Towards a practical parallelisation of the simplex method

J. A. J. Hall

23rd April 2007

Abstract

The simplex method is frequently the most efficient method of solving linear programming (LP) problems. This paper reviews previous attempts to parallelise the simplex method in relation to efficient serial simplex techniques and the nature of practical LP problems. For the major challenge of solving general large sparse LP problems, there has been no parallelisation of the simplex method that offers significantly improved performance over a good serial implementation. However, there has been some success in developing parallel solvers for LPs that are dense or have particular structural properties. As an outcome of the review, this paper identifies scope for future work towards the goal of developing parallel implementations of the simplex method that are of practical value.

Keywords: linear programming, simplex method, sparse, parallel computing

MSC classification: 90C05

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 simplex method and interior point methods are the two main approaches to solving LP problems. In a context where families of related LP problems have to be solved, such as integer programming and decomposition methods, and for certain classes of single LP problems, the simplex method is usually more efficient.

The application of parallel and vector processing to the simplex method for linear programming has been considered since the early 1970's. However, only since the beginning of the 1980's have attempts been made to develop implementations, with the period from the late 1980's to the late 1990's seeing the greatest activity. Although there have been a few experiments using vector processing and shared memory machines, the vast majority of implementations have made use of distributed memory multiprocessors and ethernet-connected clusters.

The initial aim of this paper is to provide a comprehensive review of past approaches to exploiting parallelism in the simplex method, including an assessment of the extent to which they have yielded implementations of practical value. To facilitate this, Section 2 introduces the simplex method and discusses issues of implementation and behaviour that influence its parallelisation. A short overview of the nature of practical LP problems and suitable test problems is given in Section 3. Terms and concepts in parallel computing that are used in this paper are introduced briefly in Section 4.

It is clear from the review of past work in Section 5 that, in most cases, the focus of attention has been the development of techniques by which speed-up can be achieved. Little thought, if any, has been given to the underlying computational scheme in terms of serial efficiency and numerical robustness. As a consequence, although many implementations have demonstrated good speed-up, and a few were worthwhile parallel challenges at the time, fewer still have been of practical value.

The second aim of the paper is to identify promising computational strategies for worthwhile future parallel implementations. This is done in Section 6. Although the feasibility of their implementation is considered, detailed discussion of techniques and architectures is beyond the scope of this paper.

2 The simplex method

The simplex method and its computational requirements are most conveniently discussed in the context of LP problems in standard form

$$\begin{array}{rl} \text{minimize} & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0}, \end{array} \tag{1}$$

where $\boldsymbol{x} \in I\!R^n$ and $\boldsymbol{b} \in I\!R^m$. The matrix A in (1) usually contains columns of the identity corresponding to *logical* (slack) variables introduced to transform inequality constraints into equations. The remaining columns of A correspond to *structural* (original) variables.

In the simplex method, the indices of variables are partitioned into sets $\mathcal{B}$ corresponding to m basic variables $\boldsymbol{x}_B$, and $\mathcal{N}$ corresponding to $n-m$ nonbasic variables $\boldsymbol{x}_N$, such that the basis matrix B formed from the columns of A corresponding to $\mathcal{B}$ is nonsingular. The set $\mathcal{B}$ itself is conventionally referred to as the basis. The columns of A corresponding to $\mathcal{N}$ form the matrix N. The components of $\boldsymbol{c}$ corresponding to $\mathcal{B}$ and $\mathcal{N}$ are referred to as, respectively, the basic costs $\boldsymbol{c}_B$ and non-basic costs $\boldsymbol{c}_N$.

When the nonbasic variables are set to zero the values $\hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$ of the basic variables, if non-negative, correspond to a vertex of the feasible region. The expression $\boldsymbol{x}_B + B^{-1}N = \hat{\boldsymbol{b}}$ derived from the equations in (1) allows the basic variables to be eliminated from the objective which becomes $(\boldsymbol{c}_N^T - \boldsymbol{c}_B^T B^{-1}N)\boldsymbol{x}_N + \boldsymbol{c}_B^T\hat{\boldsymbol{b}}$. If none of the components of the vector of *reduced costs* $\hat{\boldsymbol{c}}_N = \boldsymbol{c}_N^T - \boldsymbol{c}_B^T B^{-1}N$ is negative then the current basis is optimal.

In each iteration of the simplex method, if the current basis is not optimal a nonbasic variable x_q with negative reduced cost is chosen to enter the basis. Increasing this variable from zero whilst maintaining the equations in (1) corresponds to moving along an edge of the feasible region such that the objective function decreases. The direction of this edge is given by the column $\hat{\boldsymbol{a}}_q$ of $\hat{N} = B^{-1}N$ corresponding to x_q. By considering the ratios of the components of $\hat{\boldsymbol{b}}$ to the corresponding components of $\hat{\boldsymbol{a}}_q$ (when positive), the simplex method finds the first basic variable to be reduced to zero as x_q is increased and, hence, the step to the next vertex of the feasible region along this edge.

There are many strategies for choosing the nonbasic variable x_q to enter the basis. The original rule of choosing the variable with the most negative reduced cost is commonly referred to as the Dantzig criterion. However, if the components of $\hat{\boldsymbol{a}}_j$ are large relative to $\hat{c}_j$ it is likely that only a small increase in x_j will be possible before one of the basic variables is reduced to zero. Alternative pricing strategies weight the reduced cost by dividing it by (a measure of) the length of $\hat{\boldsymbol{a}}_j$. The exact steepest edge strategy, described by Goldfarb and Reid [34], maintains weights $s_j = 1 + \|\hat{\boldsymbol{a}}_j\|^2$ corresponding to the step length for a unit change in x_j. Practical (approximate) steepest edge techniques are described by Forrest and Goldfarb [29] and the *Devex* strategy due to Harris [42] maintains approximate edge weights. Using these strategies, the number of iterations required to solve an LP problem in practice can be taken as $O(m+n)$ and no problem is known for which the theoretical complexity of $O(2^n)$ is achieved.

A popular technique for choosing the variable to leave the basis is the EXPAND procedure of Gill *et al.* [32]. By expanding the constraint bounds by a small amount, this strategy often enables the leaving variable to be

chosen from a set of possibilities on grounds of numerical stability.

The two main variants of the simplex method correspond to different means of calculating the data required to determine the step to the new vertex. The first variant is the standard simplex method in which the reduced costs and the directions of all edges at the current vertex are maintained in a rectangular tableau. In the revised simplex method, the reduced costs and the direction of the chosen edge are determined by solving systems involving the basis matrix B.

2.1 The standard simplex method

In the standard simplex method the matrix $\hat{N}$, the (reduced) right-hand-side vector $\hat{\boldsymbol{b}}$, the reduced costs $\hat{\boldsymbol{c}}_N$ and current value of the objective $\hat{f} = \boldsymbol{c}_B^T \hat{\boldsymbol{b}}$ are maintained in a tableau of the following form.

	$\mathcal{N}$	RHS
$\mathcal{B}$	$\hat{N}$	$\hat{\boldsymbol{b}}$
	$\hat{\boldsymbol{c}}_N^T$	$-\hat{f}$

Each iteration of the standard simplex method requires a Gauss-Jordan elimination operation to be applied to the columns of the tableau so that the updated tableau corresponds to the new basis.

The simplex method is commonly started from a basis for which $B = I$ so the matrix in the standard simplex tableau is N. As such the tableau is sparse. It is widely assumed that the extent of fill-in caused by the elimination operations is such that it is not worth exploiting sparsity. As a consequence, the standard simplex method is generally implemented using a dense data structure.

The standard simplex method is inherently numerically unstable since it corresponds to a long sequence of elimination operations with pivots chosen by the simplex algorithm rather than on numerical grounds. If the simplex method encounters ill-conditioned basis matrices, any subsequent tableau corresponding to a well-conditioned basis can be expected to have numerical errors reflecting the earlier ill-conditioning. This can lead to choices of entering or leaving variables such that, with exact arithmetic, the objective does not decrease monotonically, feasibility is lost, or the basis matrix becomes singular. Robustness can only be achieved by monitoring errors in the tableau and, if necessary, recomputing the tableau in a numerically stable manner. Error checking can be performed by comparing the updated reduced cost with the value computed directly using the pivotal column and basic costs. Alternatively, since operations with the inverse of the basis matrix can be

performed using appropriate entries of the tableau, calculating the pivotal column directly and comparing this with the tableau entries would provide a more extensive but expensive error checking mechanism.

2.2 The revised simplex method

The computational components of the revised simplex method are illustrated in Figure 1. At the beginning of an iteration, it is assumed that the vector of reduced costs $\hat{\boldsymbol{c}}_N$ and the vector $\hat{\boldsymbol{b}}$ of current values of the basic variables are known and that a representation of B^{-1} is available. The first operation is CHUZC which scans the (weighted) reduced costs to determine a good candidate q to enter the basis. The *pivotal column* $\hat{\boldsymbol{a}}_q$ is formed using the representation of B^{-1} in an operation referred to as FTRAN.

The CHUZR operation determines the variable to leave the basis, with p being used to denote the index of the row in which the leaving variable occurred, referred to as the *pivotal row*. The index of the leaving variable itself is denoted by p'. Once the indices q and p' have been interchanged between the sets $\mathcal{B}$ and $\mathcal{N}$, a *basis change* is said to have occurred. The RHS vector $\hat{\boldsymbol{b}}$ is then updated to correspond to the increase $\alpha = \hat{b}_p/\hat{a}_{pq}$ in x_q.

`CHUZC:` Scan $\hat{\boldsymbol{c}}_N$ for a good candidate q to enter the basis.
`FTRAN:` Form the pivotal column $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$, 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. Let $\alpha = \hat{b}_p/\hat{a}_{pq}$.
Update $\hat{\boldsymbol{b}} := \hat{\boldsymbol{b}} - \alpha\hat{\boldsymbol{a}}_q$.
`BTRAN:` Form $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$.
`PRICE:` Form the pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$.
Update reduced costs $\hat{\boldsymbol{c}}_N^T := \hat{\boldsymbol{c}}_N^T - \hat{c}_q\hat{\boldsymbol{a}}_p^T$.
`If` {growth in representation of B} `then`
`INVERT:` Form a new representation of B^{-1}.
`else`
`UPDATE:` Update the representation of B^{-1} corresponding to the basis change.
`end if`

Figure 1: Operations in an iteration of the revised simplex method

Before the next iteration can be performed it is necessary to obtain the reduced costs and a representation of the new matrix B^{-1}. Although the reduced costs may be computed directly using the following operations,

$$\boldsymbol{\pi}_B^T = \boldsymbol{c}_B^T B^{-1}; \quad \hat{\boldsymbol{c}}_N^T = \boldsymbol{c}_N^T - \boldsymbol{\pi}_B^T N, \tag{2}$$

it is more efficient computationally to update them by computing the pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{e}_p^T B^{-1} N$ of the standard simplex tableau. This is obtained in two steps. First the vector $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$ is formed using the representation of B^{-1} in an operation known as BTRAN, and then the vector $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$ of values in the pivotal row is formed. This sparse matrix-vector product with N is referred to as PRICE. Once the reduced costs have been updated, the UPDATE operation modifies the representation of B^{-1} according to the basis change. Note that, periodically, it will generally be either more efficient, or necessary for numerical stability, to find a new representation of B^{-1} using the INVERT operation.

When using the Devex strategy [42] the pivotal row computed to update the reduced costs is used to update the Devex weights at no significant computational cost. To update the exact steepest edge weights requires, in addition to the pivotal row, a further BTRAN operation to form $\hat{\boldsymbol{a}}_q^T B^{-1}$ and a further PRICE operation to form the product of this vector with N. It is also computationally expensive to initialise steepest edge weights if the initial basis matrix is not the identity. As a consequence of these overheads, and since the Devex strategy performs well in terms of reducing the number of iterations required to solve LP problems, Devex is commonly the default in efficient sequential implementations of the revised simplex method.

2.3 The revised simplex method with multiple pricing

When in-core memory was severely restricted, a popular variant of the revised simplex method was to incorporate minor iterations of the standard simplex method, restricted to a small subset of the variables. This is described by Orchard-Hays [61] and is referred to as *multiple pricing*. The major computational steps of this variant, together with those required when using Devex or steepest edge weights, are illustrated in Figure 2. The features of these variants, in particular those incorporating Devex or steepest edge, make exploiting parallelism particularly attractive.

The CHUZC operation scans the (weighted) reduced costs to determine a set $\mathcal{Q}$ of good candidates to enter the basis. The inner loop then applies the standard simplex method to the LP problem corresponding to the candidates in $\mathcal{Q}$ so requires the corresponding columns $\hat{\boldsymbol{a}}_j = B^{-1}\boldsymbol{a}_j$, $j \in \mathcal{Q}$, of the standard simplex tableau. These columns are formed as a multiple FTRAN. The matrix formed by the columns $\hat{\boldsymbol{a}}_j$, $j \in \mathcal{Q}$ and the corresponding vector of reduced costs for the candidates $j \in \mathcal{Q}$ are conveniently denoted by $\hat{\boldsymbol{a}}_Q$ and $\hat{\boldsymbol{c}}_Q$.

In each minor iteration, CHUZC_MI scans the (weighted) reduced costs of the candidates in $\mathcal{Q}$ and selects one, q say, to enter the basis. Once a basis

`CHUZC:` Scan $\hat{\boldsymbol{c}}_N$ for a set $\mathcal{Q}$ of good candidates to enter the basis.
`FTRAN:` Form $\hat{\boldsymbol{a}}_j = B^{-1}\boldsymbol{a}_j$, $\forall\, j \in \mathcal{Q}$, where $\boldsymbol{a}_j$ is column j of A.
`Loop` {minor iterations}
    `CHUZC_MI:` Scan $\hat{\boldsymbol{c}}_Q$ for a good candidate q to enter the basis.
    `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 $\alpha = \hat{b}_p/\hat{a}_{pq}$.
    `UPDATE_MI:` Update $\mathcal{Q} := \mathcal{Q}\backslash\{q\}$ and $\hat{\boldsymbol{b}} := \hat{\boldsymbol{b}} - \alpha\hat{\boldsymbol{a}}_q$.
    Update the columns $\hat{\boldsymbol{a}}_Q$ and reduced costs $\hat{\boldsymbol{c}}_Q$.
    `If` {Devex} `then`
        Update the Devex weights for the candidates in $\mathcal{Q}$.
    `else if` {steepest edge} `then`
        Update the steepest edge weights for the candidates in $\mathcal{Q}$.
    `end if`
`End loop` {minor iterations}
`For` {each basis change} `do`
    `If` {Devex} `then`
        `BTRAN:` Form $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$.
        `PRICE:` Form pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$.
        Update reduced costs $\hat{\boldsymbol{c}}_N := \hat{\boldsymbol{c}}_N - \hat{c}_q\hat{\boldsymbol{a}}_p^T$ and Devex weights.
    `else if` {steepest edge} `then`
        `BTRAN:` Form $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$.
        `PRICE:` Form pivotal row $\hat{\boldsymbol{a}}_p^T = \boldsymbol{\pi}_p^T N$.
        Update reduced costs $\hat{\boldsymbol{c}}_N := \hat{\boldsymbol{c}}_N - \hat{c}_q\hat{\boldsymbol{a}}_p^T$.
        `BTRAN:` Form $\boldsymbol{w} = \hat{\boldsymbol{a}}_q^T B^{-1}$.
        `PRICE:` Form $\boldsymbol{w}^T\boldsymbol{a}_j$ for nonzero components of pivotal row $\hat{\boldsymbol{a}}_p^T$.
    `end if`
    `If` {growth in factors} `then`
        `INVERT:` Form a new representation of B^{-1}.
    `else`
        `UPDATE:` Update the representation of B^{-1} corresponding to the basis change.
    `end if`
`End do`
`If` {Dantzig} `then`
    `BTRAN:` Form $\boldsymbol{\pi}_B^T = \boldsymbol{c}_B^T B^{-1}$.
    `PRICE:` Form $\hat{\boldsymbol{c}}_N^T = \boldsymbol{c}_N^T - \boldsymbol{\pi}_B^T N$.
`end if`

Figure 2: The revised simplex method with multiple pricing and the Dantzig, Devex and steepest edge pricing strategies

change has been determined by CHUZR and the RHS has been updated, the standard simplex tableau corresponding to the new basis is obtained by updating the previous tableau in an operation known as UPDATE_MI. The matrix $\hat{\boldsymbol{a}}_Q$ and reduced costs $\hat{\boldsymbol{c}}_Q$ are updated by the standard Gauss-Jordan elimination step. Any Devex weights are updated using row p of the updated tableau. Any steepest edge weights may be computed directly using the known tableau columns. Minor iterations are terminated when either $\mathcal{Q} = \emptyset$ or there are no negative reduced costs for $q \in \mathcal{Q}$.

An original advantage of multiple pricing was that the reduced costs were only computed (from scratch) every major iteration using (2). As a result, the frequency with which BTRAN and, in particular, PRICE were performed was reduced significantly.

Leaving aside any advantages due to efficient use of memory, the value of the revised simplex method with multiple pricing depends on the extent to which the variables that were good candidates to enter the basis when $\mathcal{Q}$ was formed remain good candidates, or even remain attractive, during the course of minor iterations. This property is commonly referred to as *candidate persistence* and its effect depends upon both the problem being solved and the number of candidates chosen when forming $\mathcal{Q}$. Since the entering variable in the second and subsequent minor iteration is not necessarily the best candidate according to the original selection criterion, it is possible that the number of basis changes required to solve the LP problem will increase, perhaps significantly. If any of the original candidates becomes unattractive and minor iterations terminate with $\mathcal{Q} \neq \emptyset$, the FTRAN and update operations required to form and maintain the tableau column for such a candidate are wasted.

2.4 The representation of B^{-1}

Forming and updating the explicit inverse matrix of B as a dense data structure is simple and, if reformed when appropriate, is numerically stable. The FTRAN and BTRAN operations thus reduce to dense matrix-vector products. For general large sparse LP problems the computational cost of an iteration of the revised simplex method with a dense inverse is $O(m^2)$. This is comparable with the $O(mn)$ cost of the standard simplex method, and so is of no greater practical interest. Efficient implementations of the revised simplex method use a data structure corresponding to a factored representation of B^{-1}. This representation is based on that obtained by the INVERT operation for some basis matrix B_0. There are various techniques for updating the factored representation of B^{-1}, the original and simplest being the product form update of Dantzig and Orchard-Hays [22].

2.4.1 Procedures for INVERT

Although there are many techniques for factorising a general unsymmetric sparse matrix, in the context of the revised simplex method, there is much commonality of approach. The general aim is to use Gaussian elimination to obtain a factored representation of B^{-1}, seeking a reconciliation of the partially conflicting goals of low computational cost, low fill-in and numerical stability. Using an active submatrix that is, initially, B, Gaussian elimination determines a sequence of m entries to be used as pivots. Each pivot is used to eliminate any other entries in the corresponding column. The active submatrix is then updated by removing the row and column corresponding to the pivot. No INVERT technique is optimal for all LP problems.

LP basis matrices frequently contain significant numbers of columns of the identity matrix corresponding to logical variables. These are examples of singleton columns whose entries can be used as pivots without incurring any fill-in. It is also likely that there will be singleton rows that can be used as pivots similarly. As singletons are identified, pivoted on and their corresponding rows and columns removed from the active submatrix, further singletons may be created. The use of this *triangularisation* phase is due to Orchard-Hays [61] and ends when no further singletons can be identified. Any remaining rows and columns contain at least two nonzeros and form what is frequently referred to as the *bump*.

For certain LP problems, in particular network problems, the triangularisation phase only terminates when all rows and columns have been pivoted on. This corresponds to permuting the rows and columns of B so that it is triangular, allowing B^{-1} to be factored without incurring any fill-in. For many LP problems, the number of rows and columns remaining after triangularisation is very small. In this case, the technique used to factor the bump is largely immaterial since the computationally intensive part of INVERT is triangularisation and any fill-in is small relative to the number of nonzeros in the representation of B^{-1}. For other LP problems, the size of the bump is comparable to that of the matrix, in which case it is important how the bump is factored. A simple technique due to Tomlin [70] is effective for all but the most numerically awkward and structurally unfortunate LP problems, for which a Markowitz [54] strategy such as that used by Suhl and Suhl [68] may be preferable.

In a typical implementation of the revised simplex method for large sparse LP problems, B_0^{-1} is represented as a product of K_I elimination operations derived directly from the nontrivial columns of the matrices L and U in the LU decomposition of (a row and column permutation of) B_0 resulting from the elimination. This invertible representation allows B_0^{-1} to be expressed

algebraically as $B_0^{-1} = \prod_{k=K_I}^{1} E_k^{-1}$, where

$$E_k^{-1} = \begin{bmatrix} 1 & & & -\eta_1^k & & & \\ & \ddots & & \vdots & & & \\ & & 1 & -\eta_{p_k-1}^k & & & \\ & & & 1 & & & \\ & & & -\eta_{p_k+1}^k & 1 & & \\ & & & \vdots & & \ddots & \\ & & & -\eta_m^k & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & \mu^k & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}. \quad (3)$$

Within an implementation, the nonzeros in the 'eta' vector

$$\boldsymbol{\eta}_k = [\, \eta_1^k \quad \ldots \quad \eta_{p_k-1}^k \quad 0 \quad \eta_{p_k+1}^k \quad \ldots \quad \eta_m^k \,]^T$$

are stored as value-index pairs and the data structure $\{p_k, \mu_k, \boldsymbol{\eta}_k\}_{k=1}^{K_I}$ is referred to as an eta file. Eta vectors associated with the matrices L and U coming from Gaussian elimination are referred to as L-etas and U-etas respectively. The operations with p_k, μ_k and $\boldsymbol{\eta}_k$ required when forming $E_k^{-1}\boldsymbol{r}$ during the standard algorithms for FTRAN and $E_k^{-T}\boldsymbol{r}$ during BTRAN are illustrated in Figure 3.

```
if (r_{p_k} ≠ 0) then
   r_{p_k} := μ_k r_{p_k}
   r := r − r_{p_k} η_k
end if
```

(a) FTRAN

$r_{p_k} := \mu_k(r_{p_k} - \boldsymbol{r}^T\boldsymbol{\eta}_k)$

(b) BTRAN

Figure 3: Standard operations in FTRAN and BTRAN

2.4.2 Updating the factored representation of B^{-1}

If E_U is used to denote the transformation of B_0 corresponding the basis changes following INVERT, such that $B = B_0 E_U$, it follows that B^{-1} may be expressed as $B^{-1} = E_U^{-1} B_0^{-1}$. The product form update leaves the factored form of B_0^{-1} unaltered and represents E_U^{-1} as a product of pairs of elementary matrices of the form (3). In the UPDATE operation, the modification of B^{-1} corresponding to the basis change is obtained directly from the pivotal column and is given by $p_k = p$, $\mu_k = 1/\hat{a}_{pq}$ and $\boldsymbol{\eta}_k = \hat{\boldsymbol{a}}_q - \hat{a}_{pq}\boldsymbol{e}_p$.

In solvers based on the product form, the representation of the UPDATE operations can be appended to the eta file following INVERT, resulting in a

single homogeneous data structure. However, in this paper, the particular properties of the INVERT and UPDATE etas and the nature of the operations with them may be exploited, so FTRAN is viewed as the pair of operations

$$\tilde{\boldsymbol{a}}_q = B_0^{-1}\boldsymbol{a}_q \qquad \text{(I-FTRAN)}$$

followed by

$$\hat{\boldsymbol{a}}_q = E_U^{-1}\tilde{\boldsymbol{a}}_q. \qquad \text{(U-FTRAN)}$$

Conversely, BTRAN is viewed as

$$\tilde{\boldsymbol{\pi}}^T = \boldsymbol{r}^T E_U^{-1} \qquad \text{(U-BTRAN)}$$

followed by

$$\boldsymbol{\pi}^T = \tilde{\boldsymbol{\pi}}^T B_0^{-1}. \qquad \text{(I-BTRAN)}$$

Other procedures for UPDATE modify the factored representation of B_0^{-1} with the aim of reducing the size of the eta file and achieving greater numerical stability. Whilst they may be more efficient in serial for some problems, there is a significant value in leaving the factored representation of B_0^{-1} unaltered. Techniques for improving the efficiency of FTRAN and BTRAN, both in serial and parallel, may require an analysis of the representation of B_0^{-1} following INVERT that is only a worthwhile investment if many FTRAN and BTRAN operations are performed with the original representation of B_0^{-1}.

2.5 Hyper-sparsity

For most sparse LP problems the pivotal column $\hat{\boldsymbol{a}}_q$ has sufficient zero entries that it is efficient to exploit this when creating the corresponding product form eta in UPDATE. Similarly, it is often found that the vector $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$ has a significant proportion of zero entries. Thus, if PRICE is performed as a set of inner products $\boldsymbol{\pi}_p^T\boldsymbol{a}_i$, $i \in \mathcal{N}$, there will be many operations with zeros. In this case it is more efficient to evaluate $\boldsymbol{\pi}_p^T N$ as a linear combination of the rows of N corresponding to the nonzeros in $\boldsymbol{\pi}_p$, even allowing for the overhead of updating a row-wise representation of N following each basis change.

For some important classes of LP problems, the number of nonzeros in the vectors $\hat{\boldsymbol{a}}_q$, $\boldsymbol{\pi}_p$, and pivotal tableau row $\hat{\boldsymbol{a}}_p^T$ is sufficiently small to be worth exploiting further. This property, which Hall and McKinnon [40] refer to as *hyper-sparsity*, has also been identified by Bixby *et al.* [13, 14]. For such problems, Hall and McKinnon [41] have shown that significant performance improvement can be gained by exploiting hyper-sparsity. This is achieved by maintaining the indices of the nonzeros in these vectors and exploiting this

information when they are both formed and used. Practical experience shows that the threshold at which it is worth exploiting hyper-sparsity corresponds to an average density of pivotal columns (and hence the standard simplex tableau itself) of about 10%. For the basis matrices of LP problems that exhibit hyper-sparsity the dimension of the bump within INVERT is very small relative to that of the matrix.

For some LP problems B^{-1} is typically sparse. This property has been exploited in some implementations of the revised simplex method based on an explicit inverse. However, it is general for the number of nonzeros in B^{-1} (even when freshly formed) to be vastly more than the number of nonzeros in an efficient factored representation. This property is illustrated, for example, by Shu [66].

2.6 PRICE when the column/row ratio is large

For LP problems with a large column/row ratio ($n \gg m$), the cost of the revised simplex variants set out in Figures 1 and 2 is dominated by that of PRICE. However, for such problems, it is typical for a good candidate to enter the basis to be identified from only a small subset of the columns. The numerous variants of partial and multiple pricing [61] stem from this observation. In general, their use leads to a reduction in the solution time that is significant, and in some cases vast, despite the fact that they are largely incompatible with the use of edge weight pricing strategies.

3 Practical LP problems

The following overview provides the necessary background on LP problems for reviewing parallel simplex techniques in Section 5 in terms of their value as practical LP solvers. In view of some of the LP problems that have been used to test parallel simplex implementations, a few observations on this issue are also in order.

A direct consequence of many modelling techniques that generate large LP problems is structure and sparsity in the constraint matrix. Variable and constraint sets corresponding to related LP subproblems are linked to build large single problems. Two important classes of practical LP problems have block-angular structure. Applications involving decentralised planning and specific problems such as multicommodity network flow yield row-linked block-angular LP problems. Stochastic programming problems in applications such as asset liability management yield column-linked block-angular LP problems. Depending on the application, the nature of the blocks may

range from being fully dense to highly sparse and structured, as in the case of multicommodity flow problems.

In determining the computational challenge of solving a given LP problem, structure and sparsity are often more important than problem dimension. In particular, they have a critical influence on which of the simplex and interior point methods is the more efficient solution procedure. For problems that exhibit hyper-sparsity when solved using the revised simplex method, it is typical for an interior point solver to be several times slower and in some cases the margin is more than an order of magnitude. For LP problems that are not (fully) hyper-sparse, an interior point solver is usually faster by similar factors. It is not properly understood what model properties determine whether the LP will exhibit hyper-sparsity, but it is often associated with a significant amount of network structure. In terms of developing a parallel simplex solver of practical value, two challenges emerge. A successful parallelisation of the revised simplex method when exploiting hyper-sparsity would lead immediately to a solver of great value, whereas a parallel simplex solver for problems that are not hyper-sparse may catch up valuable "lost ground" in the competition with interior point methods. These challenges are discussed in greater detail in Section 6.

There are a few applications such as the design of filters and wavelet analysis which yield large dense LP problems. Their existence is used to justify the practical value of developing parallel implementations of the standard simplex method and the revised simplex method with dense matrix algebra. However, the existence of these applications should not be over-played, nor obscure the fact that the development of methods for the efficient solution of sparse LP problems remains the overwhelmingly important goal.

As well as sparsity and structure, the degeneracy and numerical properties of an LP problem are important factors affecting the computational challenge posed by its solution. Both of these attributes is usually a consequence of the modelling process. For this reason, randomly generated problems offer a poor means of testing the performance of an LP solver, with random sparsity patterns being particularly misleading. Despite the availability of sparse LP test problems, it is disappointing how many authors give results for randomly generated problems.

For many years the Netlib set [31] was the standard source of general sparse LP test problems. Although a few are difficult to solve for numerical or structural reasons and some are highly degenerate, none can now be viewed as being large. The original Kennington set [18] contains significantly larger problems. The test problems used by Mittelmann for benchmarking commercial LP solvers [59] are large by modern standards and reflect the full range of attributes that contribute to the computational challenge of solving

large sparse practical LP problems.

4 Parallel computing

In classifying the approaches to attempts to parallelise the simplex method, reference is made to terms and concepts in parallel computing. This section introduces the necessary terms. A full and more general introduction to parallel computing is given by Kumar *et al.* [49].

4.1 Parallel multiprocessors

When classifying parallel multiprocessor architectures, an important distinction is between *distributed memory*, when each processor has its own local memory, and *shared memory*, when all processors have access to a common memory. Modern massively parallel machines may consist of a set of distributed clusters, each of which has a number of processors with shared memory. On smaller multiprocessors memory may be either purely distributed or shared.

4.2 Speed-up, scalability and Amdahl's law

In general, success in parallelisation is measured in terms of *speed-up*, the time required to solve a problem with more than one parallel processor compared with the time required using a single processor. The traditional goal is to achieve a speed-up factor equal to the number of processors. Such a factor is referred to as *linear speed-up* and corresponds to a *parallel efficiency* of 100%. The increase in available cache, and frequently RAM, with the number of parallel processors occasionally leads to the phenomenon of *super-linear speed-up* being observed. Parallel schemes for which, at least in theory, performance improves linearly and without limit as the number of processors increases are said to be *scalable*. If parallelism is not exploited in all major algorithmic operations then the speed-up is limited, according to Amdahl's law [2], by the proportion of the execution time for the non-parallel operations.

4.3 Programming paradigms and their implementation

There are two principal parallel programming paradigms. If the work of a major algorithmic operation may be distributed across a set of processors then

data parallelism may be exploited. Conversely, if it is possible to perform several major algorithmic operations simultaneously, then *task parallelism* may be exploited. In practice, it may be possible to exploit both data parallelism and task parallelism for a particular (set of) major algorithmic operation(s).

There are two fundamental means of implementing algorithms on parallel machines. On distributed memory machines the concept of communicating data between processors using instructions initiated by explicit calls to *message-passing* subroutines is natural and immediate. On shared memory machines the conceptually natural technique is *data-parallel* programming in which operations are coded as if they were to be executed serially but translated into a parallel executable by use of an appropriate compiler. Many message-passing environments are also supported on shared memory machines and data-parallel programming techniques are also supported on distributed memory machines.

On distributed memory machines, the overhead of sending messages to communicate data between processors is determined by both *latency* and *bandwidth*. The former is a time overhead that is independent of the size of the message and the latter is the rate of communication. For generic message-passing environments, the latency and bandwidth on a particular architecture may be significantly higher than for an architecture-dependent environment that is usually supplied and tuned by the vendor. When an algorithm has a high ratio of communication to computation, growing communication overhead may outweigh any improvement in potential computational performance resulting from the use of an increased number of processors.

5 Parallel simplex and simplex-like methods

Past approaches to exploiting parallelism in the simplex method and simplex-like methods are conveniently classified according to the variant of the simplex method that is considered and the extent to which sparsity is exploited. This classification is correlated positively with the practical value of the implementation as a means of solving LP problems, and negatively with the success of the approaches in terms of speed-up.

A few of the parallel schemes discussed below offered good speed-up relative to efficient serial solvers of their time. However, some techniques that have been parallelised are only seen as being inefficient in the light of serial revised simplex techniques that were either little known at the time or developed subsequently. This is identified below to emphasise that, as a result of the huge increase in efficiency of serial revised simplex solvers both during and since the period of research into parallel simplex, the task of developing

a practical parallel simplex-based solver has increased enormously.

5.1 Parallel simplex using dense matrix algebra

The dense standard simplex method and the revised simplex method with a dense explicit inverse have been implemented in parallel many times. The simple data structures involved and potential for linear speed-up makes them attractive exercises in parallel computing. However, for solving general large scale sparse LP problems, the serial inefficiency of these implementations is such that only with a massive number of parallel processes could they conceivably compete with a good sparsity-exploiting serial implementation of the revised simplex method.

Early work on parallelising the simplex method using dense matrix algebra was mainly restricted to discussion of data distribution and communication schemes, with implementations limited to small numbers of processes on distributed memory machines. Surveys are given by Zenios [74] and Luo *et al.* [53], and examples of other early references to work in this area are due to Finkel [28], Boffey and Hay [17], Babayev and Mardanov [8] and Agrawal *et al.* [1]. A relatively early work which is representative of this era is due to Stunkel [67] who implemented both the dense standard simplex method and the revised simplex method with a dense inverse on a 16-processor Intel hypercube, achieving a speed-up of between 8 and 12 for small problems from the Netlib set [31]. Cvetanovic *et al.* [20] report a speed-up of 12 when solving two small problems using the standard simplex method, a result that is notable for being achieved on a 16-processor *shared* memory machine. Luo and Reijns [52] obtained speed-ups of more than 12 on 16 transputers when using the revised simplex method with a dense inverse to solve modest Netlib problems. However, they developed their parallel scheme on the assumption that the computational cost of updating the inverse using the expression $B^{-1} := E_k^{-1}B^{-1}$ was $O(m^3)$, rather than $O(m^2)$ if the structure of E_k^{-1} (3) is properly exploited, an error which might well explain their high efficiency.

Eckstein *et al.* [25] parallelised the standard simplex method and the revised simplex method with a dense inverse on the massively parallel Connection Machine CM-2 and CM-5, incorporating the steepest edge pricing strategy [34] in their standard simplex implementation. When solving a range of larger Netlib problems and very dense machine learning problems, speed-ups of between 1.6 and 1.8 were achieved when doubling the number of processors. They also presented results which indicated that the performance of their implementation was generally superior to the Minos 5.4 serial sparse revised simplex solver, particularly for the denser problems. Thomadakis and Liu [69] also used steepest edge in their implementation of the standard

simplex method on a MasPar MP-1 and MP-2. Solving a range of large, apparently randomly-generated problems, they achieved a speed-up of up to three orders of magnitude on the 128×128 processor MP-2. More recently, theoretical work on parallel implementations of the standard simplex method with steepest edge, and practical implementation on modest numbers of processors, has also been reported by Yarmish [73].

Reports of small-scale parallel implementations of the dense standard simplex method continue to appear. Very recently, Badr *et al.* [9] presented results for an implementation on eight processors, achieving a speed-up of five when solving small random dense LP problems.

5.2 Parallel simplex using sparse matrix algebra

The real challenge in developing a parallel simplex implementation of general practical value is to exploit parallelism when using efficient sparse matrix algebra techniques. Only then could the resulting solver be competitive with a good serial implementation when solving general large sparse LP problems using a realistic number of processors.

At the time when the parallelisation of the simplex method was first and most widely considered, practical parallel factorisation and solution methods for sparse unsymmetric linear systems were in their infancy. As a consequence, although work on parallelising the simplex method using dense matrix algebra has generally exploited data parallelism, it was a commonly held view that when using sparse matrix algebra, there was relatively little scope for exploiting data parallelism (with the exception of PRICE). This has led some to conclude that there is no real scope for developing a worthwhile parallel implementation. However, not only is this (now) open to question: despite the apparently sequential nature of the computational components of the revised simplex method, there is scope for exploiting task parallelism.

5.2.1 Sparse standard simplex method

It was observed above that the standard simplex method is generally implemented using dense matrix algebra. However, based on the observation that fill-in in the tableau can be low, Lentini *et al.* [50] developed a parallel implementation of the standard simplex method with the tableau stored as a sparse matrix. In this remarkably novel approach, they incorporated a Markowitz-like criterion into column and row selection with the aim of reducing fill-in. When solving medium sized Netlib problems on four transputers they achieved a speed-up of between 0.5 and 2.7, with a super-linear speed-up of 5.2 on one problem with a relatively large column-row ratio. They com-

pared their results on eight transputers with a sparse revised simplex solver running on a mainframe (without giving any justification of the validity of this comparison) and found that their parallel solver was a little faster on smaller problems but slower on larger problems, particularly when solving four proprietary problems. Although they describe their results as disappointing, this brave and imaginative approach to parallelising the simplex method deserves to be better known.

5.2.2 Revised simplex method with a sparse inverse

When the revised simplex method with an explicit inverse is implemented, dense data structures are generally used to maintain B^{-1}. However, Shu [66] identified that B^{-1} is by no means full and considered parallelising the revised simplex method with B^{-1} maintained using a sparse data structure. At first sight, her results using up to 64 processors on a Touchstone Delta, and up to 16 processors on an Intel iSPC/2 hypercube, appear good. Using the sparse representation of B^{-1}, as well as the traditional dense representation, Shu obtained a speed-up of up to 17 on the Touchstone Delta when solving small to modest Netlib problems. Unfortunately, for all but one of these problems the average density of B^{-1} is between 33% and 47% so there is little difference in the solution times when using a sparse or dense B^{-1}. Of those problems for which comparative dense-sparse results are given, only in the case of SHIP08L is the average sparsity (6%) of B^{-1} worth exploiting: the serial sparse implementation was twice as fast as the dense implementation and a speed-up of 16 was achieved, compared with a speed-up of 17 for the dense case. Results for significantly larger problems, including some of the CRE and OSA problems from the Kennington test set [18], are also given, although it is not clear how B^{-1} was represented. Speed-ups of up to 8 were achieved on both machines. However, for these larger problems, analysis by Shu [66] of the computational cost of components of her implementation of the revised simplex method with a factored inverse indicated that PRICE, which parallelises easily, constitutes between 65% and 96% of the solution time. Although these proportions would be lower in the explicit inverse implementation, they are much higher than would be the case were Shu to have updated reduced costs and use a row-wise PRICE, exploiting the moderate to high levels of hyper-sparsity that these problems are now known to possess [41]. If the results for the problems where PRICE dominates are excluded, the speed-ups obtained are in low single figures, even when using up to 64 processors.

5.2.3 Revised simplex method with a factored inverse

Efficient serial simplex solvers are based on the revised simplex method with a factored inverse since the practical LP problems whose solution poses a computational challenge are large and sparse. For such problems, the superiority of the revised simplex method with a factored inverse over the serial simplex techniques underlying the parallel schemes reviewed above is overwhelming. It follows that the only scope for developing a really worthwhile parallel simplex solver is to identify how the revised simplex method with a factored inverse may be parallelised.

The natural data parallelism of PRICE has been exploited by most authors. Some have then considered the data parallelism in other computational components, whereas others have studied the extent to which task parallelism can be exploited by overlapping operations that are then executed either in serial or, in the case of PRICE, by exploiting data parallelism.

Theoretical pure data parallel approaches

In an early work, Pfefferkorn and Tomlin [63] discussed how parallelism could be exploited in each computational component of the revised simplex method. This purely data-parallel scheme, devised for the ILLIAC IV, was never implemented. After observing the parallelism of a column PRICE operation, they then considered FTRAN and BTRAN. They identified that if a sequence $E_s^{-1} \ldots E_r^{-1}$ within the factored representation of B^{-1} has the property that the pivots are in distinct rows and the eta vectors have no entries in these rows, then the application of these etas is independent in both FTRAN and BTRAN. This may be illustrated, without loss of generality, by assuming that $r = 1$, $s = K$ and that the pivots are in rows 1, ..., K, in which case it is readily established that

$$E_K^{-1} \ldots E_1^{-1} = \begin{bmatrix} I & \\ E & I \end{bmatrix} \begin{bmatrix} M & \\ & I \end{bmatrix}$$

where $E_{ij} = -\eta_{K+i}^j$ and $M = \text{diag}\{\mu^1, \ldots, \mu^K\}$. The corresponding operations $E_K^{-1} \ldots E_1^{-1}\boldsymbol{r}$ in FTRAN may now be performed as

$$\boldsymbol{r}_K := M\boldsymbol{r}_K; \quad \boldsymbol{r}_{K'} := \boldsymbol{r}_{K'} + E\boldsymbol{r}_K \quad \text{where} \quad \boldsymbol{r} = \begin{bmatrix} \boldsymbol{r}_K \\ \boldsymbol{r}_{K'} \end{bmatrix}$$

and, in BTRAN, $\boldsymbol{r}_K^T := (\boldsymbol{r}_K^T + \boldsymbol{r}_{K'}^T E)M$. Note that for this technique to be effective it is necessary for most of the (initial) components of $\boldsymbol{r}_K$ to be nonzero or, equivalently, that most of the etas will have to be applied in the conventional view of FTRAN and BTRAN. For LP problems exhibiting

hyper-sparsity this does not occur so this technique will not be of value. For other problems, the assumption that such sequences exist to a significant extent may be unreasonable. Any techniques developed by Pfefferkorn and Tomlin for exploiting parallelism during INVERT, which is based on that of Tomlin [70], are either not stated or unclear, possibly referring to the potential for parallelising the search for singletons in the triangularisation phase.

Helgason *et al.* [43] discussed the scope for parallelising FTRAN and BTRAN based on the quadrant interlocking factorisation of Evans and Hatzopoulos [26]. This relatively unknown factorisation scheme corresponds to forming an LU decomposition of (a row and column permutation of) B in which, along their diagonals, L has 2×2 blocks and U has 2×2 identity matrices. Thus the entire eta file following INVERT consists of sequences of the type identified by Pfefferkorn and Tomlin [63], each of length two. It follows that if $\boldsymbol{r}$ fills in sufficiently fast during FTRAN and BTRAN, there is the potential for speed-up bounded above by two. Developed with a shared memory machine in mind, this scheme was never implemented. Whilst the potential speed up of two is small, an extension of this scheme using larger blocks could have greater potential.

McKinnon and Plab [57] considered how data parallelism could be exploited in FTRAN for both dense and sparse right hand sides. They also investigated how the Markowitz criterion could be modified in order to increase the potential for exploiting data parallelism in subsequent FTRAN operations [56].

Since the time when the parallelisation of the simplex method was first and most actively considered, there have been huge efforts and advances in practical parallel factorisation and solution methods for sparse unsymmetric linear systems. However, the techniques thus developed have not yet been applied to the revised simplex method, the scope for which is discussed in Section 6.

Parallel PRICE for the dual simplex method

Virtually all attempts to exploit parallelism in the simplex method have focused on the primal simplex method. The dual simplex method, which is very much more efficient for many single LP problems and is valuable when solving LP subproblems in branch-and-bound for integer programming, has computational requirements in each iteration that are very similar to those of the primal algorithm. There is, however, one important distinction between the primal and dual simplex methods. In the latter there is no variant corresponding to partial pricing in the primal simplex method whereby PRICE

is restricted to just a subset of the nonbasic variables. For problems with very large column/row ratios, the overwhelmingly dominant cost of serial dual simplex solvers is PRICE. Hence its parallelisation can be expected to yield a significant improvement in performance over that of efficient serial dual simplex solvers.

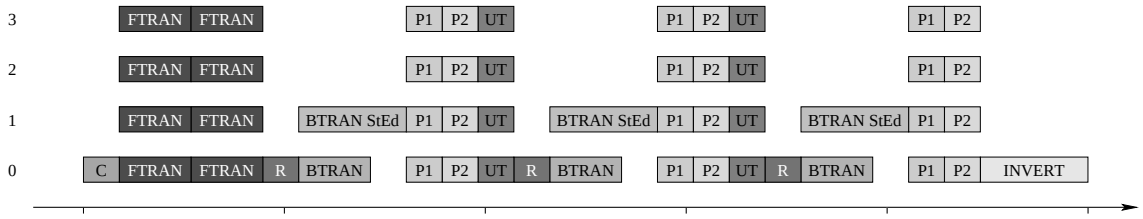
Bixby and Martin [15] investigated the scope for parallelism in the dual simplex method and chose to parallelise only those operations whose cost is related to the number of columns in the problem, that is PRICE, the dual ratio test and update of the dual variables. Much the most expensive of these is PRICE which, in the context of the dual algorithm, is used to refer to the matrix-vector product $\boldsymbol{z}^T N$. Each process formed a part of $\boldsymbol{z}^T N$, with the corresponding columns of N held row-wise so that sparsity in $\boldsymbol{z}$ can be exploited. The work of Bixby and Martin is also notable since the underlying dual simplex code being parallelised was CPLEX. Thus any speed-up was necessarily relative to a state-of-the-art serial solver. They implemented the dual simplex method on several architectures but, for the full Netlib set, the overhead of exploiting parallelism exceeded any gain in performance. This is unsurprising since few problems in the Netlib set have significantly more columns than rows. Focusing on test problems with very large column/row ratios, using two Sun workstations connected by ethernet they achieved very little speed-up. Performance using up to 4 processors on an IBM SP2 was better, with speed-up ranging from 1 to 3. Switching to experiments using 2 processors on shared memory machines, little speed-up was obtained with a Sun S20-502 but better results were obtained with a Silicon Graphics multiprocessor. In later work reported in this paper [15], the speed-up achieved using 4 processors of a shared memory SGI Power challenge was better still, being near linear for most problems with more than 64 columns per row. Strangely this behaviour tailed off for the problems with most extreme numbers of columns per row.

Task and data parallel revised simplex

The first attempt to exploit task parallelism in the revised simplex method was reported by Ho and Sundarraj [46]. In addition to the natural data parallelism of (column-wise) PRICE, Ho and Sundarraj identified that INVERT can be overlapped with simplex iterations. Once formed, the representation of B_0^{-1} is generally not up-to-date with respect to basis changes that occurred during INVERT. However, when using the product form update, a representation of B^{-1} is readily obtained by incorporating the eta vectors formed since INVERT was started. By dedicating one processor solely to INVERT, Ho and Sundarraj observe that the frequency of INVERT should be increased.

This can be expected to reduce the cost of FTRAN and BTRAN since the representation of B^{-1} will involve fewer eta vectors. Whilst they claim that having fewer eta vectors will have advantages in terms of numerical stability, they do not discuss the potential for numerical instability identified earlier by Hall and McKinnon [37]. As can be expected for only a partially parallelised algorithm, the practical performance of Ho and Sundarraj's implementation, on an Intel iPSC/2 and Sequent Balance 8000, is limited in accordance with Amdahl's law. On a set of small Netlib and proprietary problems, they report an average saving of 33% over the serial solution time.

To date, there have only been four implementations of the revised simplex method that have attempted to exploit parallelism anything like fully: one by Shu [66], one by Wunderling [71, 72] and two by Hall and McKinnon [38, 39]. The most ambitious of the parallel simplex implementations developed by Shu [66] remains the only pure data parallel implementation of the revised simplex method. It was based a parallel triangularisation phase for INVERT, a distributed sparse LU decomposition of the basis matrix for parallel FTRAN and BTRAN, as well as parallel PRICE. She did not parallelise the application of the product form update etas, claiming wrongly that its parallelisation is inconceivable. Unfortunately no speed-up was achieved. For one problem the solution time using two processors was the same as the serial time and, otherwise, the parallel solution time deteriorated rapidly.



C: CHUZC R: CHUZR P1: Update PRICE P2: Steepest edge PRICE

Figure 4: Wunderling's parallel revised simplex with steepest edge

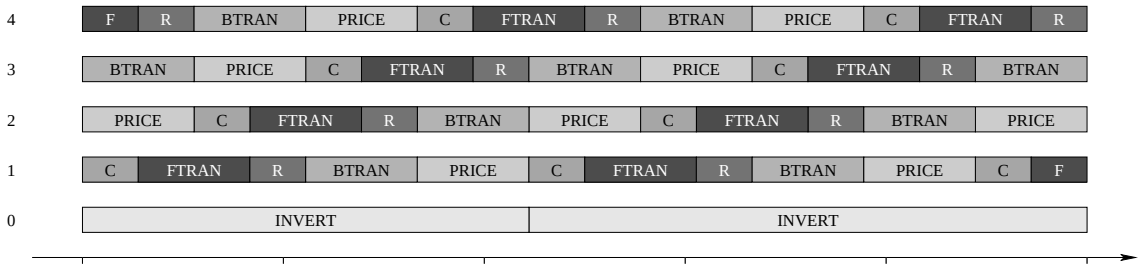
Wunderling's variant of the revised simplex method was based on multiple pricing with steepest edge weights. The theoretical task parallelism is conveniently illustrated by the Gantt chart in Figure 4. Wunderling did not perform explicit multiple pricing since all steepest edge weights were updated after each basis change. However, since the remaining tableau columns for attractive candidates were updated, the variant was equivalent to multiple pricing. Since INVERT was not overlapped with other calculations, the implementation was not quite fully parallel. If this serial operation is excluded then the implementation is fully parallel for two processors so long as there is

not a significant load imbalance between the two BTRAN operations. However, a good simplex solver will exploit the fact that the right hand side of the standard 'update' BTRAN, $\boldsymbol{\pi}_p^T = \boldsymbol{e}_p^T B^{-1}$, contains only one nonzero (in a pivotal row), whereas it may not be worth exploiting any sparsity in the right hand side of the additional steepest edge BTRAN, $\boldsymbol{w} = \hat{\boldsymbol{a}}_q^T B^{-1}$. Thus the parallelism of the BTRAN operations may be illusory.

It is unfortunate that the only published account of Wunderling's very interesting work on parallel (and serial) simplex is his PhD Thesis [71] which is written in German. At a workshop presentation [72], he described his parallel simplex implementation and gave limited results. Wunderling's parallel scheme was implemented on a 2-processor Sun 20 and a Cray T3D using modules of Soplex, his efficient public-domain simplex solver. On general LP problems any performance improvement that was achieved was minimal and significant speed-up was only achieved for problems with large column/row ratios.

The first of Hall and McKinnon's two parallel revised simplex schemes was ASYNPLEX [39]. This corresponds to a variant of the revised simplex method in which reduced costs are computed directly (as $\hat{\boldsymbol{c}}_N^T = \boldsymbol{c}_N^T - \boldsymbol{\pi}_B^T \boldsymbol{c}_B^T B^{-1} N$) rather than being updated and Dantzig pricing is used. Ideally, one processor is devoted to INVERT and the remaining processors perform simplex iterations. The theoretical operations of these processors are illustrated by the Gantt chart in Figure 5. If a single path through the feasible region is to be followed then, clearly, two processors cannot perform basis changes simultaneously. An additional processor was introduced to act as 'basis change manager' to ensure that a processor is only allowed to start CHUZR if its basis is up-to-date and no other processor has been allowed to start CHUZR at the current basis. Although each processor has reduced costs, more up-to-date reduced costs may be known on other processors. Thus, once a processor has computed the full set of reduced costs, a small set of good candidates are communicated to a further processor. This acts as a 'column selection manager' and responds to requests made prior to FTRAN by the iteration processors, by communicating the best (known) candidate to enter the basis. Note that, due to the fact that basis changes were determined during the course of calculating reduced costs, column selection was generally performed using out-of-date reduced costs. These reduced costs became further out-of-date as a consequence of basis changes occurring during FTRAN.

ASYNPLEX was implemented on a Cray T3D using modules of Hall's highly efficient revised simplex solver and tested using four modest but representative Netlib test problems. Using between eight and twelve iteration processors, the speed with which simplex iterations were performed was in-



C: CHUZC R: CHUZR

Figure 5: ASYNPLEX

creased by a factor of about five in all cases. However, the increase in the number of iterations required to solve the problem (resulting from the use of out-of-date reduced costs) led to the speed-up in solution time ranging from 2.4 to 4.8. Although the speed-up obtained was felt to be satisfactory, it was limited by various factors. A further consequence of limited candidate persistence was the time wasted on candidates which, upon completion of FTRAN, were found to be unattractive. Efficiency was also reduced by the overhead of communicating the pivotal column to all other iteration processors, together with the duplicated work of bringing the eta file up-to-date on all processors. Numerical instability, due to the need to re-use eta vectors from the previous representation of B^{-1}, was observed for many test problems.

Hall and McKinnon's second parallel scheme was PARSMI [38]. This was developed in an attempt to address the deficiencies of ASYNPLEX [39]. In order to make realistic comparisons with good serial solvers, PARSMI updates the reduced costs and uses Devex pricing. To reduce the communication overheads, each processor is associated with one of the major computational components of the revised simplex method, with a variant of multiple pricing being used to increase the scope for parallelism. The operations performed on processors of different types and the inter-processor communications are illustrated in Figure 6, where MI represents the processors performing minor iterations and MI_CT is the single processor coordinating the MI processors.

The FTRAN processors receive sets of attractive candidates to enter the basis from the PRICE processors and apply the INVERT etas. The resulting partially completed tableau columns are distributed over the MI processors where the update etas are applied. The MI_CT processor determines the best candidate for which the up-to-date tableau column is known and coordinates CHUZR, which is distributed over the MI processes. Once the pivotal row has been determined, the MI_CT processor applies the update etas to start BTRAN. This is done very efficiently using the technique described by

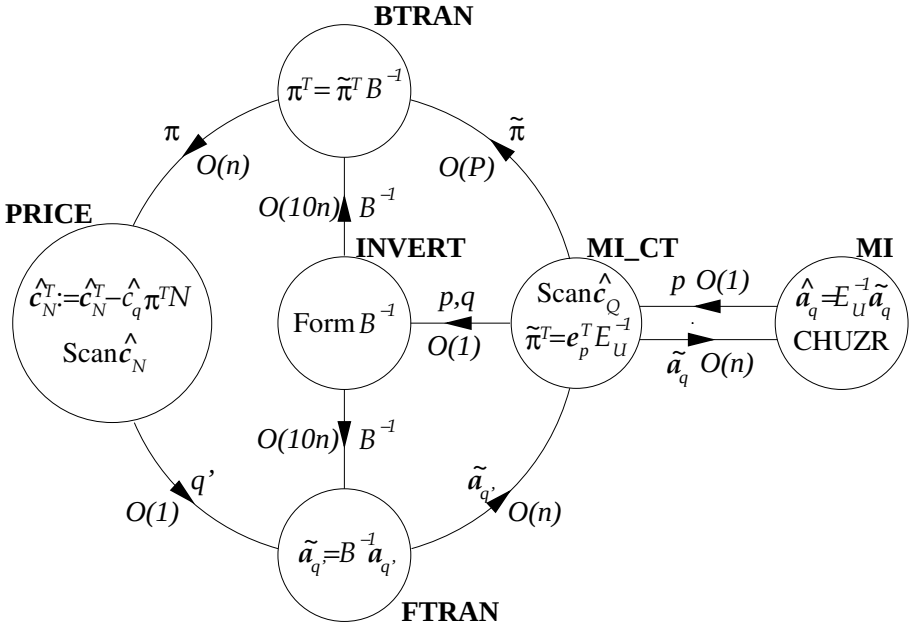
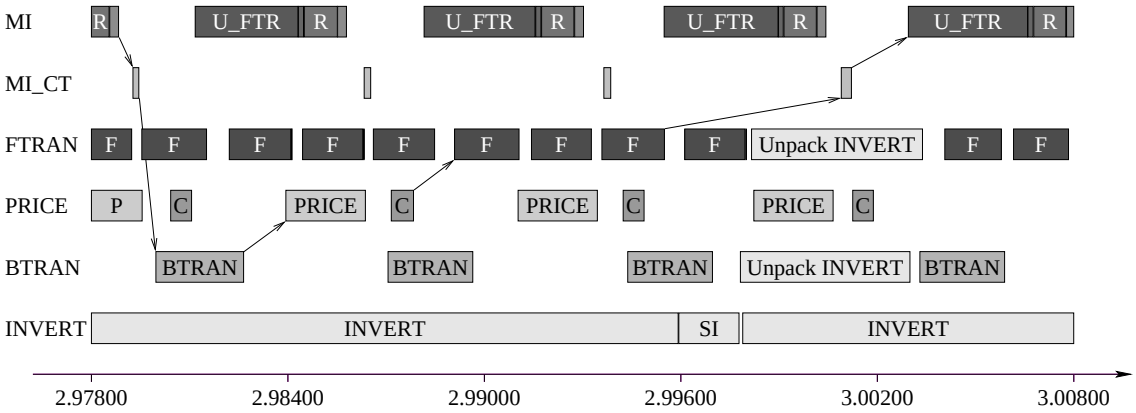


Figure 6: Operations and inter-processor communications for PARSMI

Hall and McKinnon [41]. The resulting partially formed vector $\boldsymbol{\pi}_p$ is then communicated to a BTRAN processor which applies the INVERT etas before distributing the completed vector $\boldsymbol{\pi}_p$ across the PRICE processors. After a distributed PRICE operation, the PRICE processors update the reduced costs and Devex weights before determining sets of candidates and communicating them to the FTRAN processors. One or more INVERT processors factorise the basis matrix serially and communicate the result to the FTRAN and BTRAN processors. Although presented sequentially, operations are overlapped as much as possible. For a six-processor implementation, the operation of the scheme is illustrated in Figure 7. Note that this Gantt chart was created from actual data when applying the T3D implementation of PARSMI to the LP test problem 25FV47.

As with ASYNPLEX, the implementation of PARSMI was a highly complex programming task, magnified by fact that communication times are not deterministic and the order of arrival of messages determined the operation of the program. Programming difficulties, together with numerical instability (for the same reason as with ASYNPLEX) meant that the implementation was never reliable. In the very limited results that were obtained on modest Netlib problems, using eight processors the speed-up in iteration speed of between 2.2 and 2.4 was eroded by the consequences of limited candidate persistence to yield a speed-up in solution time of between 1.7 and 1.9.



C: CHUZC F: I-FTRAN U_FTR: U-FTRAN R: CHUZR SI: Send INVERT

Figure 7: Real processor activity and illustrative communication for PARSMI

5.2.4 The simplex method on supercomputers

From the results quoted by Beasley [11] in a review of experience with the simplex method on Cray supercomputers, it may be concluded that these machines were used principally to study algorithmic performance when solving large LP problems, rather than to investigate the scope for exploiting their vector arithmetic facilities. When the latter was not used, the performance decreased by only a few tens of percent.

The most notable attempt to exploit a vector facility is reported by Forrest and Tomlin [30]. They vectorised (column-wise) PRICE by identifying those sets of columns of the constraint matrix for which the numbers of nonzeros are equal and at least 20, and storing the values and row indices in dense rectangular arrays. Using a set of moderate test problems, some of which were drawn from the Netlib set, they achieved speed-ups that are generally modest (less than 2), although for one problem the speed-up of 3.8 compares well with the figure of 4.5 which is the maximum attainable on the particular architecture. They did not consider the consequences of $\boldsymbol{\pi}$ being sparse, in which case their vectorised PRICE would involve many operations with zero. For a sparse $\boldsymbol{\pi}$, a row-oriented PRICE avoids operations with zero but, since the numbers of nonzeros in N change from one iteration to another, it is hard to imagine their vectorisation technique being of value. Forrest and Tomlin go on to consider the extent to which a vector processor can be exploited in FTRAN, BTRAN and UPDATE. In their serial revised simplex implementation, the latter has a significant computational overhead since it modifies the factors obtained by INVERT but, typically, yields a significantly smaller

eta file than would be obtained with the product form update. Estimated speed-ups of 1.5 and 2.0 for FTRAN, BTRAN and UPDATE are reported for two illustrative problems. As for other computational components, Forrest and Tomlin observe that when packing the pivotal column prior to CHUZR in 'easy' problems (where the pivotal column contains a large proportion of zeros) much time is spent identifying nonzeros and identify that this operation vectorises well. In modern revised simplex solvers, exploiting such hyper-sparsity during FTRAN leads to list of indices of (potential) nonzeros being known so the full packing operation is unnecessary. The overall effect of exploiting a vector processor is a general speed-up of up to 4, with a speed-up of 12 for one problem, relative to a (then) state-of-the-art revised simplex solver.

5.2.5 Following parallel paths

An alternative approach to exploiting parallel computation when using the simplex method is to allow each processor to follow different paths through the feasible region and, periodically, determine which processor has made the best progress and then repeat the process from the corresponding basis. The simplest possible scheme of this kind would correspond to a parallelisation of the *maximum improvement* rule for column selection. This is a criterion in which CHUZR is performed for all attractive candidates to enter the basis and the basis change performed is that which yields the greatest reduction in the objective. Unless it is performed within the context of the standard simplex tableau, as a variant for serial computation it is prohibitively expensive. However it parallelises naturally. A more practical approach that has been considered is to distribute the best candidates, one to each processor, and allow the simplex method to perform a fixed number of basis changes before determining the process that has made the best progress. Whilst this is simple to implement and has minimal communication overhead, it cannot be expected to yield worthwhile speed-up. One argument why is that, if the simplex method starts from a basis of logical variables and the optimal basis consists of structural variables, then the number of simplex iterations that must be performed is bounded below by m. Since efficient serial simplex solvers will, typically, determine the optimal basis in a small multiple of m iterations, this is an immediate bound on the speed-up that can be achieved. For some LP problems, typically those that are highly degenerate, the number of iterations required even by good solvers is a significant multiple of m. However, even for such problems, it is doubtful that the number of iterations required would be reduced sufficiently by any parallel path-following implementation.

Another approach to following parallel paths has been investigated by Maros and Mitra [55]. Rather than using the same variant of the revised simplex method to follow paths, Maros and Mitra's strategy is to start from different edges at a common vertex and use eight different pricing strategies, including Dantzig, Devex, multiple pricing and several variants of partial pricing. They refer to this as a parallel *mixed* strategy. Although Devex pricing is commonly used as a default strategy, it is well known that for certain LP problems it is far from being optimal. This is particularly so for problems with large column/row ratios and Maros and Mitra give evidence of this. Further, it is also widely acknowledged that, during the course of solving a given LP problem, the best pricing strategy is likely to change. Early in the solution process, when the basis matrix contains a high proportion of logical variables, Devex may be significantly less effective than partial or multiple pricing. Many efficient serial simplex solvers have a run-time pricing strategy that first decides whether the column/row ratio is such that partial pricing it is expected to be more efficient and, if not, starts with some cheap pricing strategy before switching to a full pricing strategy such as Devex.

Maros and Mitra implemented their mixed strategy on a twelve processor Parsytec CC-12, using an extension of their efficient FortMP simplex solver and tested it on fifteen of the largest Netlib problems. Note that the default pricing strategy for FortMP is dynamic cyclic partial multiple pricing. The mixed strategy is generally more efficient than the best single serial strategy (by an average factor of 1.19), and always more efficient than both the FortMP default strategy (by an average factor of 1.69) and Devex (by an average factor of 1.56). Whilst the the mixed strategy was more than three times faster than both the default strategy and Devex (for a different single LP problem in both cases), the average speed-up is modest. It would be interesting to compare the performance of the mixed strategy with a serial run-time mixed strategy such as that described above.

5.3 Parallelising the network simplex method

When the simplex method is applied to a minimum cost network flow problem, exploiting its structure yields the network simplex method. Although it exploits sparsity and structure very efficiently, the data structures and algorithmic requirements for implementing the network simplex method are much simpler than those when implementing the revised simplex method for general LP problems. The parallelisation of the network simplex method has been considered by several authors, including Chang *et al.* [19], Peters [62] and Barr and Hickman [10]. The latter two are of interest since they identify the value of overlapping computational components, subject to sufficient co-

ordination to ensure that the simplex method follows a single path through the feasible region. As such they predate the task-parallel approaches to the revised simplex method of Hall and McKinnon [38, 39] and Wunderling [71, 72].

5.4 Parallelising simplex-like methods

The parallelisation of other simplex-like methods for solving LP problems has also been studied. These techniques offer more immediate scope for parallelisation but, in serial, are generally uncompetitive with a good implementation of the revised simplex method. They are also frequently only of relevance to LP problems which possess a particular, but common, structure. This is either inherent, as a consequence of the underlying model, or identified by analysing the constraint matrix.

In the technique of Dantzig-Wolfe decomposition [21] for row-linked block-angular problems, the requirement to solve a (possibly large) number of independent smaller LP subproblems is a natural source of parallelism. This was identified by Ho *et al.* [45] and load-balancing issues were addressed by Gnanendran and Ho [33]. Boduroglu [16] parallelised a special case of the little-known technique of Kaul [47] for block-angular problems and obtained some impressive results, even beating a world class revised simplex solver on several standard test problems. The solution of problems with block-angular structure using parallel bundle methods is described by Medhi [58].

For column-linked block-angular LP problems, Benders decomposition [12] may be used. The requirement to solve (nested) sets of independent LP problems has been exploited in a number of parallel implementations, for example [24, 44, 60]. Rosen and Mayer [65] describe a method for row-linked block-angular problems in which, after forming the dual problem, a procedure similar to Benders decomposition yields a set of independent LP subproblems to be solved in parallel.

Work on identifying block-angular structure in general LP problems was reported by Pinar and Aykanat [64] and more recent work is reported by Aykanat *et al.* [7]. Ferris and Horn [27] discuss how block-angular structure may be identified in general LP problems prior to applying parallel bundle methods, for which they achieved an efficiency of approximately 90% when using 32 processors of a Thinking Machines CM-5.

Klabjan *et al.* [48] describe a primal-dual simplex algorithm in which several primal subproblems are solved in parallel. Their algorithm is only efficient for problems with very high column/row ratios and, for such problems, they achieve a speed-up of up to 4 on a cluster of 16 PCs.

5.5 Summary

Attempts to exploit parallelism in the simplex method have been associated with many of the leading figures in the development of efficient serial techniques for the revised simplex method. That the success relative to good serial implementations has been limited is a measure of the difficulty of the task.

None of the attempts to exploit parallelism in the simplex method has, for general large sparse LP problems, offered significantly improved performance over a good serial implementation of the revised simplex method. Parallelising the standard or revised simplex method with dense matrix algebra is uncompetitive unless a massive number of processors is used. With sparse matrix algebra, parallelising just the PRICE operation, even for LP problems with a high column/row ratio, may still be uncompetitive with the revised simplex method if the latter uses partial pricing. For problems with a high column/row ratio, if the dual simplex method is preferred to the revised simplex method with partial pricing, then parallelising PRICE has yielded a worthwhile performance improvement. The success of attempts to exploit task parallelism has been limited by lack of candidate persistence, numerical instability and the communication overheads of distributed memory multiprocessors.

6 Directions for the future

The period when most of the work on parallelising the simplex method took place ended in the late 1990s. Since then, a number of developments have taken place that affect the scope for future work in this area and its expected value.

The identification and exploitation of hyper-sparsity within the revised simplex method has led to huge improvements in the performance of serial revised simplex solvers. For hyper-sparse LP problems, the revised simplex method is preferable to interior point methods so any parallel implementation that offers significant speed-up over a good serial solver will yield advances at the cutting edge of LP solution techniques.

An important area of computational linear algebra that has seen much activity over the past ten years has been solution techniques for general unsymmetric sparse linear systems. For problems with a significant bump during INVERT, the lack of techniques for parallelising its factorisation had led to INVERT being viewed as inherently serial. This is no longer the case. Similarly, techniques for exploiting parallelism when applying a sparse fac-

tored inverse, even with a sparse right-hand-side, have also been developed and it can be expected that they will facilitate the parallelisation of FTRAN and BTRAN.

Since the late 1990s, parallel computing hardware has developed greatly. There have been advances in the availability of shared memory multiprocessors, from desktop machines with small numbers of processors to large high performance computing resources. Massive distributed memory machines have ever larger numbers of processors and total memory.

Against the backdrop of these events, there have been no significant advances in parallelising the simplex method. The scope for exploiting these developments, as well as other novel research avenues, is explored below.

6.1 The standard simplex method

Although the parallelisation of the standard simplex method with dense matrix algebra has been fully explored on modest numbers of processors and has been implemented successfully using large numbers of processors, exploiting modern massively parallel resources offers several major challenges. Implementations must be robust, offer very high scalability and address the issue of numerical stability. Techniques used by efficient serial simplex solvers to reduce the number of iterations required to solve a given LP problem should also be incorporated (as has been done in several previous implementations). For several reasons, developing a massively parallel implementation of the standard simplex method with these properties and using dense matrix algebra would be worthwhile as an exercise in parallel programming. However, as established in the analysis below, for most practical LP problems, alternative methods of solution would be very much more efficient.

For large LP test problems (with $mn \approx 10^{11}$), to store the standard simplex tableau as a full array would require of the order of 1TB of memory. This is available on many of the world's current top 500 computer systems and, at their maximum practical performance, about 10% of them would be capable of performing 100 iterations per second if each tableau entry were updated every iteration. Since the number of iterations required to solve LP problems of this size would be of order 10^5, the total solution time for most large sparse test problems would be no better than a good serial revised simplex implementation. Very few of the world's top 500 computer systems would be able to store the tableau as a full array for problems with ten times as many variables and constraints in order to solve LP problems that, if sparse, could still be expected to be solved by a good serial solver. Thus a massively parallel implementation of the standard simplex method would not be expected to offer the opportunity to solve larger sparse problems than

a serial revised simplex solver.

There are a few classes of slightly smaller sparse LP problems for which fill-in makes the memory requirement of serial revised simplex and interior point solvers prohibitive. For these problems, as well as large dense LP problems, a robust and numerically stabilised parallel implementation of the standard simplex method would be of value. Note that a dense interior point solver would have a practical computational cost of $O((m^3 + m^2 n) \log n)$ so cannot be expected to compete with the $O(m^2 n + mn^2)$ cost of the dense standard simplex method unless $n \gg m$.

For hyper-sparse LP problems, the proportion of nonzeros in standard simplex tableau columns ranges from an average of 10% to several orders of magnitude less. If the tableau update does not test whether the multiplier of the pivotal row is zero then an overwhelming majority of the floating point operations add zero so should be avoided. While skipping the update if the multiplier is zero leads to an immediate and significant serial performance improvement, for a parallel implementation it may have an impact on scalability. An investigation of this and its impact on data distribution strategies would be interesting.

As identified earlier, the work of Lentini *et al.* [50] represents the only study of the parallel sparse standard simplex method. For the LP problems they considered, the average density of the optimal tableau was found to be about 10%. For large problems, such a density would make the serial sparse standard simplex method prohibitively expensive and only with a massive number of parallel processes could it conceivably compete with a good serial revised simplex solver. For LP problems that exhibit the greatest degree of hyper-sparsity, if they are large the average tableau density still corresponds to a huge memory overhead. However, developing a massively parallel implementation of the sparse standard simplex method would represent a worthwhile challenge. It would facilitate further study of hyper-sparsity and may allow larger hyper-sparse LP problems to be solved than would be possible on a serial machine.

A massively parallel implementation of the standard simplex method would have to be robust with respect to hardware faults. This could be achieved using the techniques described in Section 2.1 to address numerically stability.

6.2 Kaul's method for block-angular LP problems

Although parallel Dantzig-Wolfe decomposition for row-linked block-angular LP problems has been investigated, Kaul's method [47] appears to be a very much more attractive means of developing a parallel solver for such problems.

Indeed, as reported in Section 5.4, impressive results have been obtained for a special case of block-angular problems. Kaul's method corresponds to the revised simplex method, with the structure of the LP problem exploited by expressing the inverse of the basis matrix in terms of the inverses of submatrices corresponding to the blocks in the LP problem. The structure of the problem is also exploited in PRICE which is performed as a set of independent sparse matrix vector products. Efficient pricing techniques such as Devex and approximate steepest edge are readily incorporated, setting it apart from Dantzig-Wolfe. A parallel implementation of Kaul's method for general row-linked block-angular problems, with an efficient approximate edge weight pricing strategy and sparse factored inverses of the basis matrix submatrices, offers considerable scope for the highly efficient solution of practical large block-angular LP problems.

6.3 Primal revised simplex method for problems with a large column/row ratio

Techniques of partial pricing mean that parallelising the computationally dominant full PRICE operation for problems with a large column/row ratio is not sufficient to achieve worthwhile speed-up. However, for such problems, when edge weight pricing strategies are particularly important in terms of reducing the number of simplex iterations, excessive parallel resources may not be required for a solver with full parallel PRICE to compete with an efficient revised simplex solver using partial or multiple pricing. No such comparisons have been made. If the parallel resources were also used to update and apply a dense representation of B^{-1} then there may be a subclass of problems with a large column/row ratio which compete with a serial revised simplex solver with a sparse factored inverse and partial or multiple pricing.

6.4 Revised simplex with a factored inverse

To date, the brave but unsuccessful work by Shu [66] represents the only real attempt to a develop anything like a full data-parallel implementation of the revised simplex method using a sparse factored inverse. Following the major advances since this time, the state of the art in parallel techniques for the factorisation of general sparse matrices is represented by, for example, SuperLU from Berkeley [23, 51] and MUMPS from Lyon-Toulouse [5, 6]. Many features distinguish the factorisation and linear system solution requirements in the revised simplex method. The fact that the matrix being factored is often highly reducible can be exploited in many ways. It has been shown by Amestoy *et al.* [4, 3] that general preprocessing and reordering techniques

can combine numerical accuracy and reducibility detection in the context of general sparse solvers. For hyper-sparse LP problems, the overwhelming cost of INVERT is the triangularisation phase and results reported by Hall [36] using a serial simulation of a parallel triangularisation scheme are very encouraging. With the fruits of these advances as yet un-tapped in the context of the revised simplex method, the time is right for more work in this area.

The task parallel scheme SYNPLEX [35] is variant of PARSMI developed by Hall that is numerically stable. This has been achieved by overlapping INVERT with multiple BTRAN, multiple PRICE, CHUZC and multiple FTRAN. SYNPLEX has the added advantage of being algorithmically independent of both the number of processors and the time required to transmit data between them. A prototype implementation by Hall on a shared memory Sun Fire E15k has achieved a speed-up of between 1.8 and 3.0 when using eight processors. The limitations due to lack of candidate persistence have been reduced but cannot be eliminated. Load balancing problems when overlapping INVERT have motivated, and should be alleviated by, the development of a parallel INVERT.

7 Conclusions

Despite its value in both the academic and commercial worlds, there has been no parallelisation of the simplex method that offers significantly improved performance over a good serial implementation of the revised simplex method for general large scale LP problems. The successes in terms of speed-up typically correspond to parallelisation of implementations that are partly or wholly inefficient in serial.

Attention has mainly been directed at inefficient simplex and simplex-like techniques in which exploiting parallelism is relatively straightforward. There have been only a few discussions of schemes by which parallelism might be exploited by (variants of) the revised simplex method with a factored inverse. Fewer still of these schemes have been implemented and none has met with any real success when applied to a full range of LP problems.

Despite the large amount of work reviewed in Section 5, the reference list in this paper is not exhaustive. The five articles that stand out as best representing the state of the art are those by Bixby and Martin [15], Eckstein *et al.* [25], Forrest and Tomlin [30], Lentini *et al.* [50] and Shu [66] and, together with this paper, should be viewed as required reading for anyone contemplating the exploitation of parallelism in the simplex method.

In Section 6, several promising avenues for future work are identified. Without underestimating the challenge of implementing them, the rewards

for any success are now all the greater since the identification of hyper-sparsity has given the revised simplex method renewed prominence. The time is ripe for further work on developing parallel simplex method implementations of practical value.

The author would like to thank Patrick Amestoy for his insight into the developments in techniques for parallel factorisation and solution techniques for general unsymmetric sparse linear systems, and the potential for their application to the revised simplex method. Comments and suggestions from the referees have also been very valuable and have improved both the content and focus of the paper.

References

[1] A. Agrawal, G. E. Blelloch, R. L. Krawitz, and C. A. Phillips. Four vector-matrix primitives. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–302, 1989.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, Reston, Va., 1967.

[3] P. Amestoy, X. S. Li, and E. G. Ng. Diagonal Markowitz scheme with local symmetrization. *SIAM Journal on Matrix Analysis and Applications*, 29(1):228–244, 2007.

[4] P. Amestoy, S. Pralet, and X. S. Li. Unsymmetric orderings using a constrained Markowitz scheme. *SIAM Journal on Matrix Analysis and Applications*, 29(1):302–327, 2007.

[5] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[6] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.

[7] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, 2004.

[8] D. A. Babaev and S. S. Mardanov. A parallel algorithm for solving linear programming problems. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 31(1):86–95, 1991.

[9] E.-S. Badr, M. Moussa, K. Papparrizos, N. Samaras, and A. Sifaleras. Some computational results on MPI parallel implementations of dense simplex method. *Transactions on Engineering, Computing and Technology*, 17:228–231, December 2006.

[10] R. S. Barr and B. L. Hickman. Parallel simplex for large pure network problems: Computational testing and sources of speedup. *Operations Research*, 42(1):65–80, 1994.

[11] J. E. Beasley. Linear programming on Cray supercomputers. *Journal of the Operatrional Research Society*, 41(2):133–139, 1990.

[12] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[13] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[14] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and practice closing the gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization: Methods, Theory and Applications*, pages 19–49. Kluwer, The Netherlands, 2000.

[15] R. E. Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12:45–56, 2000.

[16] İ. İ. Boduroğlu. *Scalable Massively Parallel Simplex Algorithms for Block-Structured Linear Programs*. PhD thesis, GSAS, Columbia University, New York, NY, 1997.

[17] T. B. Boffey and R. Hay. Implementing parallel simplex algorithms. In *CONPAR 88*, pages 169–176, Cambridge, UK, 1989. Cambridge University Press.

[18] W. J. Carolan, J. E. Hill, J. L. Kennington, S. Niemi, and S. J. Wichmann. An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research*, 38(2):240–248, 1990.

[19] M. D. Chang, M. Engquist, R. Finkel, and R. R. Meyer. A parallel algorithm for generalized networks. *Annals of Operations Research*, 14:125–145, 1988.

[20] Z. Cvetanovic, E. G. Freedman, and C. Nofsinger. Efficient decomposition and performance of parallel PDE, FFT, Monte-Carlo simulations, simplex, and sparse solvers. *Journal of Supercomputing*, 5:19–38, 1991.

[21] G. B. Dantzig. The decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

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

[23] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[24] M. A. H. Dempster and R. T. Thompson. Parallelization and aggregation of nested Benders decomposition. *Annals of Operations Research*, 81:163–187, 1998.

[25] J. Eckstein, İ. İ. Boduroğlu, 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.

[26] D. J. Evans and M. Hatzopoulos. A parallel linear system solver. *International Journal of Computer Mathematics*, 7:227–238, 1979.

[27] M. C. Ferris and J. D. Horn. Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 80:35–61, 1998.

[28] R. A. Finkel. Large-grain parallelism—three case studies. In L. H. Jamieson, D. Gannon, and R. J. Douglas, editors, *The Characteristics of Parallel Algorithms*, pages 21–63. MIT Press, Cambridge, MA, 1987.

[29] J. J. Forrest and D. Goldfarb. Steepset-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992.

[30] 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.

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

[32] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.

[33] S. K. Gnanendran and J. K. Ho. Load balancing in the parallel optimization of block-angular linear programs. *Mathematical Programming*, 62:41–67, 1993.

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

[35] J. A. J. Hall. SYNPLEX, a task-parallel scheme for the revised simplex method. Contributed talk at the Second International Workshop on Combinatorial Scientific Computing (CSC05), June 2005.

[36] J. A. J. Hall. Parallel matrix inversion for the revised simplex method - a study. Contributed talk at the CERFACS Sparse Days Meeting, June 2006.

[37] 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.

[38] 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.

[39] J. A. J. Hall and K. I. M. McKinnon. ASYNPLEX, an asynchronous parallel revised simplex method algorithm. *Annals of Operations Research*, 81:27–49, 1998.

[40] J. A. J. Hall and K. I. M. McKinnon. Exploiting hyper-sparsity in the revised simplex method. Technical Report MS99-014, Department of Mathematics and Statistics, University of Edinburgh, 1999.

[41] J. A. J. Hall and K. I. M. McKinnon. Hyper-sparsity in the revised simplex method and how to exploit it. *Computational Optimization and Applications*, 32(3):259–283, December 2005.

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

[43] R. V. Helgason, L. J. Kennington, and H. A. Zaki. A parallelisation of the simplex method. *Annals of Operations Research*, 14:17–40, 1988.

[44] R. S. Hiller and J. Eckstein. Stochastic dedication: designing fixed income portfolios using massively parallel Benders decomposition. *Management Science*, 39(11):1422–1438, 1993.

[45] J. K. Ho, T. C. Lee, and R. P. Sundarraj. Decomposition of linear programs using parallel computation. *Mathematical Programming*, 42:391–405, 1988.

[46] 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.

[47] R. N. Kaul. An extension of generalized upper bounding techniques for linear programming. Technical Report ORC 65-27, O. R. Center U. C. Berkley, San Fransisco, CA, 1965.

[48] D. Klabjan, E. L. Johnson, and G. L. Nemhauser. A parallel primal-dual simplex algorithm. *Operations Research Letters*, 27:47–55, 2000.

[49] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2nd edition, 2003.

[50] M. Lentini, A. Reinoza, A. Teruel, and A. Guillen. SIMPAR: a parallel sparse simplex. *Computational and Applied Mathematics*, 14(1):49–58, 1995.

[51] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.

[52] J. Luo and G. L. Reijns. Linear programming on transputers. In J. van Leeuwen, editor, *Algorithms, Software, Architecture*, volume A-12 of *IFIP Transactions A (Computer Science and Technology)*, pages 525–534. Elsevier, 1992.

[53] J. Luo, G. L. Reijns, F. Bruggeman, and G. R. Lindfield. A survey of parallel algorithms for linear programming. In E. F. Deprettere and A.-J. van der Veen, editors, *Algorithms and Parallel VLSI Architectures*, volume B, pages 485–490. Elsevier, 1991.

[54] H. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–296, 1957.

[55] I. Maros and G. Mitra. Investigating the sparse simplex algorithm on a distributed memory multiprocessor. *Parallel Computing*, 26:151–170, 2000.

[56] K. I. M. McKinnon and F. Plab. A modified Markowitz criterion to increase parallelism in inverse factors of sparse matrices. Technical report, Department of Mathematics and Statistics, University of Edinburgh, 1997.

[57] K. I. M. McKinnon and F. Plab. An upper bound on parallelism in the forward transformation within the revised simplex method. Technical report, Department of Mathematics and Statistics, University of Edinburgh, 1997.

[58] D. Medhi. Parallel bundle-based decomposition algorithm for large-scale structured mathematical programming problems. *Annals of Operations Research*, 22:101–127, 1990.

[59] H. D. Mittelmann. Benchmarks for optimization software. http://plato.asu.edu/bench.html, July 2006.

[60] S. N. Nielsen and S. A. Zenios. Scalable Benders decomposition for stochastic linear programming. *Parallel Computing*, 23:1069–1088, 1997.

[61] W. Orchard-Hays. *Advanced Linear programming computing techniques*. McGraw-Hill, New York, 1968.

[62] J. Peters. The network simplex method on a multiprocessor. *Networks*, 20:845–859, 1990.

[63] 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.

[64] A. Pinar and C. Aykanat. An effective model to decompose linear programs for parallel solution. In J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, editors, *Applied Parallel Computing*, volume 1184 of *Lecture Notes in Computer Science*, pages 592–601. Springer, 1996.

[65] J. B. Rosen and R. S. Maier. Parallel solution of large-scale, block-angular linear programs. *Annals of Operations Research*, 22:23–41, 1990.

[66] W. Shu. Parallel implementation of a sparse simplex algorithm on MIMD distributed memory computers. *Journal of Parallel and Distributed Computing*, 31(1):25–40, November 1995.

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

[68] U. H. Suhl and L. M. Suhl. Computing sparse LU factorizations for large-scale linear programming bases. *ORSA Journal on Computing*, 2(4):325–335, 1990.

[69] M. E. Thomadakis and J.-C. Liu. An efficient steepest-edge simplex algorithm for SIMD computers. In *International Conference on Supercomputing*, pages 286–293, 1996.

[70] J. A. Tomlin. Pivoting for size and sparsity in linear programming inversion routines. *J. Inst. Maths. Applics*, 10:289–295, 1972.

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

[72] R. Wunderling. Parallelizing the simplex algorithm. ILAY Workshop on Linear Algebra in Optimzation, Albi, April 1996.

[73] G. Yarmish. *A Distributed Implementation of the Simplex Method*. PhD thesis, Polytechnic University, Brooklyn, NY, March 2001.

[74] S. A. Zenios. Parallel numerical optimization: current status and annotated bibliography. *ORSA Journal on Computing*, 1(1):20–43, Winter 1989.