to two. This is partly explained by the increasing time spent bringing unattractive pivotal columns up-to date and applying basis changes determined on other processors, and partly by an increase in the time required to perform BTRAN and FTRAN as the number of simplex iterations between reinversion increases. Finally, note that the speed-up in the solution time will be less than the speed-up in iteration frequency if the use of out-of-date reduced costs increases the number of iterations taken.

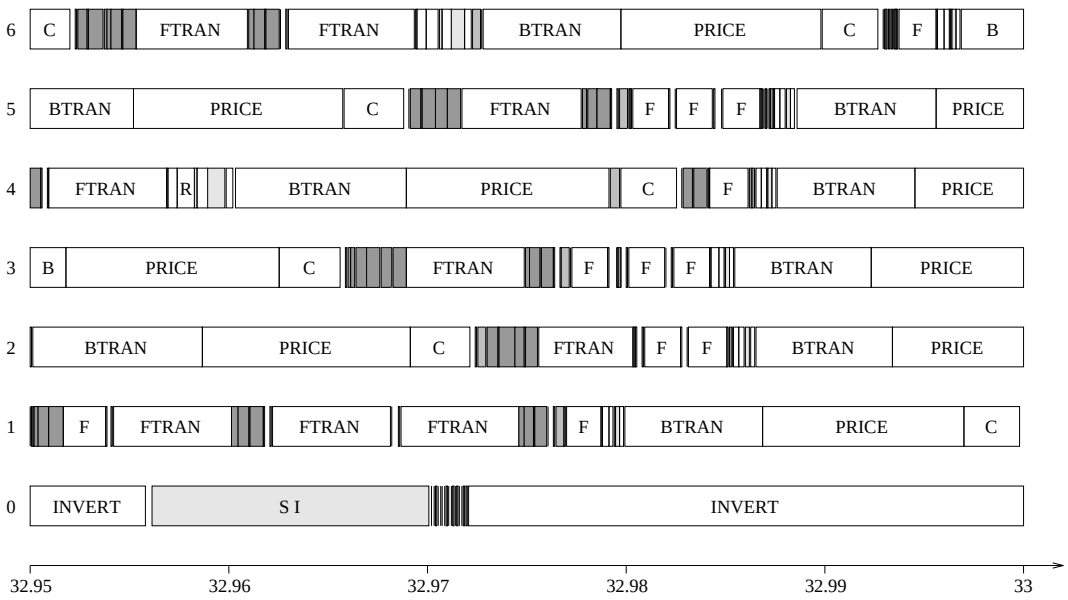
In the results for SHELL given in Table 2, it is seen that the number of iterations does not increase with the number of iteration processors. As a

Iteration processors	Simplex iterations	Unattractive candidates	Iteration frequency (s^{-1})	Speed-up	Solution time (s)	Speed-up
0	11821	-	31	-	384	-
1	11914	0	37	1.2	319	1.2
2	14679	5852	64	2.1	231	1.7
3	15883	10560	82	2.7	193	2.0
4	16848	14966	98	3.2	171	2.2
5	15444	16898	110	3.7	137	2.8
6	17692	21057	120	4.0	144	2.7
8	20556	28532	130	4.3	155	2.5
10	18077	29873	150	4.9	119	3.2

Table 5: Results for GREENBEB using ASYNPLEX

result, the speed-up in the solution time is similar to that for the iteration frequency. For SCTAP3 there is some increase in the number of simplex iterations so the efficiency of using several iteration processors is not so marked. The practical performance of the parallel solver when applied to 25FV47 is less good than with the first two problems. The number of iterations and unattractive candidates increases significantly, limiting the speed-up that can be achieved. Although the candidate persistence for GREENBEB is poor, FTRAN is relatively inexpensive so significant speed-up is still achieved.

An illustration of typical processor activity when solving LP problems is given in the Gantt chart illustrated in Figure 7, the data for which comes from approximately three-quarters of the way through the solution of GREENBEB. Processor activity is seen to be almost continuous, although some of this is spent calculating pivotal columns for candidates which prove to be unattractive and may be identified by FTRANs without a subsequent BTRAN. Note that CHUZR is normally so fast that only one is actually identified in the chart. Lightly-shaded boxes correspond to time associated with communication and, with the exception of the time spent broadcasting the new factored inverse, is very low and only visible in one band for processors 4 and 6. Note that the average time required by INVERT for this problem is $0.13s$, approximately ten times the time taken to broadcast the factored inverse in this illustration. However, the work of updating with respect to simplex iterations determined on other processors is duplicated and appears in heavily-shaded boxes. Once the new factored inverse is received, it is clear that the cost of performing FTRAN is immediately reduced.



C: CHUZC F: FTRAN R: CHUZR SI: Send new factored inverse

Figure 7: Processor activity when solving GREENBEB

5 Potential for implementation on a shared memory multiprocessor

Although ASYNPLEX was described in Section 2 for a distributed memory machine with communication between processes defined in terms of message passing instructions, the underlying algorithm is appropriate for shared memory multiprocessors (SMPs). The details of the transfers of data between processors would not be needed in such an implementation as these transfers would be done automatically by the hardware. The main algorithmic difference would be that the work of updating the right hand side in the UPDATE_BASIS step would be eliminated. This would be at the expense of having to transfer the current values of the right hand side into the cache of the processor about to start the next CHUZR.

In the revised simplex method the ratio of number of memory accesses to number of arithmetic operations is very high. It is therefore very important that the data needed by a processor is available in its local memory or cache. In our Cray T3D distributed implementation, this is achieved by processors sending data when they generate it to the local memory of other processors which will need this data later. The transfer is done by the very fast SHMEM 'put' operation. This has the advantage that the data transfer can be fully

overlapped with calculation. In contrast, on a shared memory machine the transfer of data to the cache of the local processor is done by hardware control. This makes a SMP much easier to program, however the resulting performance is more difficult to predict. Normally on a shared memory machine, when one processor changes data, the transfer of this changed data to another processor's cache occurs when that processor accesses the changed data. Thus the transfer is not overlapped with the calculation. There is also a danger that several processors will require access simultaneously to changed data, causing a bottleneck in transferring it into their caches.

In our T3D implementation the entire data for the problem is duplicated on each processor. Since all the data must be stored in main memory, which is 64MB for each processor, this places a limit on the size of problem which could be tackled. This would not be a restriction on a shared memory machine. However the performance on such a machine would degrade unless its caches were large enough to hold the data which are accessed each iteration, which are the majority of the data.

The actions of the column selection and basis change managers are still required on a SMP. However they would be much simpler to implement. The Cray T3D allows only a single process per processor, so the managers take up two processors in the T3D implementation, even although the work they do is negligible. This could be avoided by doing the managers' tasks on different iteration processors and using locks to ensure that no two iteration processors were doing column or basis change manager tasks simultaneously. This can be done with negligible loss of speed, however it would be more difficult to program than using dedicated processors. On a shared memory machine it is normal to have multiple processes or threads on a single processor, and this would make it very easy to implement the manager processes.

6 Conclusions

A parallel algorithm for the revised simplex method has been described and practical results demonstrating speed-up of between 2.5 and 4.8 have been given. These results could be improved by further performance optimization. However, the limitations of the algorithm should also be addressed. The Dantzig column selection criterion is not often best and the reduced costs are calculated from scratch each time. If the reduced costs are updated then the PRICE operation can be considerably cheaper for sparse problems such as SHELL. This is because the vector for PRICE is $\boldsymbol{e}_p^T B_k^{-1}$ which is generally rather more sparse than $\boldsymbol{c}_B^T B_k^{-1}$.

An algorithmic development which leads from this observation is to

maintain reduced costs by updating them on one or more processors. Steepest edge weights for column selection could be updated on dedicated processors in a similar manner, allowing the expected total number of simplex iterations to be reduced.

Although most of the communication overhead has been minimized by the use of SHMEM routines, the cost of broadcasting the new factored inverse is still significant. This factored inverse may only be applied a few times on each processor and it may prove more efficient to communicate the inverse to just one or two processors which would then communicate partially FTRANned columns and complete BTRAN for columns whose BTRAN operation has been started elsewhere. An obvious algorithmic extension to ASYNPLEX is to use several INVERT processes to increase the frequency with which a new factored inverse becomes available. This would reduce the average time for FTRAN and BTRAN at a cost of increased communication.

These algorithmic developments are currently being considered and are expected to form the basis of future work.

The authors would like to thank the referees for their comments which have led to a great improvement in the presentation of this paper.

References

[1] R. E. Bixby and A. Martin. Parallelizing the dual simplex method. Technical Report SC-95-45, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1995.

[2] G. B. Dantzig and W. Orchard-Hays. The product form for the inverse in the simplex method. *Math. Comp.*, 8:64–67, 1954.

[3] J. Eckstein, I. Boduroglu, L. Polymenakos, and D. Goldfarb. Data-parallel implementations of dense simplex methods on the Connection Machine CM-2. *ORSA Journal on Computing*, 7(4):402–416, 1995.

[4] J. J. H. Forrest and J. A. Tomlin. Vector processing in the simplex and interior methods for linear programming. *Annals of Operations Research*, 22:71–100, 1990.

[5] D. M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, 13:10–12, 1985.

[6] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Sparse matrix methods in optimization. *SIAM J. Sci. Stat. Comput.*, 5:562–589, 1984.

[7] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linear and nonlinear programming. Technical Report SOL 88-4, Systems Optimization Laboratory, Stanford University, 1990.

[8] D. Goldfarb and J. K. Reid. A practical steepest-edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977.

[9] J. A. J. Hall and K. I. M. McKinnon. Update procedures for the parallel revised simplex method. Technical Report MSR 92-13, Department of Mathematics and Statistics, University of Edinburgh, 1992.

[10] J. A. J. Hall and K. I. M. McKinnon. PARSMI, a parallel revised simplex algorithm incorporating minor iterations and Devex pricing. In J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, editors, *Applied Parallel Computing*, volume 1184 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 1996.

[11] P. M. J. Harris. Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5:1–28, 1973.

[12] J. K. Ho and R. P. .Sundarraj. On the efficacy of distributed simplex algorithms for linear programming. *Computational Optimization and Applications*, 3(4):349–363, 1994.

[13] IBM. *Optimization Subroutine Library, guide and reference, release 2*, 1993.

[14] J. Luo, A. N. M. Hulsbosch, and G. L. Reijns. An MIMD work-station for large LP problems. In E. Chiricozzi and A. D'Amico, editors, *Parallel Processing and Applications*, pages 159–169. Elsevier Science Publishers B.V. (North-Holland), 1988.

[15] C. E. Pfefferkorn and J. A. Tomlin. Design of a linear programming system for the ILLIAC IV. Technical Report SOL 76-8, Systems Optimization Laboratory, Stanford University, 1976.

[16] W. Shu and M. Wu. Sparse implementation of revised simplex algorithms on parallel computers. In *Proceedings of* 6^{th} *SIAM Conference on Parallel Processing for Scientific Computing*, pages 501–509, 1993.

[17] C. B. Stunkel. Linear optimization via message-based parallel processing. In *International Conference on Parallel Processing*, volume III, pages 264–271, August 1988.

[18] R. Wunderling. Paralleler und objektorientierter simplex. Technical Report TR-96-09, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1996.