High performance simplex solver

Qi Huangfu

Doctor of Philosophy
University of Edinburgh
2013

Abstract

The dual simplex method is frequently the most efficient technique for solving linear programming (LP) problems. This thesis describes an efficient implementation of the sequential dual simplex method and the design and development of two parallel dual simplex solvers.

In serial, many advanced techniques for the (dual) simplex method are implemented, including sparse LU factorization, hyper-sparse linear system solution technique, efficient approaches to updating LU factors and sophisticated dual simplex pivoting rules. These techniques, some of which are novel, lead to serial performance which is comparable with the best public domain dual simplex solver, providing a solid foundation for the simplex parallelization.

During the implementation of the sequential dual simplex solver, the study of classic LU factor update techniques leads to the development of three novel update variants. One of them is comparable to the most efficient established approach but is much simpler in terms of implementation, and the other two are specially useful for one of the parallel simplex solvers. In addition, the study of the dual simplex pivoting rules identifies and motivates further investigation of how hyper-sparsity maybe promoted.

In parallel, two high performance simplex solvers are designed and developed. One approach, based on a less-known dual pivoting rule called suboptimization, exploits parallelism across multiple iterations (PAMI). The other, based on the regular dual pivoting rule, exploits purely single iteration parallelism (SIP). The performance of PAMI is comparable to a world-leading commercial simplex solver. SIP is frequently complementary to PAMI in achieving speedup when PAMI results in slowdown.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Qi Huangfu*)

Acknowledgements

I would like to thank my PhD supervisor, Julian Hall for his support and encouragement over the years. I would also like to thank other staff members in the School of Maths for their help and conversation, in particular, my second PhD supervisor Jacek Gondzio.

I am grateful to the Format International Ltd. and NAIS for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Linear programming (LP) has been widely and successfully used in many practical areas since the introduction of the simplex method in the 1950s. Though an alternative approach, the interior point method (IPM) has become competitive and popular since the 1980s, the advanced dual revised simplex method is still frequently the preferred choice for solving LP problems. In a cover story [16] of one issue of the New Scientist magazine in 2012, the simplex method is called "the algorithm that runs the world".

The crucial advantage of the modern simplex method is its ability to directly exploit the inherent (hyper-)sparsity of LP problems. This ability has been emphasised and enhanced throughout the development of the (revised) simplex method, especially, by the sparse LU factorization and the hyper-sparse linear system solution technique. Also importantly, the introduction and application of advanced dual simplex method algorithmic variants, particularly the dual steepest-edge (DSE) and bound flipping ratio test (BFRT), have promoted the performance of the simplex method dramatically since the 1990s. Both of these techniques underpin the efficiency of modern simplex solvers.

The simplex method has been parallelized many times. Most of the existing parallelizations are based on the tableau simplex method, using dense matrix algebra. This generally achieves remarkable (from tens to up to a thousand) speedup, but the overall performance is still inferior to a good sparsity-exploiting sequential implementation of the revised simplex method. On the other hand, the simplex parallelization based on the revised simplex method has been considered relatively little and less successfully in terms of speedup. Nevertheless, it is still worthwhile since it corresponds to the computationally efficient serial technique. Because of the generally poor speedup gained with the revised simplex method, it has been considered not suitable for parallelization.

However, the context for simplex parallelization has changed. Since the introduction of DSE and BFRT in the 1990s, the preferred simplex variant to use has changed from the primal simplex algorithm to the dual, but the only published work on the dual simplex parallelization is due to Bixby and Martin [8]. Although it appeared in the early 2000s, their dual simplex parallelization neither included the BFRT nor the hyper-sparse linear system solution technique. In terms of the application scope, in the past (the 1990s), parallelization was aimed at dedicated high performance computers to achieve the best performance; nowadays, when every desktop computer is a multi-core machine, any speedup is desirable in terms of solution time reduction for daily usage. Therefore, the simplex method, especially, the dual revised simplex method deserves renewed parallelization effort.

To achieve a worthwhile simplex parallelization, it should be based on a good sequential simplex solver. Though there are many public domain simplex implementations, they are either too complicated to be extended for parallelization or too simple to be based on. Thus, a sequential dual simplex solver, named HSOL, has been implemented from scratch. It includes the sparse LU factorization, hyper-sparse linear system solution technique, efficient approaches to updating LU factors and sophisticated dual revised simplex pivoting rules. These techniques, some of which are novel, lead to serial performance which is comparable with one of the best public domain simplex solvers, providing a solid foundation for the parallelization. Then, based on the sequential solver, two dual simplex parallel solvers are designed and developed.

This thesis reports the implementation experience of the sequential dual simplex method and the development of the parallel dual simplex solvers.

1.1 Structure of this thesis

This thesis discusses three major topics: the linear system solution techniques in Chapter 2, the advanced dual simplex method in Chapter 3 and the simplex parallelization in Chapter 4. For each of these topics, the corresponding chapter firstly gives brief introduction to existing techniques and then reports novel developments during this research. Chapter 5 summarizes the contributions of this thesis and provides future work suggestions.

Chapter 2 discusses linear system solution techniques. Solving linear systems is a key computational component of the revised simplex method. Though the basic logic (Gaussian elimination) is well established, the challenge is how to properly exploit sparsity and hyper-sparsity to achieve efficiency. During this research, classic techniques including the Suhl and Suhl sparse LU factorization, the hyper-sparse linear system solution technique and the Forrest-Tomlin LU factor update procedure are implemented. Details of these techniques are introduced in Sections 2.2, 2.3 and 2.4 respectively. In particular, a detailed description of the Forrest-Tomlin update relating to maintaining the ability to exploit hyper-sparsity is provided.

Study and implementation of classic update procedures led to the development of three novel update techniques. One of these, the middle product form update (MPF), is comparable with the Forrest-Tomlin update in terms of efficiency, but much simpler for software implementation. The other two, the alternate product form update (APF) and the collective Forrest-Tomlin update (CFT) are key components of one of the simplex parallelization framework. Design and development of these three novel simplex update techniques are reported in Section 2.5.

Chapter 3 introduces the advanced dual simplex method. Sections 3.1 and 3.2 introduce the basic concepts of the simplex algorithm and the advanced computational components of the modern dual revised simplex method. Particularly, the implementation of dual steepest-edge (DSE) algorithm, including an adaption of the hyper-sparse technique for the primal simplex algorithm, is discussed in Section 3.2.2. A detailed description of the bound-flipping ratio test (BFRT) and the Harris two-pass ratio test is provided in Section 3.2.3.

As an enhancement to the dual ratio test, cost perturbation is also included in the implementation. The application of cost perturbation leads to the discovery of a phenomenon called hyper-sparsity promotion. With cost perturbation, the solution of linear systems during the simplex iterations became significantly sparser for certain family of LP problems. This phenomenon is thus studied to reveal the underlying reason. The investigation of the phenomenon also leads to two alternate approaches, which promote the hyper-sparsity further. Details are presented in Section 3.3.

Chapter 4 firstly introduces (in Section 4.1) and analyses (in Section 4.2) existing simplex parallelization attempts and then describes the design and development of two parallel dual simplex solvers.

A relatively sophisticated dual parallel solver, called PAMI (parallelism across multiple iterations) is documented in Section 4.3. PAMI is based on a less-known dual pivoting strategy called suboptimization. Suboptimization chooses a bunch of candidates and then works with them until none is attractive. Though it is inevitable that the suboptimization leads to a generally worse simplex path, it provides more scope for task parallelism. Details of the PAMI parallelization framework are reported in Section 4.3. In addition to the design, a pivot candidate quality control mechanism is experimented and reported. The performance of PAMI is comparable with a world-leading commercial dual simplex solver.

A relatively simple dual parallel solver, called SIP (single iteration parallelism), based on the regular dual pivoting rule (DSE), is described in Section 4.4. Though the achieved performance in average is worse than PAMI, the SIP is found complementary to PAMI in achieving speedup when PAMI results in slowdown.

1.2 The developing environment

The development of HSOL was achieved using the C++ programming language, and the parallelization was achieved with OpenMP directives.

C++ is chosen for several reasons, particularly, for its straightforward memory management, which is especially useful in developing the LU factorization. Also important, using C++ provides the opportunity of an evolutionary development process, from a proof-of-idea prototype with simple but less efficient data structures to an efficient elaborated implementation. OpenMP is chosen partially because the resulting simplex parallelization framework is based on a memory-sharing model, and partially because of its simple and powerful parallel directive, especially the task directive introduced in OpenMP version 3.0 in 2008.

A good compiler is important for software performance, especially the performance of the parallelization part. HSOL is compiled using the Intel C++ Compiler (version 13.0) with OpenMP 3.1.

The hardware used for the assessing the performance of HSOL and its parallelization is `compute64b` of the School of Maths in the University of Edinburgh. It is a workstation with 16 (Intel Xeon E5620, 2.4GHz) cores.

1.3 The reference set of testing LP problems

Throughout this report, a selection of 30 LP problems called the *reference set* as listed in Table 1.1, is used for assessing the performance of HSOL and its parallelization.

Most of LP problems in the reference set are taken from a comprehensive list [48] provided by Mittelmann. Mittelmann's list is a developing list consisting of various representative LP problems.

The reference set reflects the wide spread of behaviour of the revised simplex method, including the dimension of the linear systems (number of rows), the density of LU factors

MODEL	#row	#col	#nnz	nnz/col	col/row	FTRAN	BTRAN
CRE-B	9648	72447	256095	3.53	7.51	100	83
DANO3MIP_LP	3202	13873	79655	5.74	4.33	1	6
DBIC1	43200	183235	1038761	5.67	4.24	100	83
DCP2	32388	21087	559390	26.53	0.65	100	97
DFL001	6071	12230	35632	2.91	2.01	34	57
FOME12	24284	48920	142528	2.91	2.01	45	58
FOME13	48568	97840	285056	2.91	2.01	100	98
KEN-18	105127	154699	358171	2.32	1.47	100	100
L30	2701	15380	51169	3.33	5.69	10	8
LINF_520C	93326	69004	566193	8.21	0.74	10	11
LP22	2958	13434	65560	4.88	4.54	13	22
MAROS-R7	3136	9408	144848	15.40	3.00	5	13
MOD2	35664	31728	198250	6.25	0.89	46	68
NS1688926	32768	16587	1712128	103.22	0.51	72	100
NUG12	3192	8856	38304	4.33	2.77	1	20
PDS-40	66844	212859	462128	2.17	3.18	100	98
PDS-80	129181	426278	919524	2.16	3.30	100	99
PDS-100	156243	505360	1086785	2.15	3.23	100	99
PILOT87	2030	4883	73152	14.98	2.41	10	19
QAP12	3192	8856	38304	4.33	2.77	2	15
SELF	960	7364	1148845	156.01	7.67	0	2
SGPF5Y6	246077	308634	828070	2.68	1.25	100	100
STAT96V4	3174	62212	490473	7.88	19.60	73	31
STORMG2-125	66185	157496	418321	2.66	2.38	100	100
STORMG2-1000	528185	1259121	3341696	2.65	2.38	100	100
STP3D	159488	204880	662128	3.23	1.28	95	70
TRUSS	1000	8806	27836	3.16	8.81	37	2
WATSON_1	201155	383927	1052028	2.74	1.91	100	100
WATSON_2	352013	671861	1841028	2.74	1.91	100	100
WORLD	35510	32734	198793	6.07	0.92	41	61

Table 1.1: The reference set consisting of 30 LP problems

(average number of non-zeros per column), the relative cost of matrix vector multiplication, (the ratio of number of columns to number of rows) and hyper-sparsity (indicated by the last two columns). The column headed FTRAN and BTRAN provides the information (the proportion of the results of FTRAN and BTRAN with density less than 10%) concerning the hyper-sparsity property of each LP problem. According to Hall and McKinnon [32], a solution of a linear system (FTRAN or BTRAN) is called hyper-sparse if its sparser than 10%. An LP problem is called a hyper-sparse model if the occurrence of hyper-sparse results is greater than 60%. According to this measurement, half of the reference set are hyper-sparse LP problems.

Two established simplex solvers, the commercial solver, Cplex 12.4 [37] and the public domain solver, Clp 1.14 [10] are used for comparing with HSOL to assess its performance.

Chapter 2

Solving linear systems

This chapter introduce classic techniques and novel developments for solving linear systems in the context of the revised simplex method. Section 2.1 sets the background for the whole chapter. Sections 2.2, 2.3 and 2.4 briefly review classic LU factorization, solving and updating techniques. Section 2.5 reports three novel simplex update techniques: two product form update variants and one Forrest-Tomlin update variant for the parallel simplex method introduced in Chapter 4. The design and development of the novel simplex update methods have been submitted as a journal paper [36].

2.1 Background

Solve. Solving linear systems with the basis matrix B in the form of

$$B\hat{\boldsymbol{x}} = \boldsymbol{x} \tag{2.1}$$

$$\hat{\boldsymbol{x}}^{\mathrm{T}}B = \boldsymbol{x}^{\mathrm{T}} \text{ (or } B^{\mathrm{T}}\hat{\boldsymbol{x}} = \boldsymbol{x}) \tag{2.2}$$

is a fundamental task in the revised simplex method. The linear systems (2.1) and (2.2) are called the *forward* system and the *transposed* system respectively. In particular, each revised simplex iteration requires the solution of the forward system

$$B\hat{\boldsymbol{a}}_q = \boldsymbol{a}_q \tag{2.3}$$

and the transposed system

$$\hat{\boldsymbol{e}}_p^{\mathrm{T}}B = \boldsymbol{e}_p^{\mathrm{T}}. \tag{2.4}$$

where $\boldsymbol{a}_q$ is q^{th} column of coefficient matrix A and $\boldsymbol{e}_p$ is the p^{th} column of the identity matrix.

Inversion. Solving linear systems requires the inverse of the basis matrix B, which is often represented by LU factors

$$B = LU, \text{ so that } B^{-1} = U^{-1}L^{-1}, \tag{2.5}$$

where L and U are lower and upper triangular matrices. The procedure to obtain the basis inverse representation is referred to as INVERT, *basis inversion* or simply *inversion*. It is also called *LU factorization* when the basis inverse is represented by LU factors. The LU factorization technique is introduced in Section 2.2. Linear system solution techniques with LU factors, especially the hyper-sparse solution technique, are discussed in Section 2.3.

Update. At the end of each simplex iteration, the basis matrix B is updated by replacing its p^{th} column with the column $\boldsymbol{a}_q$ from the coefficient matrix A as

$$B := B + (\boldsymbol{a}_q - B\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}}. \tag{2.6}$$

Often the basis inverse representation B^{-1} is updated accordingly until a fresh inverse representation is needed. The procedure to update the basis inverse representation is normally named the *simplex update*. The classic update approaches are discussed in Section 2.4. The design and development of novel update variants are presented in Section 2.5.

2.2 Constructing LU factors

This section introduces sparse LU factorization techniques for the revised simplex method.

2.2.1 The eta matrix

The *eta* matrix is the core constituent element of LU factors. It is introduced here alone to avoid any distraction when talking about detailed LU factorization techniques.

An eta matrix E differs from the identity matrix in only its p^{th} column as

$$E = \begin{bmatrix} 1 & & \eta_1 & & \\ & \ddots & \vdots & & \\ & & \eta_p & & \\ & & \vdots & \ddots & \\ & & \eta_m & & 1 \end{bmatrix}, \tag{2.7}$$

where η_p is referred to as the *pivotal entry* or simply the *pivot*, and the remaining entries in the p^{th} *pivotal column* form the *eta vector* $\boldsymbol{\eta}$, for which the p^{th} entry is zero. The inverse of an eta matrix is also an eta matrix

$$E^{-1} = \begin{bmatrix} 1 & & -\eta_1/\eta_p & & \\ & \ddots & \vdots & & \\ & & 1/\eta_p & & \\ & & \vdots & \ddots & \\ & & -\eta_m/\eta_p & & 1 \end{bmatrix}, \tag{2.8}$$

whose pivot and eta vector are $1/\eta_p$ and $(-1/\eta_p) \times \boldsymbol{\eta}$ respectively. Therefore solving the linear systems with an eta matrix can be arranged as simple linear algebra operations. Specifically, the operation to solve the forward system (2.1) and the transposed system (2.2) with E are given by

$$\textsc{eta-ftran}\ (\boldsymbol{x} := E^{-1}\boldsymbol{x}): \quad x_p := x_p/\eta_p \quad \text{and then} \quad \boldsymbol{x} := \boldsymbol{x} - x_p\boldsymbol{\eta} \tag{2.9}$$

$$\text{and} \quad \textsc{eta-btran}\ (\boldsymbol{x} := E^{-\mathrm{T}}\boldsymbol{x}): \quad x_p := (x_p - \boldsymbol{x}^{\mathrm{T}}\boldsymbol{\eta})/\eta_p \tag{2.10}$$

respectively. Because the linear algebra operation is performed *in place* with the right-hand-side vector $\boldsymbol{x}$, the solving operation is called *transformation*. Obviously, transformation with the eta matrix only requires the pivot η_p and the eta vector $\boldsymbol{\eta}$ of E, so that in actual implementation, E^{-1} is also represented by η_p and $\boldsymbol{\eta}$.

Within the context of revised simplex method (as a result of LU factorization and simplex update), the basis matrix is always expressed as a product of a series of eta matrices so that $B = E_1 E_2 \dots E_k$, and its inverse is given by $B^{-1} = E_k^{-1} \dots E_2^{-1} E_1^{-1}$. Therefore, solving the forward system (2.1) requires applying all the eta transformations E_1 through E_k forwardly to $\boldsymbol{x}$ as $B^{-1}\boldsymbol{x} = E_k^{-1} \dots E_2^{-1} E_1^{-1}\boldsymbol{x}$, and thus is called *forward transform* or FTRAN. Similarly, solving the transposed system (2.2) is achieved by applying all the eta transformations in the *reverse* order and thus is traditionally called *backward transform* or BTRAN. The elementary operations given by (2.9) and (2.10) with a single eta matrix are called ETA-FTRAN and ETA-BTRAN respectively.

2.2.2 Gaussian elimination

It is well known that a non-singular $m \times m$ square matrix (for example the basis matrix), can be decomposed into LU factors by $m - 1$ iterations of Gaussian elimination.

Starting from $B^{(1)} = B$, at each stage k, the Gaussian elimination works on an active partition of $B^{(k)}$, where the k^{th} row is used to eliminate the sub-diagonal elements of the k^{th} column, resulting in $B^{(k+1)}$ with a smaller active partition. The elimination work at each stage k can be achieved by subtracting row k of $B^{(k)}$ from each row i below with a multiplier e_{ik} as

$$e_{ik} = \frac{b_{ik}}{b_{kk}}. \tag{2.11}$$

The elimination can also be expressed as an *eta* matrix E_k, so that $E_k B^{(k)} = B^{(k+1)}$. When $m = 6$ and $k = 3$, the elimination matrix and the active partitions (underlined elements) before and after elimination can be illustrated as

$$\begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & -e_{43} & 1 & & \\ & & -e_{53} & & 1 & \\ & & -e_{63} & & & 1 \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ & & \underline{b_{33}} & \underline{b_{34}} & \underline{b_{35}} & \underline{b_{36}} \\ & & \underline{b_{43}} & \underline{b_{44}} & \underline{b_{45}} & \underline{b_{46}} \\ & & \underline{b_{53}} & \underline{b_{54}} & \underline{b_{55}} & \underline{b_{56}} \\ & & \underline{b_{63}} & \underline{b_{64}} & \underline{b_{65}} & \underline{b_{66}} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ & & b_{33} & b_{34} & b_{35} & b_{36} \\ & & & \underline{b_{44}} & \underline{b_{45}} & \underline{b_{46}} \\ & & & \underline{b_{54}} & \underline{b_{55}} & \underline{b_{56}} \\ & & & \underline{b_{64}} & \underline{b_{65}} & \underline{b_{66}} \end{bmatrix}. \tag{2.12}$$

Repeatedly applying Gaussian elimination to $B^{(k)}$ eventually results in $B^{(m)}$, which is an upper triangular matrix. This is the factor

$$U = B^{(m)} = E_{m-1} \dots E_2 E_1 B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ & & b_{33} & b_{34} & b_{35} & b_{36} \\ & & & b_{44} & b_{45} & b_{46} \\ & & & & b_{55} & b_{56} \\ & & & & & b_{66} \end{bmatrix}. \tag{2.13}$$

As each elimination matrix E_k is an *eta* matrix, the inverse of $E_{m-1} \dots E_2 E_1$ is easily available as the product $E_1^{-1}E_2^{-1}\dots E_{m-1}^{-1}$, yielding the lower triangular factor

$$L = E_1^{-1}E_2^{-1}\dots E_{m-1}^{-1} = \begin{bmatrix} 1 & & & & & \\ e_{21} & 1 & & & & \\ e_{31} & e_{32} & 1 & & & \\ e_{41} & e_{42} & e_{43} & 1 & & \\ e_{51} & e_{52} & e_{53} & e_{54} & 1 & \\ e_{61} & e_{62} & e_{63} & e_{64} & e_{65} & 1 \end{bmatrix}. \tag{2.14}$$

Note that each column k of the factor L is the eta vector of the corresponding E_k^{-1}. It is a feature of the triangular matrix rather than a happy "coincidence". While the factor L is constructed by multiplying $m-1$ eta matrices, the factor U can also be expressed as a product of m eta matrices by taking (in reverse order) each diagonal entry u_{kk} as the pivot and the rest of column $\boldsymbol{u}_k$ as the eta vector.

Therefore, the basis matrix is represented by LU factors, which are further expressed as $2m-1$ eta matrices or, for completeness and symmetry, appending an identity matrix as the last L eta matrix, as $2m$ eta matrices

$$B = LU = \prod_{i=1}^{m} L_i \prod_{i=m}^{1} U_i, \tag{2.15}$$

where eta matrices L_i and U_i correspond to the i^{th} column of L and U respectively. During the Gaussian elimination, L_i and U_i are formed at the i^{th} stage by storing sub-diagonal entries of column i to the eta vector of L_i, the diagonal entry as the pivot of U_i and the rest entries to the eta vector of U_i.

Although the technique to build the LU factors by Gaussian eliminations (LU factorization) is well established, it needs to be adapted to deal with the sparse structures when applied to the basis matrix within the simplex method.

2.2.3 Pivoting and permutation

Before discussing more sophisticated LU factorization techniques, it is necessary to introduce the pivoting and permutation which are fundamental elements of these advanced approaches.

When applying Gaussian elimination to obtain LU factors, if $b_{kk} = 0$ at stage k, then it can not be used to eliminate other rows because the multiplier defined as $e_{ik} = b_{ik}/b_{kk}$ (2.11) can not be computed. However, if there exists an entry b_{tk} from the active partition of column k, where $b_{tk} \neq 0$, then the regular elimination can be carried out by exchanging the row k and row t in advance. The work to find a suitable choice for elimination is called *pivoting*, and the operations to bring the choice to the pivotal position (position of b_{kk}) is called *permutation*. On the other hand, if all entries of the active partition of column are zeros, then the basis matrix is declared *singular* and the LU factorization is terminated.

More generally, pivoting is also applied for finding a better choice in terms of numerical accuracy and sparsity, where the searching may involve the whole active partition and permutation may also applied among basis columns. If the pivoting is only performed row-wise in the k^{th} column, then it is called *partial pivoting*; if the pivoting is applied to the whole active partition, then it is called *complete pivoting*. After LU factorization with *complete pivoting*, by incorporating all permutations in the original basis matrix B, the formation of LU factors can be expressed as

$$LU = PBQ, \tag{2.16}$$

where matrices P and Q represent row-wise and column-wise permutations respectively.

With complete pivoting, solving a system with the basis matrix requires permutation of the right-hand-side. Considering the fact that $P^{-1} = P^{\mathrm{T}}$ and $Q^{-1} = Q^{\mathrm{T}}$ as the feature of a permutation matrix, by rearranging the LU factorization equation (2.16), it easy to have

$$B = P^{\mathrm{T}}LUQ^{\mathrm{T}} \quad \text{and} \quad B^{-1} = QU^{-1}L^{-1}P. \tag{2.17}$$

Therefore, solving linear systems with the basis matrix requires two permutations, before and after applying regular LU factors.

In practice, the column-wise permutation is often combined with the basis matrix. Within the context of the simplex method, the basis matrix, as an ordered subset of the coefficient matrix, is always held as a vector called `base`, consisting of index of the corresponding coefficient matrix columns. Thus, applying the column-wise permutation to the basis matrix can be simply achieved by permuting the `base` vector with the column-wise matrix Q, so that the permuted basis matrix $\tilde{B}$ and its inverse can be expressed as

$$\tilde{B} = BQ = P^{\mathrm{T}}LU \quad \text{and} \quad \tilde{B}^{-1} = U^{-1}L^{-1}P. \tag{2.18}$$

Clearly, solving linear systems with the permuted basis matrix $\tilde{B}$ only requires one permutation on the right-hand-side by row-wise permutation matrix P.

The row-wise permutation P on the right-hand-side can also be avoided. By rearranging the LU factorization equation (2.16), it is easy to obtain another permuted basis $\bar{B}$ and its inverse as

$$\bar{B} = BQP = P^{\mathrm{T}}LUP \quad \text{and} \quad \bar{B}^{-1} = P^{\mathrm{T}}U^{-1}L^{-1}P, \tag{2.19}$$

where the row-wise permutation P is transferred into another column-wise permutation and thus combined with the basis matrix by permuting the `base` vector. This approach may appear to be awkward at first glance, as solving with $\bar{B}$ now again requires two permutations, before and after applying the LU factors. However, further decomposition of the LU factors reveals the underlying simplicity. The permuted LU factors $P^{\mathrm{T}}LUP$ can be expanded by inserting $PP^{\mathrm{T}} = I$ into the middle of eta matrices, yielding

$$\bar{B} = P^{\mathrm{T}}LUP = P^{\mathrm{T}}LPP^{\mathrm{T}}UP = \prod_{i=1}^{m}(P^{\mathrm{T}}L_iP)\prod_{i=m}^{1}(P^{\mathrm{T}}U_iP) = \prod_{i=1}^{m}\bar{L}_i\prod_{i=m}^{1}\bar{U}_i = \bar{L}\bar{U}, \tag{2.20}$$

where the permutation matrix P and its transpose are merged into the LU factors by symmetrically permuting each eta matrix. The permuted result $\bar{L}_i = P^{\mathrm{T}}L_iP$ and $\bar{U}_i = P^{\mathrm{T}}U_iP$, are also eta matrices. Therefore, solving with the permuted basis matrices $\bar{B}$ can be achieved by solving with the permuted factors $\bar{L}$ and $\bar{U}$, represented by a product of $2m$ permuted eta matrices, without any permutation of the right-hand-side.

Moreover, rather than applying the permutation after the LU factorization, the permuted eta matrices $\bar{L}_i$ and $\bar{U}_i$ are naturally formed directly during the LU factorization. It can be explained by examining the formation of the elimination matrix E_k, which differs from $L_k = E_k^{-1}$ merely in the sign of the eta vector, following the example given in (2.12). At stage $k = 3$, if row 5 was chosen and thus exchanged with row 3, assuming that is the only permutation involved during whole elimination, then the permutation matrix P, the elimination matrix E_3, and the permuted elimination matrix $\bar{E}_3 = P^{\mathrm{T}} E_3 P$ can be expressed as

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, E_3 = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & -e_{43} & 1 & & \\ & & -e_{33} & & 1 & \\ & & -e_{63} & & & 1 \end{bmatrix}, \text{ and } \bar{E}_3 = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & -e_{33} & \\ & & & 1 & -e_{43} & \\ & & & & 1 & \\ & & & & -e_{63} & 1 \end{bmatrix}.$$

Comparing $\bar{E}_3$ with E_3, it is easy to find that, while E_3 works on the permuted basis matrix, $\bar{E}_3$ works on the original one by directly eliminating other rows by row 5 without explicit permutation. Therefore, $\bar{E}_i$ (and thus $\bar{L}_i$) can be directly formed if Gaussian elimination is performed *without explicit permutation*. The same simplicity is true for the permuted eta matrix $\bar{U}_i$.

In the future text, for clarity, the true triangular L and U are refereed to as *LU factors* and the permuted result $\bar{L}$ and $\bar{U}$ are called *permuted LU factors*.

2.2.4 Numerical accuracy

Another major concern when implementing the LU factorization is the numerical accuracy. In modern computer software, the value of a non-zero entry is always represented by *double precision* number, which has approximately 16 significant decimals. The implication of this (16 significant decimals) representation is, as demonstrated in Table 2.1, that when add a large value (a) to a smaller value (for example, $x = 1 \times 10^{-7}$) and then subtract it, the result ($\bar{x}$) becomes a different value. When the difference of the magnitude is less than 13 ($a < 1 \times 10^{6}$) the difference between ($\bar{x}$) and (x) is small. If a has large magnitude, the difference between ($\bar{x}$) and (x) becomes significant. The disaster happens with $a = 1 \times 10^{10}$, where $\bar{x} = 0$.

a	$\bar{x} = (x + a) - a$
1×10^{-4}	$1.000000000000 \times 10^{-7}$
1×10^{-3}	$1.000000000001 \times 10^{-7}$
1×10^{-2}	$9.999999999941 \times 10^{-8}$
1×10^{-1}	$1.000000000029 \times 10^{-7}$
1×10^{0}	$1.000000000584 \times 10^{-7}$
1×10^{1}	$9.999999939225 \times 10^{-8}$
1×10^{2}	$9.999999406318 \times 10^{-8}$
1×10^{3}	$9.999996564147 \times 10^{-8}$
1×10^{4}	$1.000007614493 \times 10^{-7}$
1×10^{5}	$1.000007614493 \times 10^{-7}$
1×10^{6}	$1.000007614493 \times 10^{-7}$
1×10^{7}	$1.005828380585 \times 10^{-7}$
1×10^{8}	$1.043081283569 \times 10^{-7}$
1×10^{9}	$1.192092895508 \times 10^{-7}$
1×10^{10}	0

Table 2.1: Demonstration of the numerical accuracy issue, $x = 1 \times 10^{-7}$

To address this issue, the customary approach is to keep non-zero values in the system small. In Gaussian elimination, this is achieved by selecting an entry with larger magnitude (than $u \max_i |b_{ik}|$, where $0 < u \leq 1$) as the pivot to keep multipliers relatively small. This approach, called *threshold pivoting*, is also applied in other components of the revised simplex method implementation, for example, the Harris ratio test [33].

2.2.5 Singletons and triangularization

The simplex basis matrix is so sparse that it often contains a column or row which has only one non-zero entry, which is called a *singleton column* or *singleton row*. Following the previous example in equation (2.12), when $m = 6, k = 3$, the singleton column and row are illustrated as Figure 2.1 and 2.2 respectively.

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ & & \underline{b_{33}} & \underline{b_{34}} & \underline{b_{35}} & \underline{b_{36}} \\ & & & \underline{b_{44}} & \underline{b_{45}} & \underline{b_{46}} \\ & & & \underline{b_{54}} & \underline{b_{55}} & \underline{b_{56}} \\ & & & \underline{b_{64}} & \underline{b_{65}} & \underline{b_{66}} \end{bmatrix}$$

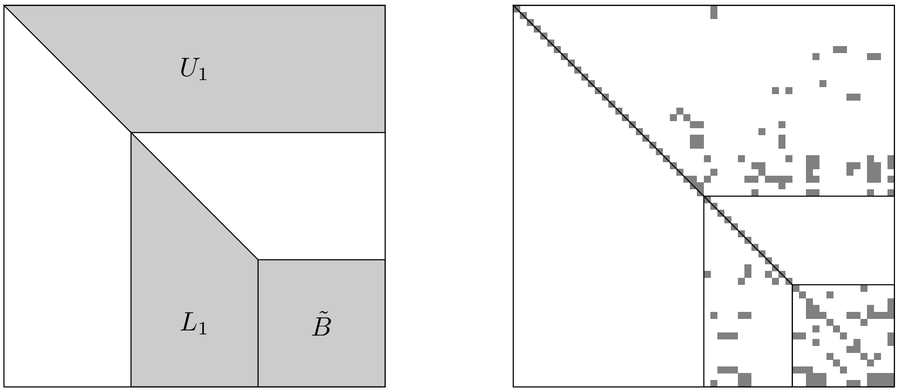
Figure 2.1: Singleton column

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ & & \underline{b_{33}} & & & \\ & & \underline{b_{43}} & \underline{b_{44}} & \underline{b_{45}} & \underline{b_{46}} \\ & & \underline{b_{53}} & \underline{b_{54}} & \underline{b_{55}} & \underline{b_{56}} \\ & & \underline{b_{63}} & \underline{b_{64}} & \underline{b_{65}} & \underline{b_{66}} \end{bmatrix}$$

Figure 2.2: Singleton row

It is obvious that if a singleton is chosen as pivot, then the corresponding Gaussian elimination is trivial and void because the remaining active partition will not be changed. Also, pivoting on a singleton and removing it from the active basis partition often creates other singletons. Therefore, the LU factorization is always started from a *triangularization* phase, where all singleton columns and singleton rows are identified and removed.

The result of triangularization is a permuted form of the basis matrix as shown in Figure 2.3. It consists of a partial upper triangular factor U_1 (from singleton columns), a partial lower triangular factor L_1 (from singleton rows) and a remaining active partition $\tilde{B}$. The $\tilde{B}$ is called *bump* in this research, and it is also known as *kernel* or *nucleus* in other literature. Efficient factorization of the bump $\tilde{B}$ requires more sophisticated Gaussian elimination techniques, which will be described in the rest of this section.



(a) Theoretical

(b) Real example (ADILTTLE)

Figure 2.3: Triangularization result

2.2.6 Bump factorization

After triangularization, further LU factorization on the bump $\tilde{B}$ becomes non-trivial. While the fundamental method has been well defined, the major difficulty is how to arrange the Gaussian elimination pivoting sequences to exploit sparsity of the basis matrix.

Markowitz merit

During LU factorization, a sparser pivotal column and row is preferred because pivoting on it involves less elimination work. Specially, for the basis matrix with a high percentage of zeros, a large proportion of the elimination operations insert non-zeros into the active basis partition, where a created non-zero by elimination is called a *fill-in*.

To measure the amount of elimination work corresponding to each potential pivot choice, Markowitz proposed the selection criterion [44] defined as

$$m_{ij} = (r_i - 1)(c_j - 1), \tag{2.21}$$

where *row count* r_i and *column count* c_j are counts of non-zero entries of row i and column j of active basis partition respectively. This merit is often referred as the *Markowitz merit* by later researchers. The number m_{ij} is the upper limit of fill-ins when entry b_{ij} is used as the pivot. For the bump in Figure 2.3(b), initial Markowitz merit values for each of the non-zero elements are shown in Table 2.2.

		c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
		3	3	6	6	3	3	2	2	3	4	3	5	4	3	7
r_1	2	2										2				
r_2	2		2				2									
r_3	2			5												6
r_4	4			15	15								12			18
r_5	9			40	40	16				16	24		32	24	16	48
r_6	2						2					2				
r_7	2				5			1								
r_8	4	6	6						3							18
r_9	3				10						6					12
r_{10}	2									2	3					
r_{11}	3						4					4		6		
r_{12}	2							1					4			
r_{13}	4	6	6	15					3							
r_{14}	7			30	30	12							24	18	12	36
r_{15}	9			40	40	16				16	24		32	24	16	48
t_j		7	7	29	28	22	4	2	6	17	19	4	26	24	22	31

Table 2.2: Initial Markowitz merit (m_{ij}) and Tomlin column ordering merit (t_j)

Practically, maintaining all the merit data and finding the best m_{ij} before each elimination step can be very expensive. Thus, based on the Markowitz merit, many researchers proposed approximation and more sophisticated implementation of it.

Tomlin's approach

Tomlin's approach [57] is an approximation to the Markowitz selection criterion. It starts by ordering all the bump columns before elimination. Tomlin's column ordering is based on a *merit count* defined as

$$t_j = \sum_{\{i|\tilde{b}_{ij} \neq 0\}} (r_i - 1).$$

The values of the merit count are also shown in Table 2.2. It can be observed form the table that the best pivoting positions ($b_{7,7}$ and $b_{12,7}$) according to Markowitz merit are also preferable with Tomlin's column ordering approach (column 7). The column ordering is applied as column permutations, so that all active basis columns are ordered by increasing Tomlin merit count.

Besides the column-wise ordering and permutation, Tomlin's approach also provides a row-wise selection criterion. It is a combined criterion which concerns both sparsity and numerical accuracy. On the sparsity aspect, a row p with smallest initial row count r_p will be chosen. On the numerical accuracy aspect, a row s with relatively large entry will be chosen by threshold pivoting. Putting both consideration together gives the row-wise selection criterion

$$r_p = \min\{r_s | \, |\tilde{b}_{sk}| \geq u \max |\tilde{b}_{ik}|\}, \tag{2.22}$$

where $u \in (0, 1]$ is the relative ratio comparing to the largest entry in column k, which is normally set to 0.01 or 0.1 as recommend by Tomlin.

Advanced searching techniques

Although Tomlin's approach provides a practical approximation to the Markowitz merit in terms of efficiency, it cannot handle the sparsity pattern during elimination, which may eventually damage its efficiency. To provide a better sparsity oriented approach which also contributes to inversion speed, Duff [13] and then Suhl and Suhl [55] provided more sophisticated implementation techniques of the Markowitz's approach.

The core of Suhl and Suhl's implementation is an improved searching approach which starts from a smaller row count r_i and column count c_j. Intuitively, the smallest merit value $m_{ij} = (r_i - 1)(c_j - 1)$ comes from a smaller r_i or c_j. Therefore, the searching can be arranged by ascending count r_i or c_j . With this searching arrangement, the smaller m_{ij} merit values are more likely to appear at the beginning, while the larger m_{ij} merit values are more likely to appear at the end. Thus the best of first few choices (for example, the first 4 as suggested by Suhl and Suhl), if is not, may be quite close to the globally best choice. To assist the advanced searching approach, Suhl and Suhl further adopted an important data structure, which can be called the *count indexed doubly linked list* to help maintaining an count-ordered list of active partition columns and rows. Besides the searching techniques, the threshold pivoting approach is also adopted.

Currently, Suhl and Suhl's approach is widely accepted as the best and default choice when implementing the revised simplex method, both in commercial software or in the public domain projects. It is implemented during this research as a solid base of the efficient revised simplex solver.

2.3 Solving with LU factors

This section introduces the basic and the hyper-sparse linear system solution techniques. The major consideration and challenge when implementing the FTRAN and BTRAN operations are how to properly exploit the inherent sparsity and the hyper-sparsity. Further complexity arises when permuted LU factors are used.

Exploiting sparsity has been a standard operation since the introduction of the revised simplex method. When solving with LU factors, exploiting sparsity is generally achieved by using packed storage (of non-zero entries) of eta matrices. Exploiting hyper-sparsity is a relative novel development. Efficient implementation of the hyper-sparse FTRAN and BTRAN requires more sophisticated data structures.

Therefore, although linear system solution techniques can be described in mathematical notations, it is clearer and move convenient to discuss them by pseudo code. This section firstly introduces the storage scheme of permuted LU factors and then, base on which, details the basic and the hyper-sparse solving techniques.

2.3.1 Storage of LU factors

The resulting eta vector of the LU factorization is generally sparse (for example, as the result of the singletons or small Markowitz merit), thus it is natural to store it in sparse format. Customarily, the m eta matrices of a factor, are stored in a compact packed format using several arrays as listed in Table 2.3.

Name	Usage
`Ustart` `Uend`	The pair `Ustart[i]` and `Uend[i]` store the start and position after the end in `Uindex` and `Uvalue` for the packed storage of $\bar{U}_i$.
`Uindex` `Uvalue`	For each `Ustart[i]` $\leq$ `k` $<$ `Uend[i]`, the pair `Uindex[k]` and `Uvalue[k]` store the index and value of a non-zero entry of the eta vector associated with $\bar{U}_i$.
`Upiv_i` `Upiv_x`	The pair `Upiv_i[i]` and `Upiv_x[i]` store the index and value of the pivotal entry of eta matrix $\bar{U}_i$.
`Ulookup`	The entry `i = Ulookup[p]` is the original position in $\boldsymbol{x}$ of pivot p.

Table 2.3: Storage scheme for permuted upper factor $\bar{U}$

The storage scheme for the permuted lower factor $\bar{L}$ is similar. The only difference is that the array `Lpiv_i` is not required because the pivotal entries of the eta matrix in the lower factor are “1”.

2.3.2 Basic solving techniques

This subsection introduces basic FTRAN and BTRAN operations in pseudo code. This uses the grammar of the C\C++ programming language with the natural numbering of FORTRAN.

Because permuted LU factors are represented as a product of a series of eta matrices, the FTRAN and BTRAN operations can be achieved by repeating the elementary ETA-FTRAN (2.9) and ETA-BTRAN (2.10) operations respectively.

By using the packed storage of permuted LU factors introduced in Section 2.3.1, the FTRAN operation can be achieved by the pseudo code shown in Figure 2.4.

When the vector $\boldsymbol{x}$ is sparse the ETA-FTRAN operation (2.9) is easily skipped by zero testing at lines 4 and 12 in Figure 2.4. However, for ETA-BTRAN (2.10), the corresponding null operation occurs when the inner product of $\boldsymbol{x}$ and $\boldsymbol{\eta}$ in ETA-BTRAN operation (2.10) is zero. This is likely to occur but cannot be identified so easily. However, it is possible to represent the LU factors

```
// 1. Solve with the lower factor
for (int i = 1; i <= m; i++) {
    double pivot = x[ Lpiv_i[i] ];
    if (pivot != 0)
        for (int k = Lstart[i]; k < Lend[i]; k++)
            x[ Lindex[k] ] += pivot * Lvalue[k];
}

// 2. Solve with the upper factor
for (int i = m; i >= 1; i--) {
    double pivot = x[ Upiv_i[i] ];
    if (pivot != 0) {
        pivot = pivot / Upiv_x[i];
        x[ Upiv_i[i] ] = pivot;
        for (int k = Ustart[i]; k < Uend[i]; k++)
            x[ Uindex[k] ] += pivot * Uvalue[k];
    }
}
```

Figure 2.4: Standard FTRAN with permuted LU factors

as a product of row eta matrices, each of which is of the form

$$R = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ r_1 & \dots & r_p & \dots & r_m \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}. \tag{2.23}$$

Thus the transposed system (2.2) may be solved via a sequence of ETA-FTRAN operations (2.9) using the row-wise representation of the LU factors.

Experience during the implementation shows that the procedure for forming the row-wise representation for a given permuted factor, although it consists of only about 20 lines of code, is particularly hard to write correctly. This is because of the underlying complexity associated with the permuted LU factors.

For a true triangular factor, forming the row-wise representation is straightforward. For the factor U, it can be achieved by taking each diagonal entry u_{ii} and the remaining entries of its row i as the pivot and row eta vector for the row eta matrix U^i, for $i = 1, \dots, m$ top-down. For the factor L, the corresponding row-wise decomposition is achieved bottom-up, yielding L^i, for $i = m, \dots, 1$. Therefore, by using a row-wise representation, the basis matrix is expressed as

$$B = LU = \prod_{i=m}^{1} L^i \prod_{i=1}^{m} U^i, \tag{2.24}$$

where L^i and U^i correspond to row i of the lower factor and the upper factor respectively.

For a permuted LU factor, forming the row-wise representation is achieved (theoretically) in three steps: (1) permuting the permuted factor back to true triangular forms; (2) decomposing the true triangular factors row-wise; and (3) permuting the decomposed row eta matrix to the permuted form:

$$\bar{U} = P(U)P^{\mathrm{T}} = P\left(\prod_{i=1}^{m} U^i\right)P^{\mathrm{T}} = \prod_{i=1}^{m}\left(PU^iP^{\mathrm{T}}\right) = \prod_{i=1}^{m}\bar{U}^i,$$

where $\bar{U}^i$ is a permuted row eta matrix. It can be observed from the formation, that eta matrix $\bar{U}_i$ and row eta matrix $\bar{U}^i$ share the same pivot, the diagonal entry u_{ii}.

In terms of storage, the row-wise representation is represented as two additional sets of arrays, with prefix `LR` and `UR` for row-wise $\bar{L}$ and $\bar{U}$ respectively. The pivotal entries, `URpiv_i`, `URpiv_x` and `URlookup` are identical to that of for the permuted upper factor because the pivotal entry of $\bar{U}_i$ and $\bar{U}^i$ are the same. This is also true for the permuted lower factor.

The pseudo code in Figure 2.5 demonstrates how to form the row-wise representation of the permuted upper factor. The additional vector `URcount` (initially all zero) counts the non-zero entries in each row-wise eta matrix. The two permutations, "permuting back" (to the true triangular) and "permuting to" (the permuted triangular) are achieved by using vectors `Ulookup` and `Upiv_i` respectively.

```
// 1. Counting non-zero entries for each UR eta matrix j
for (int i = 1; i <= m; i++) {
    for (int k = Ustart[i]; k < Uend[i]; k++) {
        int iRow = Ulookup[ Uindex[k] ]; // index in the triangular factor
        URcount[iRow]++;
    }
}

// 2. Constructing the URstart pointer by accumulation
URstart[1] = 1;
for (int i = 2; i <= m; i++)
    URstart[i] = URstart[i - 1] + URcount[i - 1];

// 3. Filling UR element, URend becomes ready afterwards
URend = URstart;
for (int i = 1; i <= m; i++) {
    for (int k = Ustart[i]; k < Uend[i]; k++) {
        int iRow = Ulookup[ Uindex[k] ];
        int iPut = URend[iRow]++;
        URindex[iPut] = Upiv_i[i];       // index in the permuted factor
        URvalue[iPut] = Uvalue[k];
    }
}
```

Figure 2.5: Form row-wise representation for a permuted factor.

2.3.3 Hyper-sparse solving techniques

When solving linear systems with the basis matrix, it has been observed, for example by Hall and McKinnon [32], that for certain families of LP problems, the solution itself is often extremely sparse, so that the zero-testing operations in the FTRAN operation are dominant. To avoid excessive zero-testing operations, FTRAN operations can be achieved by explicit exploiting the sparsity pattern of the factor to skip zero entries of the RHS. The BTRAN operations with row-wise representation is essentially the same. The advanced FTRAN operation is called the *hyper-sparse* FTRAN.

There are two major implementation approaches for achieving the hyper-sparse FTRAN.

The one briefly mentioned by Gilbert and Peierls [20] consists of two stages. It starts by constructing a list of all required eta matrices via depth first search (DFS), and then performs the FTRAN operation with the eta matrices on the list only (in reverse order). Note that by using the two-stage hyper-sparse FTRAN, the overall FTRAN operation is also split into two parts, firstly search-and-solve with the lower factor, and then with the upper factor.

The approach detailed by Hall and McKinnon [32] maintains a list of required eta matrices and performs the hyper-sparse FTRAN operations by searching for the "next" eta matrix in the list. When solving with the lower (upper) factor, the next eta matrix is the one associated with smallest (largest) index. By merging the two sets of eta matrices $L_1, L_2 \ldots, L_m$ and $U_m, \ldots, U_2, U_1$, and re-labeling them by $E_1, E_2, \ldots, E_{2m}$, it is possible to merge these two situations, as described in the original paper.

Because the detail of the the DFS based hyper-sparse FTRAN is missing from the original paper [20], in particular how to solve with permuted factors, it is included in this report for completeness. Pseudo code for the search and solve stages of the two-stage hyper-sparse FTRAN are provided in Figures 2.6 and 2.7 respectively.

```
int listCount = 0;    // Number of FTRAN to-do
int stackSize = 0;    // Usage of the stack (0 means empty)
for (int t = 1; t <= Xcount; t++) {

    int i = Hlookup[ Xindex[t] ]; // ith eta matrix of H
    int k = Hstart[i];            // the next non-zero position to visit

    if (visited[i] == 0) {
        visited[i] = 1;

        for (;;) {
            // Keep searching current ETA until finish
            if (k < Hend[i]) {

                // Move to a child if it is not yet been visited
                int child = Hlookup[ Hindex[k++] ];
                if (visited[child] == 0) {
                    visited[child] = 1;

                    // Store current eta (the father) to stack
                    stack[++stackSize] = i;
                    stack[++stackSize] = k;

                    // Start to search the child
                    i = child;
                    k = Hstart[child];
                }
            } else {
                // Put current eta to the FTRAN to-do list
                list[++listCount] = i;

                // Get another eta (the father) from the stack or quit
                if (stackSize == 0)
                    break;
                k = stack[stackSize--];
                i = stack[stackSize--];
            }
        }
    }
}
```

Figure 2.6: DFS based hyper-sparse FTRAN: search stage

The DFS search stage identifies a topological ordering of the required eta matrices when solving with one set of eta matrices, where the eta matrices identified in the “deepest” search are performed last. The DFS search is given in Figure 2.6. The search procedure is identical for all four sets of eta matrices (`L, U, LR, UR`), so the prefix `H` is used to represent on of the four set of eta matrices. Because exploiting hyper-sparsity involves the non-zero index of the RHS vector $\boldsymbol{x}$, it is held in an indexed array format by two vectors `Xarray, Xindex` and one scalar `Xcount`. `Xcount` is the number of non-zero entries in $\boldsymbol{x}$ (before and after the FTRAN operation), `Xindex` contains the (unordered) index of the non-zero entries of `Xarray`. After the search a to-do `list` is formed so that applying the eta matrices in it in reverse order achieves the second stage of the hyper-sparse FTRAN. The number `listCount` indicates the length of the list. In additional, the vector `visited` (initially all zero) is used to mark whether an eta matrix has been visited, and the `stack` is used to store the search status with the current eta matrix before visiting a child.

```
 1  Xcount = 0;
 2  for (int t = listCount; t >= 1; t--) {
 3      int i = list[t];  // ith eta matrix of H
 4      visited[i] = 0;
 5
 6      int    ipivot = Hpiv_i[i];
 7      double xpivot = Xarray[ipivot];
 8      if (xpivot != 0) {
 9          xpivot /= Hpiv_x[i];
10          Xarray[ipivot]   = xpivot;
11          Xindex[Xcount++] = ipivot;
12
13          for (int k = Hstart[i]; k < Hend[i]; k++)
14              Xarray[ Hindex[k] ] -= pivot * Hvalue[k];
15      }
16  }
```

Figure 2.7: DFS based hyper-sparse FTRAN: solve stage

After the identification of the to-do `list`, performing hyper-sparse FTRAN is a straightforward task. It is slightly different from the standard FTRAN operations in that the RHS vector is maintained in the indexed array form. The hyper-sparse FTRAN procedure is identical for all four set of eta matrices (by explicitly using `Lpiv_x[i]=1`). Using `H` to represent one of the four sets of eta matrices, the second stage of hyper-sparse FTRAN operation can be achieved by the pseudo code shown in Figure 2.7. During the hyper-sparse FTRAN operations, the vector `visited` is restored to zero (line 4). A zero-test is still performed (line 8) in case of cancellation.

2.4 Updating LU factors

This section introduces existing simplex update methods. The simplex update refers to the update of the basis matrix and its inverse at the end of every simplex iteration. While the basis matrix update formula (2.6) is well defined, its inverse representation update is still an active research area.

2.4.1 Product form update

The product form (PF) update rearranges (2.6) so that

$$\bar{B} = B + (\boldsymbol{a}_q - B\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}} = B\big(I + (\hat{\boldsymbol{a}}_q - \boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}}\big) = BE$$

where $E = I + (\hat{\boldsymbol{a}}_q - \boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}}$ is an eta matrix whose vector $\hat{\boldsymbol{a}}_q$ is naturally available as the result of the forward system (2.3). Applying this k times yields the following representation of the basis matrix and its inverse.

$$B_k = B_0 E_1 E_2 \dots E_k \quad \Rightarrow \quad B_k^{-1} = E_k^{-1} \dots E_2^{-1} E_1^{-1} B_0^{-1} \tag{2.25}$$

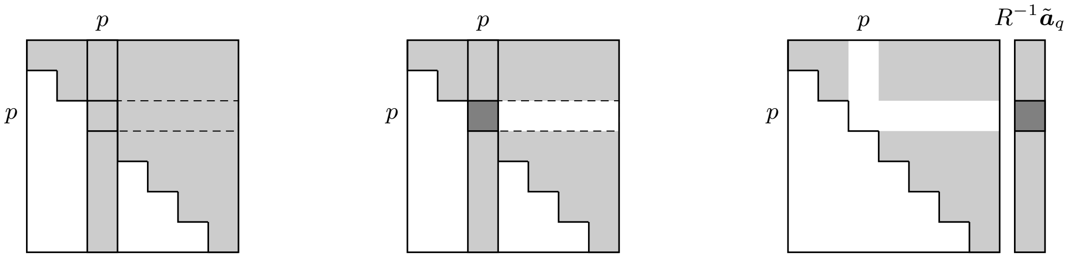
The original decomposition $B_0 = L_0 U_0$ remains unaltered when the basis matrix changes.

2.4.2 Forrest-Tomlin update

By allowing the decomposition $B = LU$ to be modified, the Forrest-Tomlin (FT) update [18] generally achieves greater efficiency with respect to sparsity than the PF update. This is done by working on the following rearrangement of the basis matrix update equation.

$$\begin{aligned} \bar{B} &= B + (\boldsymbol{a}_q - B\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}} \\ \Rightarrow \quad L^{-1}\bar{B} &= U + (L^{-1}\boldsymbol{a}_q - U\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}} \\ &= U + (\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)\boldsymbol{e}_p^{\mathrm{T}} = U' \end{aligned} \tag{2.26}$$

Whilst the basis update equation (2.6) replaces column p of basis matrix B by $\boldsymbol{a}_q$ in (2.26), column p of the factor U is replaced by the partial FTRAN result $\tilde{\boldsymbol{a}}_q = L^{-1}\boldsymbol{a}_q$. As illustrated in Figure 2.8(a), the replacement yields a spiked upper factor U'.



(a) Spiked upper: U' (b) After elimination: $\bar{U}$ (c) Actual implementation

Figure 2.8: Forrest-Tomlin update: column spike and elimination

From U', the Forrest-Tomlin update restores triangularity by elimination. Specifically, it uses other rows to eliminate the off-diagonal entries of row p, yielding a permuted triangular matrix $\bar{U}$ as shown in Figure 2.8(b).

The elimination process can be represented by a single row transformation R^{-1}, so that $\bar{U} = R^{-1}U'$. Note that this row eta matrix R is a special case of (2.23) since it has pivotal entry $r_p = 1$ so R and its inverse can be expressed as $R = I + \boldsymbol{e}_p\boldsymbol{r}^{\mathrm{T}}$ and $R^{-1} = I - \boldsymbol{e}_p\boldsymbol{r}^{\mathrm{T}}$ respectively. The computation of the eta vector $\boldsymbol{r}$ was identified by Forrest and Tomlin [18] as an additional partial BTRAN operation $\boldsymbol{r}^{\mathrm{T}} = \bar{\boldsymbol{u}}_p^{\mathrm{T}} U^{-1}$, where $\bar{\boldsymbol{u}}_p^{\mathrm{T}}$ is row p of U without the diagonal entry u_{pp}. It can easily be verified that applying R^{-1} to U as $R^{-1}U = U - \boldsymbol{e}_p\bar{\boldsymbol{u}}_p^{\mathrm{T}}$ modifies only the entries in row p, eliminating all its off-diagonal entries. Since U and U' differ only in their p^{th} column, applying R^{-1} to U' also eliminates its off-diagonal entries in row p. Meanwhile, applying R^{-1} to column p of U' modifies only the p^{th} entry of the newly inserted vector $\tilde{\boldsymbol{a}}_q$ which becomes $\tilde{a}_{pq} := \tilde{a}_{pq} - \boldsymbol{r}^{\mathrm{T}}\tilde{\boldsymbol{a}}_q$. Therefore, applying the row transformation R^{-1} to U' yields the permuted triangular matrix $\bar{U}$ as shown in Figure 2.8(b).

As identified later by Tomlin [58], the additional partial BTRAN operation can be avoided by forming $\boldsymbol{r}$ from the intermediate result $\tilde{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}}U^{-1}$ of the regularly solved transposed system (2.4). This can be identified by observing that $\bar{\boldsymbol{u}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}}U - u_{pp}\boldsymbol{e}_p^{\mathrm{T}}$ so

$$\boldsymbol{r}^{\mathrm{T}} = \bar{\boldsymbol{u}}_p^{\mathrm{T}}U^{-1} = (\boldsymbol{e}_p^{\mathrm{T}}U - u_{pp}\boldsymbol{e}_p^{\mathrm{T}})U^{-1} = \boldsymbol{e}_p^{\mathrm{T}} - u_{pp}\tilde{\boldsymbol{e}}_p^{\mathrm{T}}. \tag{2.27}$$

Since its p^{th} entry is zero, it follows that $\boldsymbol{r}$ is given by scaling $\tilde{\boldsymbol{e}}_p$ by $-u_{pp}$ and setting the p^{th} entry to zero. Thus, if the partial BTRAN result $\tilde{\boldsymbol{e}}_p$ is stored, it can be assumed that the eta vector $\boldsymbol{r}$ is available at negligible cost.

Clearly $\bar{U} = R^{-1}U'$ is not a triangular matrix, but it can be put into this form via a symmetric cyclic permutation of rows and columns p to m. However, in practice, no permutations are performed since all that is required is an invertible representation of $\bar{U}$ rather than an explicit triangular matrix. Within an implementation, triangular matrices in basis matrix decompositions are represented in product form as sequences of eta vectors, pivotal entries and pivotal indices. Thus the representation of $\bar{U}$ is obtained by deleting the eta vector corresponding to column p of U, setting all entries corresponding to row p of U to zero and appending a new eta vector $R^{-1}\tilde{\boldsymbol{a}}_q$, pivotal entry $\tilde{a}_{pq}$ and index p to the sequence, as illustrated in Figure 2.8(c).

Combining $\bar{B} = LU'$ (2.26) and $\bar{U} = R^{-1}U'$ yields the following updated representation of the basis matrix and its inverse.

$$\bar{B} = LR\bar{U} \quad \text{and} \quad \bar{B}^{-1} = \bar{U}R^{-1}L^{-1}$$

Repeating these operations, after k updates, the basis matrix B_k and its inverse can be expressed as

$$B_k = LR_1R_2 \dots R_kU_k \quad \Rightarrow \quad B_k^{-1} = U_k^{-1}R_k^{-1} \dots R_2^{-1}R_1^{-1}L^{-1}. \tag{2.28}$$

The essential difference between the FT and PF updates is that the former stores two partially transformed results $\tilde{\boldsymbol{a}}_q$ and $\tilde{\boldsymbol{e}}_p$ and deletes one row and one column from the eta file, whereas the PF update simply stores the final FTRAN result $\hat{\boldsymbol{a}}_q$. These deletions and the sparsity of the partially transformed results relative to $\hat{\boldsymbol{a}}_q$ is such that the FT update frequently has a significantly lower storage requirement than the PF update. This, in turn, leads to generally superior performance when the invertible representation based on the FT update is used to solve linear systems. However, the operations of the FT update require dynamic data structures to accommodate deletion and insertion, making it significantly more difficult to implement than the PF update.

2.4.3 FT update implementation concerning hyper-sparsity

After a Forrest-Tomlin update, the upper factor is altered. Therefore, to maintain the ability to exploit sparsity and hyper-sparsity, it is necessary to update the row-wise representation of the upper factor and the hyper-sparsity data structures respectively.

As demonstrated by the pseudo codes in Figure 2.9 and 2.10, although the FT update when maintaining hyper-sparsity related data structures can be achieved in about one hundred lines of code, writing such code correctly and elegantly is particularly hard. This is due to the inherent difficulty of implicit permutation and additional complexity associated with maintaining the row-wise representation, and partially because the detailed implementation technique has never been documented before. Therefore, this report fills the gap. The procedures listed in Figure 2.9 include all major operations of the FT update except the updating of the row-wise representation, which is provided as a standalone code in Figure 2.10. Details of the update operations are discussed in the remainder of this subsection with reference to the pseudo code.

Steps 1 and 2: checking and looking up. The first two steps in part 1 of the FT update are simple steps. The first step of part 1 checks the space for filling the partial FTRAN (in packed form represented by `aq`) and BTRAN (in packed form represented by `ep`) results. The second step obtains the reference p to the pivotal eta matrix $\bar{U}_p$ and row eta matrix $\bar{U}^p$ for a given pivoting index.

The subsequent steps of part 1 deal with the modification to the upper factor and R. The modification to the upper factor is essentially achieved by three steps: (1) deleting the pivotal eta matrix $\bar{U}_p$, (2) zeroing the pivotal position for eta matrices $\bar{U}_i, i > p$, and (3) appending a new eta matrix given by $R^{-1}\tilde{\boldsymbol{a}}_q$ to the end of the permuted upper factor as $\bar{U}_{m+1}$, yielding a new set of m eta matrices

$$\bar{U} = \prod_{i=m+1}^{p+1} \bar{U}_i \prod_{i=p-1}^{1} \bar{U}_i.$$

Step 3: Deleting the pivotal column. The operations to delete $\bar{U}_p$ correspond to the step 3 of the part 1 FT update procedure in Figure 2.9. Deleting $\bar{U}_p$ from the column-wise representation is simply achieved by "marking" rather than actually removing it, for example by setting `Upiv_i[p] = 0`. For this reason, it is necessary to add an additional test in the FTRAN and hyper-sparse FTRAN operations to identify and skip the disabled eta matrix. Deleting $\bar{U}_p$ from the row-wise representation involves searching and moving: (1) firstly searching for the corresponding position of each non-zero entry of $\bar{U}_p$ in the associated row-wise eta matrix, and then (2) moving the last non-zero entry of the row-wise eta matrix to replace the deleted entry, resulting in shorter packed row eta vector. The search is assisted by visiting via the row index `iRow = Uindex[k]` of each non-zero entry in the eta vector of $\bar{U}_p$, where the associated row eta matrix index is given by `Ulookup[iRow]`.

Step 4: Zeroing out the pivotal row. In the original paper, the operation to zero out the pivot row entries is described as a series of search-and-move operations on all of the column eta matrix $\bar{U}_i, i = p+1, \ldots, m$, because of the lack of a row-wise representation. This can be an expensive operation when the LP problem is hyper-sparse (the cost is obviously greater than performing m zero-testing). However, with aid of the row-wise representation, it can be achieved by considering the column index `URindex[k]` of each non-zero entry in the row eta vector of $\bar{U}^p$. The operation is essentially the same as that for deleting $\bar{U}_p$ from the row-wise eta matrices. This is achieved in step 4 of the part 1 FT update procedure. Note that marking $\bar{U}^p$ as disabled is already done by setting `Upiv_i[p] = 0`, as $\bar{U}^p$ and $\bar{U}_p$ share the same pivotal information. It is worth observing that the row-wise representation leads to a reduction in the cost of searches when updating the column-wise representation.

Step 5: Appending $R\tilde{\boldsymbol{a}}_q$. Appending the new eta transformation to the upper factor is achieved in step 5 of the part 1 FT update. It consists of recording the pivotal entry and

```
// 1. Initial check of additional storage spaces
if (UcountX + aq_npack > UlimitX || RcountX + ep_npack > RlimitX)
    return NO_SPACE;

// 2. Obtain pointer to the pivotal column (row) of U (UR)
int p = Ulookup[ipivot];

// 3. Delete the pivotal column of U from U and UR
Upiv_i[p] = 0; // Marking as disabled for U
for (int k = Ustart[p]; k < Uend[p]; k++) {
    int i = Ulookup[Uindex[k]];
    for (int iFind = URstart[i]; iFind < URend[i]; iFind++) {
        if (URindex[iFind] == ipivot) {
            URindex[iFind] = URindex[URend[i] - 1];
            URvalue[iFind] = URvalue[URend[i] - 1];
            URspace[i]++;
            URend  [i]--;
            break;
        }
    }
}

// 4. Delete the pivotal row of UR from U
for (int k = URstart[p]; k < URend[p]; k++) {
    int i = Ulookup[URindex[k]];
    for (int iFind = Ustart[i]; iFind < Uend[i]; iFind++) {
        if (Uindex[iFind] == ipivot) {
            Uindex[iFind] = Uindex[Uend[i] - 1];
            Uvalue[iFind] = Uvalue[Uend[i] - 1];
            Uend[i]--;
            break;
        }
    }
}

// 5. Append partial FTRAN result to U
Upiv_i[m + t] = ipivot;
Upiv_x[m + t] = Upiv_x[p] * a_pq;
Ustart[m + t] = UcountX;
for (int k = 0; k < aq_npack; k++)
    if (aq_ipack[k] != ipivot) {
        Uindex[UcountX]   = aq_ipack[k];
        Uvalue[UcountX++] = aq_xpack[k];
    }
Uend[m + t]   = UcountX;

// 6. Append partial BTRAN result to R
Rpiv_i[t] = ipivot;
Rstart[t] = RcountX;
for (int k = 1; k <= ep_npack; k++) {
    if (ep_ipack[k] != ipivot) {
        Rindex[RcountX]   = ep_ipack[k];
        Rvalue[RcountX++] = ep_xpack[k] * (-Upiv_x[p]); // See equation (2.26)
    }
}
Rend[t]   = RcountX;

// 7. Update the lookup table and pivotal pointers for UR
Ulookup[ipivot]  = m + t;
URstart[ m + t ] = URstart[p];
URend   [ m + t ] = URstart[p];
URspace[ m + t ] = URspace[p] + URend[p] - URstart[p];
```

Figure 2.9: FT update implementation Part 1: Deleting pivotal eta vectors and appending partial FTRAN and BTRAN results to U and R respectively.

copying the other entries of the partial FTRAN result $\tilde{\boldsymbol{a}}_q$ as the new eta vector. The pivotal entry, as indicated in the original paper [18], is available by either computing $R\tilde{\boldsymbol{a}}_q$ or directly using $\bar{u}_{pp}\hat{a}_{pq}$, where $\hat{a}_{pq}$ is the p^{th} entry of the final FTRAN result $\hat{\boldsymbol{a}}_q$. When copying the partial FTRAN result $\tilde{\boldsymbol{a}}_q$ (actually need to copy $R\tilde{\boldsymbol{a}}_q$, but $R\tilde{\boldsymbol{a}}_q$ and $\tilde{\boldsymbol{a}}_q$ are identical except for the pivotal entry), the pivotal entry is skipped. Updating the row-wise representation accordingly is a more complicated operation and thus is described in part 2 (Figure 2.10) of the FT update standalone.

Step 6: Updating R. As has been identified by Tomlin [58], forming the row eta vector for the elimination matrix R can be achieved effortlessly by appending a multiple $(-\bar{u}_{pp})$ of the partial BTRAN result $\tilde{\boldsymbol{e}}_p$. This is achieved in step 6 of the part 1 FT update. It is similar to appending partial FTRAN results as discussed with reference to step 5. The difference is that no further operations are required. The elimination matrix R is only held in row-wise form.

Step 7: Storing the pivotal entry. Step 7 of part 1 concludes the major FT update operations by updating the "lookup" table to refer to the pivotal index of the $(m+t)^{\text{th}}$ column eta matrix $\bar{U}_{m+t}$. To keep using the same lookup table for row-wise representation, a row-wise eta matrix $\bar{U}^{m+t}$ is appended to the row-wise representation. The pivotal entry of $\bar{U}^{m+t}$ is shared with that of $\bar{U}_{m+t}$. The row eta vector is initially empty (`URstart[m+t] = URend[m+t]`). In terms of storage of the eta vector, it reuse the space of $\bar{U}^p$.

```
for (int k = Ustart[ m + t ]; k < Uend[ m + t ]; k++) {
    // 1. Identify the row eta to insert
    int i = Ulookup[Uindex[k]];

    // 2. Make space if necessary
    if (URspace[i] == 0) {
        // 2.1 Determine new spaces and check memory
        int count = URend[i] - URstart[i];
        int space = count * 0.1 + 5;
        if (URcountX + count + space > URlimitX)
            return NO_SPACE;
        // 2.2 Copy the packed storage to the new position
        for (int j = 0; j < count; i++) {
             URindex[ URcountX + j ] = URindex[ URstart[i] + j ];
             URvalue[ URcountX + j ] = URvalue[ URstart[i] + j ];
        }
        // 2.3 Update pointers
        URstart[i] = URcountX;
        URend  [i] = URcountX + count;
        URspace[i] = space;
        URcountX   = URcountX + count + space;
    }

    // 3. Put into the next available space
    URindex[URend[i]] = ipivot;
    URvalue[URend[i]] = Uvalue[k];
    URspace[i]--;
    URend  [i]++;
}
```

Figure 2.10: FT update implementation Part 2: Update the row-wise representation.

Part 2 of the FT update procedure updates the row-wise representation. Mathematically, it is a simple task. Implementationally, it requires certain memory management to deal with the filling operation. As shown in Figure 2.10, it is achieved by visiting each non-zero entry of the new column eta vector, and inserting it into the corresponding row eta vector.

2.4.4 Reinversion

After many update operations, solving with the updated basis inverse will inevitably become slower and thus rebuilding the basis inverse representation is advantageous. However, the LU factorization is a relatively time-consuming operation compared to the iteration time of the simplex method. Therefore a balance is required.

Ideally, a reinversion is performed when the accumulated time spent on the update part (t_U^+) since last basis inversion, is equals to the time spent on the last basis inversion (t_I) $t_U^+ = t_I$.

This threshold is deduced by two assumptions: that the average time required by applying each update *eta* matrix a constant value (t_U), the average time of each INVERT is also a constant value, and the time spent on the other operations is not affected by the reinversion interval. Therefore, the task of determining the best reinversion is simplified to minimizing the overall time spent on INVERT and the update part of *eta* transformations. Assuming that solving an LP takes n iterations, and the reinversion interval is k, then the accumulated time (since the last basis reinversion) spent on the update part (t_U^+) can be expressed as $t_U^+ = \sum_{i=0}^{k} i \times t_U \approx \frac{k^2}{2} t_U$. Therefore to minimize the overall time of the INVERT and the updated part $t = \frac{n}{k} \times (\frac{k^2}{2} t_U + t_I)$ it requires that $t' = \frac{n}{2} t_U - \frac{n t_I}{k^2} = 0$, resulting in the optimal threshold

$$t_I = \frac{k^2}{2} t_U \approx t_U^+.$$

In the implementation, t_I is represented by counting the operations involved the inverse stage, and the accumulated t_U^+ is obtained by counting the accumulated operations on the FT (or PF) update part since the last reinversion. Of course, determining the optimal reinversion interval is not possible. However, computational experience shows that overall solution time is not sensitive to the reversion interval so a coarse model is sufficient.

2.5 Novel update techniques

This section introduces two novel variants of the product form update and an extension of the Forrest-Tomlin update. Although applicable in general, all are motivated by the requirements of the high performance implementation introduced in Chapter 4.

2.5.1 Alternate product form update

Although the variants of the product form update set out below are relatively simple, the author is unaware of them having been described before. This is possibly because, until now, the scope for them to be useful has not arisen.

The elementary matrix for a general rank-one update

The updates introduced below are based on elementary matrices for rank-one updates which are more general than the eta matrices of LU factors or the PF and FT updates. This general elementary matrix can be expressed as

$$T = I + \boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}, \tag{2.29}$$

where $\boldsymbol{u}$ and $\boldsymbol{v}$ are arbitrary compatible vectors. When $\boldsymbol{v} = \boldsymbol{e}_p$, T is an eta matrix E (2.7) and when $\boldsymbol{u} = \boldsymbol{e}_p$, T is a row-wise eta matrix R (2.23). When T is nonsingular,

$$T^{-1} = I - \frac{1}{\mu}\boldsymbol{u}\boldsymbol{v}^{\mathrm{T}}, \tag{2.30}$$

where $\mu = 1 + \boldsymbol{v}^{\mathrm{T}}\boldsymbol{u}$. Clearly T and its inverse can be represented by $\boldsymbol{u}$ and $\boldsymbol{v}$. Although strictly unnecessary, when operating with T^{-1} it is also convenient to record the value of μ. Solving linear systems with T is straightforward, the operations with T^{-1} for FTRAN being

$$T^{-1}\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{v}^{\mathrm{T}}\boldsymbol{x}}{\mu}\boldsymbol{u}, \tag{2.31}$$

and those for BTRAN

$$\boldsymbol{x}^{\mathrm{T}}T^{-1} = \boldsymbol{x}^{\mathrm{T}} - \frac{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{u}}{\mu}\boldsymbol{v}^{\mathrm{T}}. \tag{2.32}$$

The alternate product form update

Working from the basis update expression (2.6), if B is taken out as a factor on the *right* then

$$\bar{B} = (I + (\boldsymbol{a}_q - B\boldsymbol{e}_p)\hat{\boldsymbol{e}}_p^{\mathrm{T}})B = (I + (\boldsymbol{a}_q - \boldsymbol{a}_{p'})\hat{\boldsymbol{e}}_p^{\mathrm{T}})B,$$

where p' is the index within A of column p of B. The matrix $T = I + (\boldsymbol{a}_q - \boldsymbol{a}_{p'})\hat{\boldsymbol{e}}_p^{\mathrm{T}}$ is in the general form (2.29) so, using (2.30), the inverse of the updated basis matrix is given by

$$\bar{B}^{-1} = B^{-1}(I - \frac{1}{\mu}(\boldsymbol{a}_q - \boldsymbol{a}_{p'})\hat{\boldsymbol{e}}_p^{\mathrm{T}}),$$

where $\mu = 1 + \hat{\boldsymbol{e}}_p^{\mathrm{T}}(\boldsymbol{a}_q - \boldsymbol{a}_{p'}) = \hat{a}_{pq}$ is the p^{th} entry of $\hat{\boldsymbol{a}}_q$. In contrast to the PF update, which requires the FTRAN result $\hat{\boldsymbol{a}}_q$, the new update formula uses the BTRAN result $\hat{\boldsymbol{e}}_p$ and thus is called the *alternate* product form (APF) update. It follows from (2.31) that the operation with T^{-1} for FTRAN is

$$T^{-1}\boldsymbol{x}: \quad y = \hat{\boldsymbol{e}}_p^{\mathrm{T}}\boldsymbol{x}/\mu \quad \text{and then} \quad \boldsymbol{x} := \boldsymbol{x} - y(\boldsymbol{a}_q - \boldsymbol{a}_{p'})$$

and, from (2.32), that the corresponding operation for BTRAN is

$$\boldsymbol{x}^{\mathrm{T}}T^{-1}: \quad y = \boldsymbol{x}^{\mathrm{T}}(\boldsymbol{a}_q - \boldsymbol{a}_{p'})/\mu \quad \text{and then} \quad \boldsymbol{x}^{\mathrm{T}} := \boldsymbol{x}^{\mathrm{T}} - y\hat{\boldsymbol{e}}_p^{\mathrm{T}}.$$

After k APF updates, the basis matrix and its inverse can be expressed as

$$B_k = T_kT_{k-1}\dots T_1B_0 \quad \Rightarrow \quad B_k^{-1} = B_0^{-1}T_1^{-1}\dots T_{k-1}^{-1}T_k^{-1}.$$

Note that, in contrast to the PF update, the elementary matrices which constitute the update are applied before rather than after the LU decomposition of B_0 when solving forward systems with B_k^{-1}.

Discussion.

The relation between the APF and PF updates may be viewed as being analogous to that between the *alternative block LU* update (introduced and implemented by Hall [27]) and the

Block LU (BLU) update of Eldersveld and Saunders [15]. However, the APF is distinctive since it avoids the requirement for an invertible representation of a Schur complement. This yields a significant overhead when large numbers of updates are performed.

Relative to the PF update, trading operations with x_p for operations with $\boldsymbol{a}_q - \boldsymbol{a}_{p'}$ makes the APF update appear unattractive. However, there is minimal additional storage overhead since $\boldsymbol{a}_q$ and $\boldsymbol{a}_{p'}$ are columns from the coefficient matrix A so may be represented by the indices q and p'. Hall and McKinnon [32] also identify (classes of) LP problems where it is typical for $\hat{\boldsymbol{a}}_q$ to be dense but $\hat{\boldsymbol{e}}_p$ to be sparse, in which case the APF update will require significantly less storage. Performance-wise, unless $\hat{\boldsymbol{e}}_p$ is significantly more sparse than $\hat{\boldsymbol{a}}_q$, the overhead of the operations with $\boldsymbol{a}_q - \boldsymbol{a}_{p'}$ is such that using the APF rather than the PF update can be expected to be less efficient.

Since the cost of using the APF update corresponds to the density of the final BTRAN result $\hat{\boldsymbol{e}}_p$, rather than partial results $\tilde{\boldsymbol{a}}_q$ and $\tilde{\boldsymbol{e}}_p$ that underpin the FT update, it is expected that, like the PF update, the storage requirement and performance of the APF update will be inferior to those of the FT update. However, it is shown in Chapter 4 that the APF is particularly valuable in the context of a high performance parallel scheme for the dual simplex method since it permits particular multiple FTRAN operations to be performed as a single FTRAN. The number of elementary APF operations that must be performed is similar to the number of FTRAN operations. Thus the saving of all but one of these FTRANs that is achieved by using the APF update is not expected to be compromised by the overhead of maintaining or applying APF updates rather than an alternative scheme.

2.5.2 Middle product form update

The *middle* product form (MPF) update inserts the updates in product form into the middle of factors L and U. In detail, the MPF update is derived as follows from the basis update expression (2.6), assuming that $B = LU$.

$$\begin{aligned}
\bar{B} &= LU + (\boldsymbol{a}_q - B\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}} \\
&= LU + LL^{-1}(\boldsymbol{a}_q - B\boldsymbol{e}_p)\boldsymbol{e}_p^{\mathrm{T}}U^{-1}U \\
&= L(I + (\tilde{\boldsymbol{a}}_q - U\boldsymbol{e}_p)\tilde{\boldsymbol{e}}_p^{\mathrm{T}})U \\
&= L(I + (\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)\tilde{\boldsymbol{e}}_p^{\mathrm{T}})U
\end{aligned}$$

where $\tilde{\boldsymbol{a}}_q = L^{-1}\boldsymbol{a}_q$ and $\tilde{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}}U^{-1}$ are partial FTRAN and BTRAN results respectively, and $\boldsymbol{u}_p = U\boldsymbol{e}_p$ is the p^{th} column of U. The matrix $T = I + (\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)\tilde{\boldsymbol{e}}_p^{\mathrm{T}}$ is in the general form (2.29) so, using (2.30), the inverse of the updated basis matrix is given by

$$\bar{B}^{-1} = U^{-1}(I - \frac{1}{\mu}(\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)\tilde{\boldsymbol{e}}_p^{\mathrm{T}})L^{-1},$$

where $\mu = 1 + \tilde{\boldsymbol{e}}_p^{\mathrm{T}}(\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p) = \hat{a}_{pq}$ is the p^{th} entry of $\hat{\boldsymbol{a}}_q$. It follows from (2.31) that the operation with T^{-1} for FTRAN is

$$T^{-1}\boldsymbol{x}: \quad y = \tilde{\boldsymbol{e}}_p^{\mathrm{T}}\boldsymbol{x}/\mu \quad \text{and then} \quad \boldsymbol{x} := \boldsymbol{x} - y(\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)$$

and, from (2.32), that the corresponding operation for BTRAN is

$$\boldsymbol{x}^{\mathrm{T}}T^{-1}: \quad y = \boldsymbol{x}^{\mathrm{T}}(\tilde{\boldsymbol{a}}_q - \boldsymbol{u}_p)/\mu \quad \text{and then} \quad \boldsymbol{x}^{\mathrm{T}} := \boldsymbol{x}^{\mathrm{T}} - y\tilde{\boldsymbol{e}}_p^{\mathrm{T}}.$$

After k updates, the basis matrix and its inverse can be expressed as

$$B_k = L_0 T_1 T_2 \dots T_k U_0 \quad \Rightarrow \quad B_k^{-1} = U_0^{-1} T_k^{-1} \dots T_2^{-1} T_1^{-1} L_0^{-1}.$$

Discussion. The relation between the MPF and PF updates is analogous to that between the *partitioned LU* update (introduced by Gill *et al.* [21] but not implemented) and the BLU update of Eldersveld and Saunders [15]. However, like the APF, the MPF is distinctive since it avoids a Schur complement.

The MPF update is comparable with the FT update since both are based on the partial FTRAN result $\tilde{\boldsymbol{a}}_q$ and partial BTRAN result $\tilde{\boldsymbol{e}}_p$. However the MPF has no deletion corresponding to that of the FT update so will have greater storage overhead. In terms of update efficiency, the MPF update is preferable since it only involves storing operations, whereas the FT update needs additional deletion and insertion work to update the factor U. In terms of solving efficiency, it is expected to be preferable to operate with the FT update since it replaces $\boldsymbol{u}_p$ by $\tilde{\boldsymbol{a}}_q$ and stores only $\tilde{\boldsymbol{e}}_p$ as an eta matrix R. Thus solving with the FT update requires only additional simple eta matrix operations with R, while solving with the MPF requires slightly more complicated and time-consuming operations with T.

2.5.3 Collective Forrest-Tomlin update

The collective Forrest-Tomlin (CFT) update is designed to perform a set of t Forrest-Tomlin operations simultaneously to obtain the FT invertible representation of B_{k+t} directly from that of B_k. This requirement arises naturally within the dual simplex method with suboptimization.

The dual simplex method with suboptimization

The details of this simplex variant are given Chapter 4. However, it is necessary to set out some of its data requirements to motivate the collective Forrest-Tomlin update described below. Suboptimization in the dual simplex method requires the vectors $\hat{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}} B_k^{-1}$ for p in a small subset $\mathcal{P}$ of the rows. Iterations of suboptimization then identify t basis changes given by $\{(p_i,\ q_i)\}_{i=0}^{t-1}$, where $\{p_i\}_{i=0}^{t-1} \subseteq \mathcal{P}$. When, during the course of suboptimization, the vector $\boldsymbol{e}_{p_i}^{\mathrm{T}} B_{k+i}^{-1}$ is required, it is obtained from $\hat{\boldsymbol{e}}_{p_i}$ by a few APF operations. Following the suboptimization iterations, updating data for the whole LP problem requires the pivotal columns $\hat{\boldsymbol{a}}_{q_i} = B_{k+i}^{-1}\boldsymbol{a}_{q_i}$ for $i = 0, \dots, t-1$, which are also obtained in two steps as $\hat{\boldsymbol{a}}_q = B_k^{-1}\boldsymbol{a}_q$ and then regular PF operations.

The Forrest-Tomlin update after suboptimization

To perform Forrest-Tomlin updates after suboptimization requires

$$\tilde{\boldsymbol{a}}_{q_i} = R_{k+i}^{-1} \dots R_k^{-1} \dots R_1^{-1} L^{-1} \boldsymbol{a}_{q_i} \quad \text{and} \quad \tilde{\boldsymbol{e}}_{p_i}^{\mathrm{T}} = \boldsymbol{e}_{p_i}^{\mathrm{T}} U_{k+i}^{-1}, \tag{2.33}$$

for $i = 0, \dots, t-1$, so that the next elimination matrix R_{k+i+1} can be constructed using $\tilde{\boldsymbol{e}}_{p_i}^{\mathrm{T}}$ and the new column transformation can be formed as $R_{k+i+1}^{-1}\tilde{\boldsymbol{a}}_{q_i}$. In the sequential simplex method, they are naturally available as partial results of the routinely solved forward (2.3) and transposed (2.4) systems. In suboptimization, initialisation of $\hat{\boldsymbol{e}}_p^T = \boldsymbol{e}_p^T B_k^{-1}$ and the subsequent computation of $\hat{\boldsymbol{a}}_q = B_k^{-1}\boldsymbol{a}_q$ yield only

$$\bar{\boldsymbol{a}}_{q_i} = R_k^{-1} \dots R_1^{-1} L^{-1} \boldsymbol{a}_{q_i} \quad \text{and} \quad \bar{\boldsymbol{e}}_{p_i}^{\mathrm{T}} = \boldsymbol{e}_{p_i}^{\mathrm{T}} U_k^{-1}. \tag{2.34}$$

The challenge, therefore, is to obtain the partial results for the appropriate basis (2.33) from the partial results (2.34) obtained naturally, for $i = 1, \dots, t-1$. The results for $i = 0$ are known. The rest of this section will discuss how to achieve this efficiently using simple linear algebra operations.

Updating the partial FTRAN results

Updating the partial FTRAN results $\bar{\boldsymbol{a}}_{q_i}$ to $\tilde{\boldsymbol{a}}_{q_i}$ is a relatively straightforward task. By comparing the available (2.34) and the required (2.33) results, the updating operation is readily identified as applying new row eta transformations after R_k,

$$\tilde{\boldsymbol{a}}_{q_i} = R_{k+i}^{-1} \dots R_{k+2}^{-1} R_{k+1}^{-1} \bar{\boldsymbol{a}}_{q_i},$$

where each new row transformation R_{k+j}, $(0 < j \leq i)$ is available once $\tilde{\boldsymbol{e}}_{p_{j-1}}$ is computed. Thus the computation of $\tilde{\boldsymbol{a}}_{q_i}$ is scheduled after the computation of $\tilde{\boldsymbol{e}}_{p_{i-1}}$, which corresponds to the latest required row transformation R_{k+i}.

Updating the partial BTRAN results

Compared to updating the partial FTRAN results, updating the partial BTRAN results from

$$\bar{\boldsymbol{e}}_{p_i}^{\mathrm{T}} = \boldsymbol{e}_{p_i}^{\mathrm{T}} U_k^{-1} \quad \text{to} \quad \tilde{\boldsymbol{e}}_{p_i}^{\mathrm{T}} = \boldsymbol{e}_{p_i}^{\mathrm{T}} U_{k+i}^{-1}$$

is more complicated because the difference between U_k and U_{k+i} when $i \geq 1$ involves (multiple) replacements and eliminations. However, by carefully rearranging the replacement and elimination involved, computation of $\tilde{\boldsymbol{e}}_{p_i}$ can still be achieved by simple linear algebra operations. Their derivation is explored in detail by updating $\bar{\boldsymbol{e}}_{p_1}$ to $\tilde{\boldsymbol{e}}_{p_1}$, with reference to the first two upper factors U_k and U_{k+1}.

Recall that the Forrest-Tomlin update derives U_{k+1} via the following elimination operations on the spiked matrix U_k'

$$U_{k+1} = R_{k+1}^{-1} U_k', \tag{2.35}$$

where $U_k' = U_k + (\tilde{\boldsymbol{a}}_{q_0} - U_k \boldsymbol{e}_{p_0})\boldsymbol{e}_{p_0}^{\mathrm{T}}$. Rearranging U_k' by taking out U_k as a factor on the left, it follows that

$$\begin{aligned} U_k' &= U_k + U_k(U_k^{-1}\tilde{\boldsymbol{a}}_{q_0} - \boldsymbol{e}_{p_0})\boldsymbol{e}_{p_0}^{\mathrm{T}} \\ &= U_k\left(I + (\hat{\boldsymbol{a}}_{q_0} - \boldsymbol{e}_{p_0})\boldsymbol{e}_{p_0}^{\mathrm{T}}\right) \\ &= U_k E_{k+1}, \end{aligned}$$

where $E_{k+1} = I + (\hat{\boldsymbol{a}}_{q_0} - \boldsymbol{e}_{p_0})\boldsymbol{e}_{p_0}^{\mathrm{T}}$ is the eta matrix used in the PF update (2.25). By substituting $U_k' = U_k E_{k+1}$ into (2.35), U_{k+1} and its inverse can be represented by

$$U_{k+1} = R_{k+1}^{-1} U_k E_{k+1} \quad \text{and} \quad U_{k+1}^{-1} = E_{k+1}^{-1} U_k^{-1} R_{k+1}. \tag{2.36}$$

By using this new representation for U_{k+1}^{-1}, the calculation of $\tilde{\boldsymbol{e}}_{p_1}$ can be considered in three steps applying E_{k+1}^{-1}, U_k^{-1} and R_{k+1} respectively. Using $\boldsymbol{y}$ to represent intermediate results, starting from $\boldsymbol{y} = \boldsymbol{e}_{p_1}$, the following three-step process is given by (2.36).

(1) Applying E_{k+1}^{-1} updates $\boldsymbol{y}$ by

$$\boldsymbol{y}^{\mathrm{T}} := \boldsymbol{y}^{\mathrm{T}} E_{k+1}^{-1} = \boldsymbol{e}_{p_1}^{\mathrm{T}} E_{k+1}^{-1} = \begin{cases} \boldsymbol{e}_{p_1}^{\mathrm{T}} + \mu \boldsymbol{e}_{p_0}^{\mathrm{T}} & p_1 \neq p_0 \\ \alpha \boldsymbol{e}_{p_1}^{\mathrm{T}} & p_1 = p_0 \end{cases}$$

where $\alpha = 1/\hat{a}_{p_0 q_0}$ and $\mu = -\hat{a}_{p_1 q_0}/\hat{a}_{p_0 q_0}$ are the pivotal entry and the p_1^{th} entry of the eta vector of E_{k+1}^{-1}. The first situation $p_1 \neq p_0$ is the common case, which is implied in the suboptimization framework, as each row in $\mathcal{P}$ is used only once. In more general applications, it is possible to have $p_1 = p_0$. However, since this case corresponds to $\mu = 0$ and a scaling of $\boldsymbol{y}$, it is reasonable and convenient to omit it from the following analysis.

(2) Applying U_k^{-1} to $\boldsymbol{y}$ gives

$$\boldsymbol{y}^{\mathrm{T}} := \boldsymbol{y}^{\mathrm{T}} U_k^{-1} = (\boldsymbol{e}_{p_1}^{\mathrm{T}} + \mu \boldsymbol{e}_{p_0}^{\mathrm{T}}) U_k^{-1} = \bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} + \mu \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}}.$$

This expresses $\boldsymbol{y}$ in terms of the available partial BTRAN results $\tilde{\boldsymbol{e}}_{p_0}$ and $\bar{\boldsymbol{e}}_{p_1}$, where both are computed with U_k^{-1}.

(3) Applying R_{k+1} to $\boldsymbol{y}$ completes the process, giving the required partial BTRAN result $\tilde{\boldsymbol{e}}_{p_1}$ thus.

$$\tilde{\boldsymbol{e}}_{p_1}^{\mathrm{T}} = \boldsymbol{y}^{\mathrm{T}} R_{k+1} = \bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} R_{k+1} + \mu \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}} R_{k+1} \tag{2.37}$$

When transposed as $R_{k+1}^{\mathrm{T}} \bar{\boldsymbol{e}}_{p_1}$, calculation of the first term in (2.37) is seen to be a standard FTRAN operation (2.9) which adds a multiple $[\bar{\boldsymbol{e}}_{p_1}]_{p_0}$ of $\boldsymbol{r}_{k+1}$ to $\bar{\boldsymbol{e}}_{p_1}$ so

$$\bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} R_{k+1} = \bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} + [\bar{\boldsymbol{e}}_{p_1}]_{p_0} \boldsymbol{r}_{k+1}^{\mathrm{T}}. \tag{2.38}$$

As for the second term in (2.37), by substituting for R_{k+1} using (2.23) and $\boldsymbol{r}_{k+1}$ using (2.27),

$$\begin{aligned} \mu \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}} R_{k+1} &= \mu \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}} (I + \boldsymbol{e}_{p_0} \boldsymbol{r}_{k+1}^{\mathrm{T}}) \\ &= \mu \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}} + \mu (\boldsymbol{e}_{p_0}^{\mathrm{T}} - u_{p_0 p_0} \tilde{\boldsymbol{e}}_{p_0}^{\mathrm{T}})/u_{p_0 p_0} \\ &= \frac{\mu}{u_{p_0 p_0}} \boldsymbol{e}_{p_0}^{\mathrm{T}} \end{aligned} \tag{2.39}$$

so the second term in (2.37) is seen as adding the scalar value $\mu/u_{p_0 p_0}$ to y_{p_0}. Substituting (2.38) and (2.39) for the two terms in (2.37) yields the following update formula.

$$\tilde{\boldsymbol{e}}_{p_1}^{\mathrm{T}} = \bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} + [\bar{\boldsymbol{e}}_{p_1}]_{p_0} \boldsymbol{r}_{k+1}^{\mathrm{T}} + \frac{\mu}{u_{p_0 p_0}} \boldsymbol{e}_{p_0}^{\mathrm{T}} \tag{2.40}$$

In practice, to obtain $\mu = -\hat{a}_{p_1 q_0}/\hat{a}_{p_0 q_0}$ requires only one inner product, $\hat{a}_{p_1 q_0} = \bar{\boldsymbol{e}}_{p_1}^{\mathrm{T}} \tilde{\boldsymbol{a}}_{q_0}$, since $\hat{a}_{p_0 q_0}$ is available as the simplex pivotal entry. Thus updating the partial BTRAN result from $\bar{\boldsymbol{e}}_{p_1}$ to $\tilde{\boldsymbol{e}}_{p_1}$ involves only one vector addition and the evaluation of one inner product.

The partial BTRAN result $\bar{\boldsymbol{e}}_{p_2}^{\mathrm{T}} = \boldsymbol{e}_{p_2}^{\mathrm{T}} U_k^{-1}$ corresponding to the third pivot choice, can be updated in two steps, firstly to $\boldsymbol{e}_{p_2}^{\mathrm{T}} U_{k+1}^{-1}$ by using (2.40), and then to $\boldsymbol{e}_{p_2}^{\mathrm{T}} U_{k+2}^{-1}$ by repeating the same operations with partial results $\tilde{\boldsymbol{e}}_{p_1}$ and $\tilde{\boldsymbol{a}}_{q_1}$. Using this stepwise updating procedure for $i = 1, \ldots, t-1$, the partial BTRAN result $\bar{\boldsymbol{e}}_{p_i}$ corresponding to the $(i+1)^{\text{st}}$ pivot choice can be updated to the required form $\tilde{\boldsymbol{e}}_{p_i}$ in i steps of the form (2.40). Note that to apply (2.40) requires the previous partial FTRAN results $\tilde{\boldsymbol{a}}_{p_j}$ for $j = 0, \dots, i-1$. Thus the updating of $\bar{\boldsymbol{e}}_{p_i}$ is scheduled after the computation of last required partial FTRAN result $\tilde{\boldsymbol{a}}_{q_{i-1}}$. Therefore, together with the data requirement of operations with $\tilde{\boldsymbol{a}}_i$ discussed previously, the calculation for updating the two types of partial transformed results is interlaced as the sequence $\tilde{\boldsymbol{a}}_{q_1}, \tilde{\boldsymbol{e}}_{p_1}, \tilde{\boldsymbol{a}}_{q_2}, \tilde{\boldsymbol{e}}_{p_2}$, etc.

2.5.4 Results and analysis

The serial efficiency of the novel updates is assessed in this section using the reference set. Numerical results are presented in Table 2.4.

MODEL	Fill-in	FTRAN	BTRAN	PF	APF	MPF	FT	CFT
CRE-B	1.01	100	83	**0.4**	1.0	0.5	0.5	0.3
DANO3MIP_LP	1.42	1	6	25.1	26.8	**11.1**	7.9	7.1
DBIC1	1.01	100	83	21.4	39.6	**10.5**	9.0	7.5
DCP2	1.02	100	97	13.6	7.4	**6.4**	2.9	2.9
DFL001	1.28	34	57	11.6	15.8	**5.3**	4.0	3.6
FOME12	1.29	45	58	81.7	124.7	**40.1**	27.7	25.9
FOME13	1.30	100	98	218.7	352.4	**108.1**	67.4	60.3
KEN-18	1.00	100	100	9.6	10.0	**4.2**	2.3	2.3
L30	2.06	10	8	6.9	6.6	**2.8**	2.1	2.1
LINF_520C	2.04	10	11	17665.1	11516.2	**3354.9**	1304.3	1272.0
LP22	1.82	13	22	12.8	12.6	**7.1**	5.0	4.6
MAROS-R7	2.00	5	13	7.5	**5.4**	5.7	3.6	3.5
MOD2	1.03	46	68	69.4	57.4	**19.0**	16.1	13.8
NS1688926	1.00	72	100	41.5	**8.7**	9.3	9.0	8.8
NUG12	3.86	1	20	120.5	126.1	**58.0**	32.7	31.3
PDS-40	1.01	100	98	17.0	34.2	**8.6**	5.9	5.5
PDS-80	1.01	100	99	39.0	106.1	**20.9**	12.5	11.7
PDS-100	1.00	100	99	47.8	147.2	**26.4**	15.5	14.5
PILOT87	2.03	10	19	4.2	4.2	**2.9**	1.8	1.7
QAP12	4.13	2	15	151.1	167.3	**74.3**	41.8	39.8
SELF	1.68	0	2	5.2	6.2	**5.1**	4.2	4.2
SGPF5Y6	1.00	100	100	25.0	20.4	**11.5**	5.2	6.0
STAT96V4	1.25	73	31	15.7	28.9	**9.0**	6.9	6.4
STORMG2-125	1.00	100	100	5.2	2.1	**1.8**	0.9	1.1
STORMG2-1000	1.00	100	100	361.0	**53.9**	60.5	29.3	28.6
STP3D	1.04	95	70	421.1	801.5	**124.2**	124.5	101.9
TRUSS	1.37	37	2	1.5	2.4	**0.9**	0.7	0.8
WATSON_1	1.00	100	100	31.1	15.9	**11.2**	4.1	4.7
WATSON_2	1.00	100	100	45.9	17.2	**14.9**	4.9	6.5
WORLD	1.04	41	61	82.4	72.5	**22.7**	19.3	16.4

Table 2.4: Performance of various simplex update approaches

Of particular relevance to the experiments discussed below is the average relative size of the matrix B_0 and its invertible representation, given in the column of Table 2.4 headed “Fill-in”, and the proportion of the results of FTRAN and BTRAN with density less than 10%, given in the corresponding columns of Table 2.4. Hall and McKinnon [32] introduced this measure of an LP problem’s hyper-sparsity and discuss reasons for the extreme variance between these values for some problems. Unsurprisingly there is clear correlation between the fill-in during INVERT and the measures of hyper-sparsity.

Experiments with the PF, APF, MPF, FT and CFT updates were performed using the same sequence of basis changes from a “logical” basis $B = I$ to an optimal solution of the LP. This eliminates variance in the results due to the fact that a change in update technique typically leads to the simplex method taking a different path to an optimal solution. The same reinversion interval determined using FT update was also used for all experiments with a given problem. This facilitates some comparisons between methods and ensures that the FT update is not disfavoured.

The principal measure of the efficiency of a particular update procedure is the total CPU time to perform the update and then perform FTRAN and BTRAN for each basis in the sequence. This is given in the columns headed PF, APF, MPF, FT and CFT in Table 2.4 where, for each LP problem, the best performance of the three product form updates is highlighted in bold.

The superiority of the MPF update over the PF and APF updates is clear: it is the best for 26 of the 30 problems. Of the other four problems, the APF update is the best for 3 problems and in all of these cases the proportion of sparse BTRAN results is at least that of FTRAN. However, on 15 of the 30 problems the APF update is the worst, with the PF update being the worst for the others. A further measure of the superiority of the MPF update is obtained by considering the CPU time relative to the better of the CPU time for the PF and APF updates. The geometric mean of this ratio shows the use of the MPF update to be 42% more efficient.

The FT update is better than the best of the product form updates for 27 of the problems and, in average, is 63% more efficient than the better of the PF and APF updates. However, the FT update is only 35% more efficient than the MPF update. Thus the efficiency of the MPF update is rather closer to that of the FT update than it is to the other product form updates.

The performance when using the collective FT update (CFT) is very similar to that when using the standard FT update for most of these instances. This demonstrates that the CFT update incurs no significant overhead. The occasional difference between the FT and CFT for some LP problems is actually because of the different arrangement of linear systems solution for CFT. When testing with CFT, up to eight FTRAN (and BTRAN) operations are performed continuously with the same basis inverse representation. This possibly affects data affinity (the same basis inverse representation is reused up to seven times) and thus leads to the difference.

Although the deletion and insertion operations of the FT update result in a CPU overhead that is not shared with the product form updates, the overall computational efficiency when using the FT update rather than the MPF or the best of the product form updates is greater than the storage efficiency. This is due to the fact that hyper-sparsity may be exploited fully when using the FT update, but only when operating with B_0^{-1} in the case of the APF and MPF updates.

2.6 Summary

This chapter has introduced the implementation of linear system solution techniques and described three novel update procedures.

Two of novel update techniques are variants of the product form and one is an extension of the Forrest-Tomlin (FT) update. Of the two product form variants, the middle product form (MPF) update is generally much more efficient than the alternate product form (APF) update. The APF is very inefficient for some problems and is not recommended as a valuable general technique. However, its limited use in a variant of the dual revised simplex method that is amenable to parallelization yields a valuable computational saving. The performance of the MPF is generally very much better than that of the APF and original product form update. Indeed, its performance frequently approaches that of the FT update. Since the MPF update does not require any elimination operations or dynamic data structures it is very much easier to implement than the FT update so is an attractive update procedure when developing a simple, relatively efficient implementation of the revised simplex method.

The collective Forrest-Tomlin (CFT) update organises the calculations required to perform multiple FT updates with the same efficiency as the corresponding sequence of standard FT updates. This ensures that there is no loss of serial efficiency when the CFT is used in the context of the dual revised simplex method with suboptimization.

Chapter 3

Sequential simplex methods

This chapter presents implementation experiences of a sequential dual revised simplex solver. Section 3.1 introduces several fundamental concepts. Section 3.2 describes the computational components of a dual revised simplex method, mainly the advanced pivoting rules. Section 3.3 firstly analyses the relation between performance improvement and usage of cost perturbation, and then based on which, discusses two experimental alternative approaches. The overall performance of the resulting dual simplex implementation is comparable to the dual simplex solver of Clp, providing a solid foundation for the paralleization.

3.1 Fundamental concepts

This section introduces several fundamental concepts of linear programming and the simplex method algorithm for advanced discussion.

3.1.1 LP problem and basic solution

A linear programming (LP) problem in standard form aims to optimize (minimize) a linear objective function of n non negative variables with a set of $m \le n$ linear constraints

$$\begin{array}{rl} \text{minimize} & c_1x_1 \;+\; c_2x_2 \;+\ldots+\; c_nx_n \\ \text{subject to} & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ & \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \\ \text{and} & x_1 \ge 0, x_2 \ge 0, \ldots, x_n \ge 0. \end{array}$$

In matrix notation, the standard form is simplified as

$$\text{minimize } f = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, \text{ subject to } A\boldsymbol{x} = \boldsymbol{b} \text{ and } \boldsymbol{x} \ge \boldsymbol{0}, \tag{3.1}$$

where the matrix $A \in \mathbb{R}^{m\times n}$ is called the coefficient matrix, vectors $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{x} \in \mathbb{R}^n$ are called the cost vector, right-hand-side (RHS) vector and variable vector respectively.

LP solution. Any vector $\boldsymbol{x}$ which satisfies the linear equation constraint set $A\boldsymbol{x} = \boldsymbol{b}$ is called a *solution* to the LP problem. If it also satisfies the non negativity requirement $\boldsymbol{x} \ge \boldsymbol{0}$, then the solution vector $\boldsymbol{x}$ becomes a *feasible solution*. The goal of linear programming is to search for a feasible solution which minimizes the objective function $f = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$. When such a feasible solution is obtained, the LP problem is said to have been solved to *optimality* and the associated feasible solution is called an *optimal solution*.

Basic solution. Assuming that the coefficient matrix A is of full rank, then it is always possible to identify a non singular basis partition $B \in \mathbb{R}^{m\times m}$ which consists of m linearly independent columns of A. The remaining columns of A are called the non basic partition, denoted by N. By partitioning $\boldsymbol{x}$ accordingly, the linear equation constraint set $A\boldsymbol{x} = \boldsymbol{b}$ can be expressed as $B\boldsymbol{x}_B + N\boldsymbol{x}_N = \boldsymbol{b}$. Because B is non singular (and therefore invertible), it can be further rewritten as

$$\boldsymbol{x}_B = B^{-1}\boldsymbol{b} - B^{-1}N\boldsymbol{x}_N, \tag{3.2}$$

where the values of basic variables $\boldsymbol{x}_B$ are computed using the values of nonbasic variables $\boldsymbol{x}_N$. When nonbasic variables are set to their bounds as $\boldsymbol{x}_N = \boldsymbol{0}$, then the basic variables are computed by $\boldsymbol{x}_B = \hat{\boldsymbol{b}} = B^{-1}\boldsymbol{b}$. The solution given by setting each nonbasic variable to its bound and computing the basic variables by equation (3.2) is called a *basic solution*. It is obvious that the nonbasic variables $\boldsymbol{x}_N$ satisfy their bounds in a basic solution. If all the computed basic variables also satisfy $\boldsymbol{x}_B = \hat{\boldsymbol{b}} \geq \boldsymbol{0}$, then the basic solution is called a *basic feasible solution* or BFS for short. It can be proved that, the optimal solution of an LP problem is either a BFS or linear combination of many such solutions with the same objective value.

Reduced cost. By replacing basic variables by (3.2) in the partitioned objective function $f = \boldsymbol{c}_B^{\mathrm{T}}\boldsymbol{x}_B + \boldsymbol{c}_N^{\mathrm{T}}\boldsymbol{x}_N$, it follows that

$$f = \boldsymbol{c}_B^{\mathrm{T}}B^{-1}\boldsymbol{b} + (\boldsymbol{c}_N^{\mathrm{T}} - \boldsymbol{c}_B^{\mathrm{T}}B^{-1}N)\boldsymbol{x}_N = \boldsymbol{c}_B^{\mathrm{T}}B^{-1}\boldsymbol{b} + \hat{\boldsymbol{c}}_N^{\mathrm{T}}\boldsymbol{x}_N,$$

where the vector

$$\hat{\boldsymbol{c}}_N^{\mathrm{T}} = \boldsymbol{c}_N^{\mathrm{T}} - \boldsymbol{c}_B^{\mathrm{T}}B^{-1}N \tag{3.3}$$

is called *relative cost factor* or *reduced cost*. Each $\hat{c}_j$ represents the per unit change of the objective function when increasing the corresponding nonbasic variable x_j.

3.1.2 The primal and dual simplex algorithms

The simplex algorithm always starts with a BFS, keeps moving from one BFS to a more profitable BFS, until the LP is solved to optimality or proved unbounded. Assuming that an initial BFS is available, each iteration of the simplex algorithm can be described as three major steps

1. *Optimality test.* Scan the reduced costs of nonbasic variables and choose variable q with $\hat{c}_q < 0$, so that the objective function will decrease when increasing the corresponding variable x_q. Often the one associated with the most negative reduced cost is chosen by

   $$q = \arg\min_{j\in\mathcal{N}} \hat{c}_j.$$

   If no such variable exists, then the simplex algorithm terminates and declares the LP problem solved to *optimality* with the current BFS.

2. *Ratio test.* By increasing x_q from its zero bound, variable q is no longer a qualified nonbasic variable. It will become a basic variable at the end of the current simplex iteration, and thus it is called an *entering* variable. Meanwhile, because of the increase of x_q, the basic variables $\boldsymbol{x}_B$ are changed by $\Delta\boldsymbol{x}_B = -\Delta x_q B^{-1}\boldsymbol{a}_q$. By increasing Δx_q, one of the basic variables, x_p will be zeroed first. It will become a nonbasic variable at the end of the current simplex iteration, and thus it is often called a *leaving* variable. Mathematically, the leaving variable x_p is identified by a *ratio test* between $\boldsymbol{x}_B$ and $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$ by choosing the smallest non negative ratio

$$p = \arg\min_i(\hat{b}_i/\hat{a}_{iq}),\ \hat{a}_{iq} > 0.$$

   If no leaving variable is identified, then the increase of the entering variable x_q is unlimited, so that simplex algorithm terminates and declares the LP problem *unbounded*. On the other hand, if the computed basic variable $x_p = 0$, then increase of x_q is limited to zero, and no objective improvement can be made from the current simplex iteration. This situation is call *degeneracy*.

3. *Update BFS.* After the entering variable x_q and the leaving variable x_p are both identified, x_q will replace x_p from the basic partition to yield a new BFS. The new BFS and old BFS are called neighbouring BFS to each other. Unless the smallest ratio $\theta_p = \hat{b}_p/\hat{a}_{pq} = 0$, the objective function associated with the new BFS will strictly decrease by $\Delta f = \hat{c}_q \times \theta_p$. After the update, the simplex algorithm continues from step 1 with the new BFS.

The logic of the simplex algorithm can be summarized as continuously identifying and increasing a nonbasic variable associated with a negative reduced cost while maintaining the feasibility of basic variables, until the LP problem is solved to optimality or found to be unbounded.

Alternately, the optimal solution can be achieved by starting from all non negative reduced cost $\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$, and then in each iteration, identifying and increasing the value of a negative (infeasible) basic variable $x_i < 0$ to its bound, while maintaining non negativity of the nonbasic reduced costs, until all basic variables satisfy $\boldsymbol{x}_B \geq \boldsymbol{0}$. Because the relation $\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$ is always maintained, when $\boldsymbol{x}_B \geq \boldsymbol{0}$ is also satisfied, then the LP is solved to optimality. The alternate algorithm, called the *dual simplex algorithm*, is firstly developed by Lemke [41] in 1950s. The original simplex algorithm is called the *primal simplex algorithm*. The computational steps of the dual simplex iteration can be introduced in a comparative manner as provided in Table 3.1.

	Primal simplex algorithm	Dual simplex algorithm
Starting BFS	Primal BFS $\boldsymbol{x}_B = \hat{\boldsymbol{b}} \geq \boldsymbol{0}$	Dual BFS $\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$
Optimality test	Choose $q = \arg\min_j \hat{c}_j < 0$ as entering variable, or declare optimal	Choose $p = \arg\min_i \hat{b}_i < 0$ as leaving variable, or declare optimal
Ratio test	Choose $p = \arg\min_i \hat{b}_i/\hat{a}_{iq}$, where $\hat{a}_{iq} > 0$ as leaving variable with $\theta_p = \hat{b}_p/\hat{a}_{pq} \geq 0$, or declare unbounded	Choose $q = \arg\min_j \hat{c}_j/(-\hat{a}_{pj})$, where $\hat{a}_{pj} < 0$ as entering variable with $\theta_d = \hat{c}_q/\hat{a}_{pq} \leq 0$, or declare dual unbounded
Updating	Replace x_p by x_q Obtain new BFS $f := f + \hat{c}_q \times \theta_p$ decreases	Replace x_p by x_q Obtain new dual BFS $f := f + \theta_d \times \hat{b}_p$ increases

Table 3.1: Description of primal and dual simplex algorithms in a comparative manner

In the dual simplex algorithm, a nonbasic variable associated with a non negative (non profitable) reduced cost is called a dual feasible variable. For a basic solution, if all the nonbasic variables are dual feasible ($\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$), then the basic solution is called a dual basic feasible solution, or dual BFS for short. The condition $\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$ is referred to as dual feasibility.

In summary, the primal simplex algorithm seeks dual feasibility ($\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$) while maintaining primal feasibility ($\boldsymbol{x}_B \geq \boldsymbol{0}$); the dual simplex algorithm seeks primal feasibility ($\boldsymbol{x}_B \geq \boldsymbol{0}$) while maintaining dual feasibility ($\hat{\boldsymbol{c}}_N \geq \boldsymbol{0}$).

3.1.3 The tableau and revised simplex methods

So far it is assumed that, a BFS or dual BFS is available before the start of primal or dual simplex algorithm. Practically, this assumption is not always true, so that it is necessary to find a BFS or dual BFS before the start of the simplex algorithm. It is well known that the simplex algorithm itself can be applied to find a BFS by solving a simpler artificial problem or subproblem based on the original LP problem. Finding a starting BFS and obtaining an optimal BFS by using the simplex algorithm are called the *phase I* and *phase II* of the *simplex method.*

In terms of simplex method implementation, there are two major choices, namely the *tableau simplex method* and the *revised simplex method.*

The tableau simplex method

The tableau simplex method works by maintaining the RHS vector $\hat{\boldsymbol{b}}$, reduced cost $\hat{\boldsymbol{c}}_N^{\mathrm{T}}$ and updated coefficient matrix $\hat{A} = B^{-1}[B|N] = [I|\hat{N}]$, in a table called the *tableau*

	$\hat{\boldsymbol{c}}_N^{\mathrm{T}}$	
I	$\hat{N}$	$\hat{\boldsymbol{b}}$

In the tableau, the basic variables are permuted to the first m positions. If the value of a basic variable is computed by the i^{th} equation of the tableau, then it is permuted to the i^{th} position. In the actual implementation, this permutation is often performed implicitly by maintaining a `base` vector, consisting of the ordered index of corresponding coefficient matrix columns. With the tableau, current values of $\hat{\boldsymbol{b}}$, $\hat{\boldsymbol{c}}_N^{\mathrm{T}}$ and each updated row are available directly, and thus the optimality test and ratio test requires no additional computation. As choosing the entering and leaving variables correspond to choosing a column and a row of the tableau, they are traditionally called CHUZC and CHUZR respectively.

In the updating step of each simplex iteration, the whole tableau is updated by Gaussian elimination by using the pivotal row ($\hat{\boldsymbol{a}}_p^{\mathrm{T}}$) to transform the pivotal column $\hat{\boldsymbol{a}}_q$ to a unit column $\boldsymbol{e}_p$ (column p of identity matrix) and update the other columns. Then the two columns p and q are swapped to form a new tableau to finish the update. In particular, the reduced costs and the basic variables are updated by

$$\boldsymbol{x}_B := \boldsymbol{x}_B - \theta_p \hat{\boldsymbol{a}}_q \text{ and } x_p = \theta_p \tag{3.4}$$

and

$$\hat{\boldsymbol{c}} := \hat{\boldsymbol{c}} - \theta_d \hat{\boldsymbol{a}}_p, \tag{3.5}$$

respectively, where $\theta_p = \hat{b}_p / \hat{a}_{pq}$ and $\theta_d = \hat{c}_q / \hat{a}_{pq}$ are primal and dual steps.

The revised simplex method

The revised simplex method is computationally preferable. In the revised simplex method, rather than maintaining the whole updated coefficient matrix $\hat{A} = [I|\hat{N}]$, an inverse (representation) of the basis matrix, B^{-1} is maintained, so that the whole updated coefficient matrix is available as $\hat{A} = B^{-1}A$. In particular, the updated pivotal column required by the ratio test $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$ is computed by FTRAN operation and the vector of basic variables $\boldsymbol{x}_B$ is often maintained and updated by using $\hat{\boldsymbol{a}}_q$ (3.4) at the end of the current iteration.

As for obtaining the reduced cost $\hat{\boldsymbol{c}}$, there are two approaches. The relatively old-fashioned approach computes the reduced cost according its definition

$$\boldsymbol{\pi}^{\mathrm{T}} = \boldsymbol{c}_B^{\mathrm{T}} B^{-1} \text{ and } \hat{\boldsymbol{c}}_N^{\mathrm{T}} = \boldsymbol{c}_N^{\mathrm{T}} - \boldsymbol{\pi}^{\mathrm{T}} N \tag{3.6}$$

where the two computations are known as BTRAN and PRICE respectively. In the implementation, CHUZC can be combined seamlessly with PRICE, where these two procedures are together called *pricing.*

The modern approach maintains and updates the reduced cost by using the tableau row $\hat{\boldsymbol{a}}_p$ as in (3.5). The computation of $\hat{\boldsymbol{a}}_p$ is also achieved in the two steps

$$\hat{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}} B^{-1} \text{ and } \hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} A \tag{3.7}$$

The matrix-vector multiplication in this approach is also referred to as PRICE.

The modern approach is preferred partially because the linear system and matrix-vector multiplication involved in (3.7) are generally sparser that in (3.6), and partially because in the advanced variant of the revised simplex method, $\hat{\boldsymbol{e}}_p^{\mathrm{T}}$ and $\hat{\boldsymbol{a}}_p^{\mathrm{T}}$ are also used by other computational components.

3.1.4 General LP problems and bound types

In real-world applications, LP problems are often provided in a general form

$$\text{minimize } f = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, \text{ subject to } \boldsymbol{b}_L \leq \bar{A}\boldsymbol{x} \leq \boldsymbol{b}_U \text{ and } \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}, \tag{3.8}$$

where the $\boldsymbol{b}_L$ and $\boldsymbol{b}_U$ are called *ranges* of constraints and $\boldsymbol{l}$ and $\boldsymbol{u}$ are called *bounds* of variables respectively. Although it is possible to transform a general form LP to standard form (3.1) to use the regular primal or dual simplex algorithm, it is more convenient and efficient to solve the LP with a so called *computational form* by advanced simplex algorithm variants.

The computational form for an LP is obtained by augmenting $\bar{A}$ by an $m \times m$ identity matrix and m *logical* variables $\boldsymbol{z}$, yielding

$$\text{minimize } f = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, \text{ subject to } [\bar{A}|I](\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{0} \text{ and } (\boldsymbol{l}, -\boldsymbol{b}_U) \leq (\boldsymbol{x}, \boldsymbol{z}) \leq (\boldsymbol{u}, -\boldsymbol{b}_L), \tag{3.9}$$

where the ranges of constraints are transformed to the bounds of logical variables. Within the augmented matrix, the original coefficient matrix $\bar{A}$ and variables $\boldsymbol{x}$ are called *structural* matrix and *structural* variables respectively, the identity matrix is called the *logical* matrix. The usage of logical variables also provide a simple starting basic solution, (not necessarily a BFS), because the associated identity matrix, when used as basis matrix, is effortlessly invertible. Using the identity matrix as the starting basis matrix is a customary approach, called logical starting basis in the revised simplex implementation.

By combining *structural* and *logical* parts, the computational form can be simplified as

$$\text{minimize } f = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}, \text{ subject to } A\boldsymbol{x} = \boldsymbol{0} \text{ and } \boldsymbol{l} \le \boldsymbol{x} \le \boldsymbol{u}, \tag{3.10}$$

where A contains an identity partition (at its end) and the last m entries of the cost vector are all zeros.

For the computational form (3.10), there are five types of bounds for each variable as summarized in Table 3.2. For basic variables, the primal feasibility condition is naturally changed to x_i satisfying $l_i \le x_i \le u_i$. Therefore, a `FREE` variable would never become a leaving variable as it cannot be chosen in primal ratio test. For nonbasic variables, the dual feasibility condition is more complicated and depended on the bound type as indicated in Table 3.2. The underlying logic of dual feasibility is still the same: that a nonbasic variable is dual infeasible if moving the variable x_j away from its bound will decrease the objective function. Therefore, a nonbasic `FIXED` variable is always dual feasible because it cannot be moved.

Names	Variable bound	Dual feasibility
`FIXED`	$-\infty < l = u < \infty$	Always feasible
`LOWER`	$-\infty < l < u = \infty$	$\hat{c} \ge 0$
`UPPER`	$-\infty = l < u < \infty$	$\hat{c} \le 0$
`BOXED`	$-\infty < l < u < \infty$	$\hat{c} \ge 0$ if $x = l$, $\hat{c} \le 0$ if $x = u$.
`FREE`	$-\infty = l < u = \infty$	$\hat{c} = 0$

Table 3.2: General bound types and dual feasibility condition

3.2 The dual revised simplex method

This section introduces the implementation of an advanced dual revised simplex method variant, including the dual steepest-edge algorithm, bound flipping ratio test and dual phase I method.

3.2.1 The standard dual revised simplex algorithm

The dual simplex algorithm, though first introduced in the 1950s by Lemke [41], was not seriously considered as a superior approach until the mid-1990s, when dedicated pivoting techniques for the dual simplex method appeared. Since then, the dual simplex solver has gradually become a popular choice for solving large scale LP problems.

Assuming a dual BFS is available, the computational components of the standard dual revised simplex algorithm can be summarised as six steps

1. CHUZR: Choose a infeasible basic variable p as leaving variable, or declare optimality.
2. BTRAN: Compute $\hat{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}} B^{-1}$
3. PRICE: Compute $\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} N$
4. CHUZC: Perform a dual ratio test (between $\hat{\boldsymbol{c}}_N$ and $\hat{\boldsymbol{a}}_p$) to choose the entering variable q, or declare unboundedness.
5. FTRAN: Compute $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$
6. UPDATE: Update $\boldsymbol{x}_B$ by $\hat{\boldsymbol{a}}_q$ (UPDATE-PRIMAL), $\hat{\boldsymbol{c}}_N$ by $\hat{\boldsymbol{a}}_p$ (UPDATE-DUAL). Update the basis inverse representation (UPDATE-FACTOR), or reinversion if necessary (INVERT).

Of all these components, BTRAN, FTRAN, UPDATE-FACTOR and INVERT are linear system solution techniques and efficient implementation of these operations has been introduced in detail in Chapter 2. Update of basic variables (UPDATE-PRIMAL) and reduced cost (UPDATE-DUAL) are simple vector additions given by (3.4) and (3.5) and thus are not discussed.

The pivoting steps CHUZR and CHUZC and their advanced variants are the key components of the dual revised simplex algorithm. They are described in Sections 3.2.2 and 3.2.3 respectively.

3.2.2 Dual optimality test

Each dual simplex iteration starts from choosing a primal infeasible variable as leaving variable. A simple approach is to choose the one associated with the biggest primal infeasibility Δx_i defined as

$$p = \arg\max_i \Delta x_i, \quad \Delta x_i = \begin{cases} l_i - x_i & x_i < l_i - \epsilon_p \\ x_i - u_i & x_i > u_i + \epsilon_p \\ 0 & \text{otherwise} \end{cases},$$

where the ϵ_p is a small value called the *optimality tolerance* or, more often, the *primal tolerance*. The primal tolerance is applied to relax the bounds of primal variables to exclude small inaccuracies during the computation of the basic variables. Practically, if all primal basic variables are feasible within the relaxed bounds, then the problem is solved to optimality. Common choices of primal tolerance are $\epsilon_p = 10^{-6}$ or $\epsilon_p = 10^{-7}$.

Dual steepest-edge algorithm

The dual CHUZR is more efficiently achieved by a normalized selection. The best known normalized scheme for the dual revised simplex method is the dual steepest edge (DSE) [17] algorithm. In DSE, the basic variable p associated with the biggest weighted infeasibility is chosen as leaving variable

$$p = \arg\max_i \Delta x_i^2 / ||\hat{\boldsymbol{e}}_i||_2^2,$$

where $\hat{\boldsymbol{e}}_i^T = \boldsymbol{e}_i^T B^{-1}$ is the logical partition of the full tableau row $\hat{\boldsymbol{a}}_i^T$ and $w_i = ||\hat{\boldsymbol{e}}_i||_2^2$ is the DSE weight. The weighted infeasibility is called *attractiveness* of a basic variable, denoted $\alpha_i = \Delta x_i^2 / w_i$.

The DSE weight can be updated in each dual simplex iteration. The update formulae given by Forrest and Goldfarb [17] can be expressed as

$$\begin{aligned} w_p &:= w_p / \hat{a}_{pq}^2 \\ w_i &:= w_i - 2(\hat{a}_{iq}/\hat{a}_{pq})\tau_i + (\hat{a}_{iq}/\hat{a}_{pq})^2 w_p \end{aligned}$$

where the vector $\boldsymbol{\tau} = B^{-1}\boldsymbol{e}_p$ is computed by an additional FTRAN operation. The pivotal weight w_p is freshly computed according to its definition as $w_p = ||\hat{\boldsymbol{e}}_p||_2^2$. The recomputation of pivotal weights refreshes the accuracy of DSE weights.

Even though, as an updated value, the DSE weights w_i for each variable inevitably become inaccurate after a large number of simplex iterations. In the worst case, it is possible for them to become negative. To alleviate this issue, as suggested in the original paper, the weight is always set to a small value (10^{-4}) if the computed weight is smaller than that value. Though this solves the negative weight issue, having $w_i = 10^{-4}$ means that the corresponding basic variable can be falsely much more attractive than it should be within the DSE framework.

To address this issue, in HSOL the DSE weight of the leaving variable is computed after BTRAN by its definition. If the updated weight w_p^u is significantly smaller than the computed weight w_p^c by

$$w_p^u < r \times w_p^c,$$

then the current leaving variable is ignored and the dual simplex iteration starts from CHUZR again with correcting w_p by the freshly computed value. In the implementation, $r = 0.5$ is used.

Exploiting hyper-sparsity

For many sparse LP problems, the hyper-sparse CHUZC approach described by Hall and McKinnon [32] for the primal simplex method can be adapted to dual CHUZR. In the primal simplex algorithm, the hyper-sparse CHUZC works by maintaining a short list $\mathcal{L}$ of top attractive dual infeasible candidates, and keeps choosing from the list to save the time for scanning the other less attractive choices. This works because, when the pivotal tableau row $\hat{\boldsymbol{a}}_p^{\mathrm{T}}$ is hyper-sparse (density less than 10%), only a small number of dual reduced costs will be updated by (3.5), thus providing a chance to trade the time of scanning the full reduced cost by maintaining a list of top attractive candidates.

For the dual revised simplex method, hyper-sparsity can be exploited in CHUZR similarly. Specifically, the hyper-sparse CHUZR is initialized only when the LP problem is found to be hyper-sparse.

When creating the infeasible list, assuming t choices are wanted, the t^{th} best attractiveness is firstly identified as the cutoff value α_{cutoff}, and then all these with better attractiveness $\alpha_i > \alpha_{\text{cutoff}}$ are put into the list $\mathcal{L}$. The infeasible list is updated when the primal variables or the DSE weight is updated. A variable is added to the list if $\alpha_i > \alpha_{\text{cutoff}}$.

When choosing the leaving variable with the infeasible list, if the best α_p of $\mathcal{L}$ is smaller than α_{cutoff}, then the best choices may be outside the list. Therefore, the list will be recreated and the current simplex iteration will start again with the new list.

3.2.3 Dual ratio test

Compared to dual CHUZR, choosing entering variables by the dual ratio test is more complicated. Besides the basic logic of the dual ratio test, this section further discusses several advanced techniques, including the bound flipping ratio test (BFRT), the adaption of primal EXPAND (mainly Harris two-pass ratio test) technique and cost perturbation.

Computing $\hat{\boldsymbol{a}}_p^{\mathrm{T}}$

The updated tableau row associated with the leaving variable $\hat{\boldsymbol{a}}_p^{\mathrm{T}}$ is required before the dual ratio test. This is achieved by BTRAN ($\hat{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}} B^{-1}$) and PRICE ($\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} A$). The PRICE computation

$$\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} A$$

can be achieved by computing the inner product of $\hat{\boldsymbol{e}}_p$ and each nonbasic column $\boldsymbol{a}_j$. Alternately, especially when $\hat{\boldsymbol{e}}_p^{\mathrm{T}}$ is sparse, it is more efficient to compute $\hat{\boldsymbol{a}}_p$ by using a row-wise version (both column-wise and row-wise copies of A are used in HSOL) of the coefficient matrix A and adding up rows for each non-zero entry of $\hat{\boldsymbol{e}}_p$ as

$$\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \sum_{\hat{e}_{pi} \neq 0} \hat{e}_{pi} \times \boldsymbol{a}_i^{\mathrm{T}}.$$

This is called the row-wise PRICE.

Basic logic

There are totally eight situations when performing the dual ratio test with the computational form, as a result of the bounds types (four) and infeasibility types (two) of the basic leaving variable. To conveniently deal with all these situations, the dual ratio is split into two major stages, (1) identify all qualified nonbasic variables and merge all situations into one, and (2) choose the entering variable by ratio test.

Qualification. First consider a simple situation. Assuming that the chosen leaving variable is primal infeasible because $x_p < l_p$, it will increase to its lower bound at the end of the current simplex iteration. For a nonbasic variable x_j at its lower bound, because $x_p = -\sum_{(j\in\mathcal{N})} \hat{a}_{pj} x_j$, only when $\hat{a}_{pj} < 0$ does increasing x_j lead to an increase of x_p. Similarly, when $x_p > u_p$, for a nonbasic variable x_j at its lower bound, $\hat{a}_{pj} > 0$ is required. One special situation is a nonbasic `FREE` variable. A `FREE` variable may move up or down, and thus it is qualified if $\hat{a}_{pj} \neq 0$, because, assuming $x_p < l_p$, moving it up when $\hat{a}_{pj} < 0$ or moving it down when $\hat{a}_{pj} > 0$ will both increase the primal leaving variable. Moreover, for a nonbasic `FIXED` variable, it can not be moved and thus does not participate in the ratio test. In total, there are eight situations to consider in the dual ratio test because of four nonbasic bound types ($x_j = l_j$, $x_j = u_j$, `FIXED` or `FREE`) and two infeasibility types ($x_p < l_p$ or $x_p > u_p$).

Merging situations. When implemented in the software, all of these situations can be summarized into one by multiplying the updated entry $\hat{a}_{pj}$ with an infeasibility indicator s_p for the leaving variable and a direction indicator δ_j for each nonbasic variable. The infeasibility indicator, representing the *source* of infeasibility, is defined as

$$s_p = \begin{cases} -1 & \text{if } x_p < l_p \\ 1 & \text{if } x_p > u_p \end{cases}.$$

The direction indicator δ_j for each nonbasic variable, representing its moving direction, is defined as

$$\delta_j = \begin{cases} 0 & \text{if } x_j = l_j = u_j \\ 1 & \text{if } x_j = l_j < u_j \text{ or } x_j \text{ is } \texttt{FREE} \text{ and } \hat{a}_{ij} < 0 \\ -1 & \text{if } x_j = u_j > l_j \text{ or } x_j \text{ is } \texttt{FREE} \text{ and } \hat{a}_{ij} > 0 \end{cases}.$$

By using s_p and δ_j, a nonbasic variable is qualified for dual ratio test if

$$\bar{a}_{pj} = \hat{a}_{pj} \times s_p \times \delta_j > 0,$$

where the eight potential situations is merged into one. The value of s_p is determined after CHUZR, the value of δ_j for each non-`FREE` nonbasic variable is effortlessly maintained throughout the simplex iterations. For a `FREE` nonbasic variable, the value of δ_j is assigned after the updated tableau row $\hat{\boldsymbol{a}}_p$ is computed. As having nonbasic `FREE` variables is a relatively rare situation, the cost of assigning δ_j for each nonbasic `FREE` variables is trivial.

Ratio test. In the ratio test stage, by multiplying each nonbasic reduced cost $\hat{c}_j$ with the corresponding moving indicator δ_j, the dual feasibility conditions of all four bound types is also merged into

$$\bar{c}_j = \hat{c}_j \times \delta_j \geq 0.$$

With the merged situations, the simplest dual ratio test can be achieved by finding the smallest ratio between $\bar{c}_j$ and positive $\bar{a}_{pj}$ as

$$q = \arg\min_j \bar{c}_j/\bar{a}_{pj},\ \bar{a}_{ij} > 0. \tag{3.11}$$

If no such variable can be chosen, then the LP is declared dual unbounded.

Bound flipping ratio test

When a `BOXED` nonbasic variable q_1 is chosen as the entering variable by the regular dual ratio test (3.11), it is possible that variable q_1 may become primal infeasible after entering the basic partition. This will happen if the primal step $\theta_p = \Delta x_p/\bar{a}_{pq_1}$ (where $x_q := x_q + \theta_p$ at the end) computed by $\Delta x_p/\bar{a}_{pq_1}$, is greater than the range $r_{q_1} = u_{q_1} - l_{q_1}$. The condition is more conveniently stated as

$$\Delta x_p > r_{q_1} \times \bar{a}_{pq_1}, \tag{3.12}$$

which can be interpreted as the maximal (feasible) change provided by of variable q_1 not being big enough to remove the infeasibility of the leaving variable.

Though primal infeasibility is allowed in the dual simplex algorithm, there exists a more efficient approach called the *bound flipping ratio test* (BFRT) [19] for dealing with this situation. In the BFRT, the `BOXED` variable q_1 is flipped to its opposite bound rather than entering the basic partition. By this flip, the primal infeasibility is reduced by $\Delta x_p := \Delta x_p - r_{q_1} \times \bar{a}_{pq_1}$, and the nonbasic variable q_2 associated with the second smallest ratio is chosen as the new entering variable. If the new entering variable q_2 again satisfies the BFRT condition (3.12), then the BFRT will flip q_2, reduce Δx_p and continue with the next smallest ratio, until the condition is no longer satisfied after $t-1$ flips, when the BFRT stops and uses q_t as entering variable. Obviously, BFRT will terminate on the first non-`BOXED` variable. Especially, if all the qualified candidates are flipped but the primal infeasibility Δx_p is still positive, then the BFRT will declare the LP dual unbounded.

Performing BFRT is desirable not only because it results in a feasible entering variable, but also, more importantly, it yields a better objective function improvement. With the regular dual ratio test, the objective improvement is given by $\Delta f = (\bar{c}_q/\bar{a}_{pq}) \times \Delta x_p$, where the primal infeasibility defines the per unit improvement (rate) of the objective function when increasing the dual ratio $\theta_d = \bar{c}/\bar{a}_{pq}$. Without BFRT, objective improvement is limited by the smallest θ_d.

With BFRT, the objective function can keep increasing at a smaller rate Δx_p after each bound flip. The rate becomes negative and thus the objective function will start to decrease if variable q_t is also flipped. So that the BFRT stops after $t-1$ flips and uses q_t as the entering variable. The total amount of objective improvement identified by increasing the dual ratio in BFRT can be plotted as a piecewise linear function as shown in Figure 3.1. In the example, the rate given by Δx_p starts to fall after q_4, so that BFRT stops at q_4 with three bound flips identified and q_4 chosen as the entering variable.

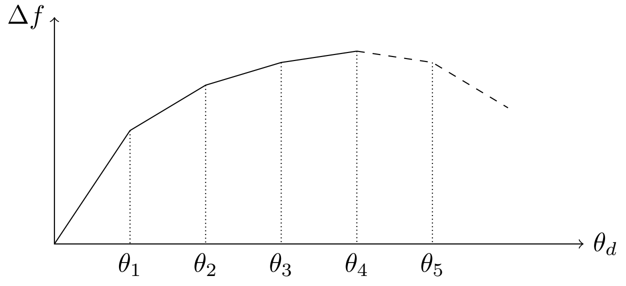


Figure 3.1: Objective improvement associated with BFRT

To further exploit the benefit of BFRT, the bounds of structural variables are tightened by presolve technique. Bound tightening is a special presolve technique which, if implemented carefully, does not require any postsolve. Detailed discussion [24, 25] of bound tightening implementation is beyond the scope of this report and thus is omitted.

A situation which must be addressed in the BFRT is a tie of ratios where many candidates would flip at the same point, resulting a negative Δx_p directly or more quickly. Even worse, if many candidates are tied with the ratio 0, then the Δx_p may become negative right after $\theta_d > 0$, so no flips can be actually identified and no objective improvement can be made. This situation is discussed in more detail in Section 3.3.

Adaption of Harris ratio test

The dual ratio test can be enhanced by the Harris two-pass ratio test [33], which was originally designed for the primal simplex method. When the primal ratios are tied or nearly tied, the Harris ratio test will identify a leaving variable associated with the greatest value of $|\hat{a}_{pq}|$, by explicitly considering and using the feasibility tolerance. This procedure can be adapted in the regular dual ratio test with trivial effort.

For the regular dual ratio test, the Harris ratio test can be implemented by choosing the smallest relaxed dual ratio in the first pass by

$$\theta^r = \min_{j \in \mathcal{N}} (\bar{c}_j + \epsilon_d)/\bar{a}_{pj}, \ \bar{a}_{ij} > 0, \tag{3.13}$$

where ϵ_d is the dual feasibility tolerance, or simply the dual tolerance, by which the dual feasibility condition is relaxed to $\bar{c}_j > -\epsilon_d$. A practical value for ϵ_d is the same as that for ϵ_p, often 10^{-6} or 10^{-7}.

In the second pass, from those candidates whose ratio is smaller than the relaxed ratio θ^r, the Harris ratio test chooses the entering variable which is associated with the greatest magnitude by

$$q = \arg\max_j \{\bar{a}_{pj} | (\bar{c}_j / \bar{a}_{pj}) \leq \theta^r \text{ and } \bar{a}_{ij} > 0\}.$$

A issue with the original Harris ratio test is that it does not handle the situation where a chosen entering variable (leaving variable in primal) is already slightly infeasible, but is feasible within the feasibility tolerance. This issue is addressed by implementing a strict zero step of the EXPAND [22] procedure in the primal simplex method. The same idea for the dual can be implemented by shifting the already infeasible dual reduced cost of the chosen entering variable to zero when it enters the basic partition, and removing the shift when it leaves the basic partition. In the full EXPAND framework, the feasibility tolerance is increased to allow a tiny step with already infeasible choices. However, implementation experience shows that the exact zero step approach is sufficient, especially, when cost perturbation (discussed in the next subsection) is performed before applying the dual simplex method.

The Harris two-pass ratio test can be introduced into the BFRT framework as well. Starting from a qualified candidates set $\mathcal{J} = \{j|\bar{a}_{pj} > 0\}$, the set $\mathcal{F}$ of flipping variables can be identified by repeating a four step BFRT search

1. Find the smallest relaxed ratio θ^r by (3.13) for current candidates set $\mathcal{J}$
2. Identify all the variables $\mathcal{F}'$, whose ratio is smaller than the relaxed ratio θ^r
3. Compute the flip changes $\Delta\mathcal{F}'$ if all the variables of $\mathcal{F}'$ is to be flipped
4. If $\Delta\mathcal{F}' < \Delta x_p$, then set $\Delta x_p := \Delta x_p - \Delta\mathcal{F}'$, $\mathcal{F} := \mathcal{F} + \mathcal{F}'$, $\mathcal{J} := \mathcal{J} - \mathcal{F}'$ and go to step 1. Otherwise BFRT terminates by choosing the biggest $\hat{a}_{pj}$ in $\mathcal{F}'$ as the entering variable.

To accommodate the Harris ratio test, for all variables in the group $\mathcal{F}'$ identified by the relaxed ratio, they are either all flipped, or none flipped with one chosen as the entering variable. A chart illustrating a real BFRT process with Harris ratio test is given in Figure 3.3.

Cost perturbation

Perturbation has been considered as an effective method in resolving ties in the ratio test. With random cost perturbations, it is nearly impossible to have two variables with exactly same reduced cost.

The cost perturbation procedure applied in HSOL is a simplified version that described by Koberstein [38]. Before starting the dual simplex method, a randomly generated small perturbation is applied as follows to the cost of each structural variable (which is also nonbasic with the logical basis starting technique)

$$c_j = \begin{cases} c_j + \xi_j, & \text{if } j \text{ is } \texttt{LOWER} \text{ or } c_j > 0 \text{ when } j \text{ is } \texttt{BOXED} \\ c_j - \xi_j, & \text{if } j \text{ is } \texttt{UPPER} \text{ or } c_j < 0 \text{ when } j \text{ is } \texttt{BOXED} \\ c_j, & \text{otherwise } (j \text{ is } \texttt{FREE} \text{ or } \texttt{FIXED}) \end{cases} \tag{3.14}$$

where ξ_j is a small positive amount computed by

$$\xi_j = (10^{-5}|c_j| + 10^{-7}\max|c_k| + 10\,\epsilon_d) \times (\mathrm{r}+1),\ r \in (0,1]. \tag{3.15}$$

As the formation of ξ_j indicates, the perturbation scale is a combination which considers the cost of variable j, the overall biggest cost ($\max|c_k|$) and the dual tolerance.

After the LP problem is solved to optimality, the perturbation is removed. It is possible for this to result in small dual infeasibilities after the removal of cost perturbation which, in practice, can be easily removed by a simple implementation of the phase II primal simplex method. The phase II primal simplex method can also be applied to remove the cost shift associated with the basic variables.

A further step to the random perturbation, suggested by Koberstein, is to scale the perturbation amount ξ_j by a weight according to the number of non-zero entries of each structural column as shown in Table 3.3. The aim of the weighted perturbation is to give more priority to sparser choices to delay the spread of denser columns. Unfortunately, no further numerical evidence or justification was provided in the original work, nor the expected effect was confirmed by computational experiences during this research. It is possibly because for certain sparse problems, nearly all the columns have the same count of non-zero entries initially, for example the PDS problems. On the other hand, for problems which demonstrated little or no sparsity, there is simply no sparsity to be promoted.

count	1	2	3	4	5	6	7	8	9	≥ 10
weight	0.01	0.1	1	2	5	10	20	30	40	100

Table 3.3: Weighted cost perturbation

However, with the "unweighted" perturbation, the simplex iteration speed was found to increase remarkably for certain families of LP problems. This phenomenon is examined in greater detail in Section 3.3.

3.2.4 Dual phase I method

An advantage of the dual simplex method is that many LP problems are initially dual feasible or can be made so by flipping bounds of corresponding non basic variables, especially when the bound tightening technique is applied. For the remaining LP problems, a dual phase I method is applied to find a starting dual basic feasible solution. Historically, many dedicated dual phase I approaches has been proposed. In this research, the most comprehensive subproblem approach recently summarized by Koberstein and Suhl [39] is applied.

The subproblem approach finds a dual feasible solution by solving a simpler LP problem with modified bounds. Artificial bounds are assigned for each of the variables according to their original bound types as shown in Table 3.4. There are two special arrangements when assigning the phase I bounds. For the `BOXED` variables, it is set to `FIXED` zero bound as it can always be made dual feasible by flipping to the other bound. (This point was not clearly expressed in the original paper.) For the `FREE` variables, larger artificial bounds are used so that a `FREE` variable will tend to leave the nonbasic partition and stay within the basic partition. Note that every variable in the dual phase I is a `BOXED` variable or `FIXED` variable and thus the dual phase I problem is always feasible.

type	original bounds	dual phase 1 bounds
`LOWER`	$[l, +\infty]$	$[\ \ 0, 1]$
`UPPER`	$[-\infty, u]$	$[-1, 0]$
`FIXED`	$[l, u], l = u$	$[\ \ 0, 0]$
`BOXED`	$[l, u], l \neq u$	$[\ \ 0, 0]$
`FREE`	$[-\infty, +\infty]$	$[-1000, 1000]$

Table 3.4: Artificial bounds for dual phase 1 subproblem approach

With the bound modified subproblem, the objective function $f_1 = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$ represents the total dual infeasibility.

When solved to optimality, the objective function $f_1 = 0$ is expected, otherwise, the LP problem is dual infeasible. After dual phase I, if any nonbasic variable has reduced cost $\hat{c}_j = 0$, but $x_j \neq 0$, it will be changed to zero. If the variable j is a `FREE` nonbasic variable, then $\hat{c}_j = 0$ implies it is feasible and will be kept so until its reduced cost changes, if ever.

3.3 Promoting hyper-sparsity

When applying cost perturbation in the dual revised simplex method, it has been observed that, for some LP problems, the iteration speed is much faster because the occurrence of hyper-sparse FTRAN and BTRAN results is increased. This phenomenon, called hyper-sparsity promotion, is studied to reveal the underlying reason and search for alternative approaches.

3.3.1 Observation and motivation

When implementing the dual revised simplex method, it was observed that for certain problems, cost perturbation played a key role in terms of reducing solution time. For example, without perturbation, to solve PDS-20 takes 49913 iterations and 46.2 seconds. When perturbation is activated, solving it takes only 38517 iterations and 7.6 seconds. With perturbation, the iteration count and solution time are reduced by 23% and 84% respectively, and the iteration speed is 4.7 times faster.

This improvement is found to be closely related to the increased occurrence of hyper-sparse BTRAN and FTRAN results in the dual simplex method. Because of the increased occurrence of hyper-sparse results, the BTRAN ($\hat{\boldsymbol{e}}_p^T = \boldsymbol{e}_p^T B^{-1}$) and FTRAN ($\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$) computations in each iteration become cheaper, especially when using the hyper-sparse FTRAN algorithm [32]. Additionally, the PRICE ($\hat{\boldsymbol{a}}_p^T = \hat{\boldsymbol{e}}_p^T A$) and FTRAN-DSE ($\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{e}}_p$) operations which are highly related to the result of BTRAN, also become less expensive. Detailed statistics of the occurrence of hyper-sparse results is given in Table 3.5. Because all of the FTRAN results are sparser than the default hyper-sparse threshold $t = 10\%$, a smaller threshold $t = 2\%$ is used in the statistics.

	Perturbation off	Perturbation on
BTRAN	76.1%	98.4%
PRICE	77.5%	99.3%
FTRAN-DSE	72.1%	95.1%
FTRAN ($t = 2\%$)	78.6%	98.1%

Table 3.5: Occurrence of hyper-sparse results when solve PDS-20 with/without perturbation

This phenomenon can be examined in greater detail by plotting the sectional (every 5000 iterations) percentage of dense (non hyper-sparse) results. The sectional percentage of dense BTRAN results when solving PDS-20 is shown in Figure 3.2.

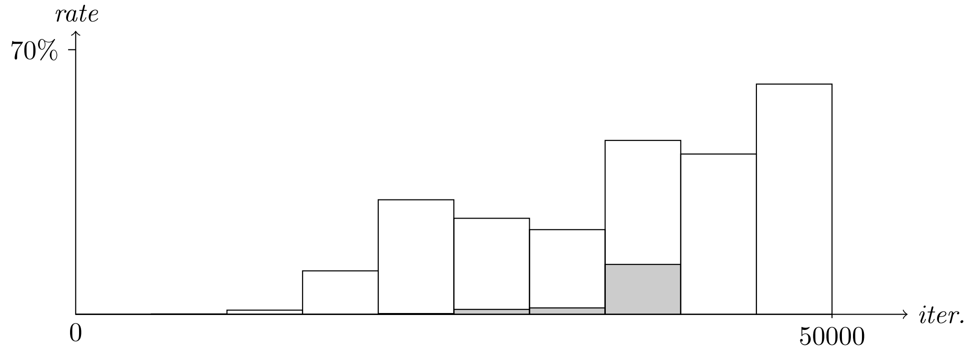


Figure 3.2: Dense BTRAN results statistics when solving PDS-20 by dual revised simplex method with (gray) and without (white) cost perturbation

It is clear that with perturbation (indicated in gray), the total number of dense results is largely reduced, and the occurrence of dense results is delayed. The chart for other three results are similar to those for BTRAN results and thus are omitted.

The phenomenon of increased occurrence of hyper-sparse FTRAN and BTRAN results is called hyper-sparsity promotion. The underlying reason for this phenomenon is analysed in Section 3.3.2, and alternative approaches for promoting hyper-sparsity are discussed in Sections 3.3.3 and 3.3.4.

3.3.2 Algorithmic interpretation of cost perturbation

This section firstly reviews previous discussion on (bound and cost) perturbation for the (primal and dual) simplex method, and then provides an explanation for the hyper-sparsity promotion behaviour of the cost perturbation for the dual simplex method.

Related discussion of the perturbation

The earliest application of perturbation for the simplex method dates back to 1950s–1970s, when the simplex method was still in very active development. In the this period, bound perturbation for the primal simplex method was applied as a degeneracy resolving technique. Typical work of this period includes the "ad hoc" approach of Wolfe [59], where a virtual perturbation is applied when degeneracy is identified, and the work of Benichou [4], which provides an improved perturbation implementation for resolving degeneracy. Chvatal [9] gave a geometric explanation that the bound perturbation splits "every degenerate vertex into a cluster of non degenerate vertices" when radically different amounts of perturbation are applied. Nevertheless, despite its wide application, the discussion of why perturbation works is relatively rare. As summarized by Maros in his book [45], "while the ideas of perturbation and shifting lack some theoretical beauty their practical usefulness is beyond any doubt."

In the 2000s, the dual simplex became much more popular, so that cost perturbation becomes mentioned more within the framework of dual revised simplex method. Bixby [7] claimed that the major performance improvement through early versions of Cplex (from version 5.0 to version 7.1, in the late 1990s and early 2000s) came from the application of "more aggressive perturbation in the dual", especially when solving PDS problems. Bixby also suggested that perturbation be treated "as an algorithmic technique rather than simply a remedy for degeneracy", but unfortunately did not provide any further explanation or detail. The relation between perturbation and sparsity has been once discussed by Koberstein [38], where weighted perturbation is applied to delay the spread of the denser columns in dual CHUZC. However, this claim is neither supported by any numerical results nor got confirmed during this research.

An algorithmic interpretation with reference to BFRT

Within the dual revised simplex method, an explanation of the hyper-sparsity promotion effect of cost perturbation is that (1) it ensures the effectiveness of BFRT, (2) and a successful BFRT reduces the infeasibility and thus the attractiveness of dense rows and (3) so that the occurrence of denser rows in CHUZR is reduced or delayed.

This explanation is initially motivated by a direct observation that when PDS-20 is solved with and without cost perturbation, the number of simplex iterations involving actual bound flips are 10672 and 2286 respectively.

Detailed interpretation can be provided by examining the BFRT behaviour during CHUZC where the BTRAN result ($\hat{\boldsymbol{e}}_p$) associated with the leaving variable is not hyper-sparse. For example, in iteration 26887 when solving PDS-20 with cost perturbation, the BTRAN result $\hat{\boldsymbol{e}}_p$ is 15.7% dense. Note that the leaving variable p is chosen by the DSE framework $p = \arg\max_i \Delta x_i^2/||\hat{\boldsymbol{e}}_i||_2^2$, where the DSE weight w_i is the L2-norm of $\hat{\boldsymbol{e}}_i$. For an LP problem like PDS-20, where the entries of updated rows and columns are often ±1 or of the same magnitude, its DSE weight w_p (8185 in this case) can be viewed as an approximation to the non-zero count (5323) of the logical part of tableau row $\hat{\boldsymbol{e}}_p$. In such a situation, a denser row with a larger w_i can only be chosen when it corresponds to a even larger primal infeasible value (in this situation $\Delta x_p = 1760, \Delta x_p^2 = 3097600$), which assumes more bound flips (10 in this case) in the BFRT.

The BFRT behaviour of iteration 26887 is illustrated in Figure 3.3. With cost perturbation, Harris two-pass ratio test identified 12 potential entering candidates (marked by short vertical lines). These corresponded to the smallest ratios and appeared in 8 groups (separated by dotted lines, representing the relaxed ratios).

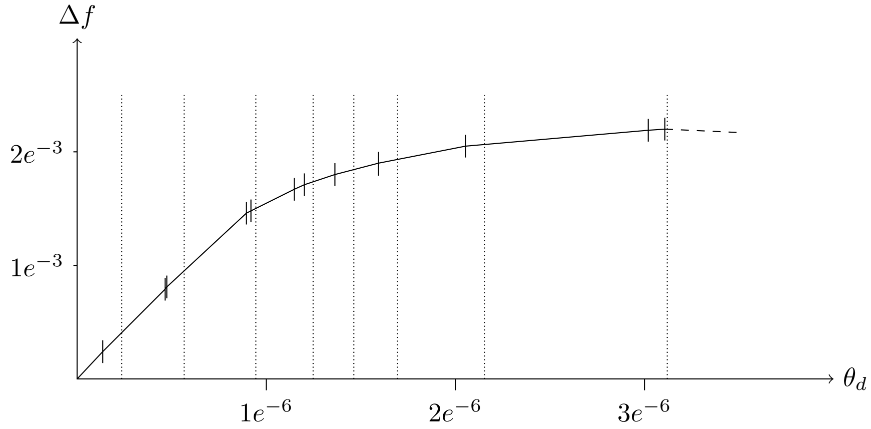


Figure 3.3: Identification of flipping variables by BFRT and Harris ratio test. Totally 8 groups by the Harris two pass ratio test (dotted lines representing the relaxed ratios) and 12 breaking points (marked by short vertical segments)

For a group with more than one variable, to accommodate the Harris ratio test, either (1) all of them are flipped in BFRT or (2) one of them is chosen as the entering variable with the other variables remaining at their current bounds. Therefore, although the turning point (for which, if it is flipped, the objective will fall) is the 12th break point, the 11th break point in the same Harris group is not flipped as well. Therefore, a total number of 10 break points will be identified as flipping variables and the 11th or 12th candidate will be chosen as the entering variable, depending on the magnitude of associated non-zero entry. Although the entering variable will still enter the basic partition as a primal infeasible variable because of the discarded bound flip, its infeasibility is relatively small, and thus it will become much less attractive for the next CHUZR.

The effects of attractiveness reduction for denser rows can be more directly illustrated by plotting the density (of $\hat{\boldsymbol{e}}_i$) of top choices against the relative attractiveness within the DSE framework. The relative attractiveness $\alpha_i^r = \alpha_i/\alpha_p$ is used because the values of real attractiveness are not in the same order. For the iteration concerned (26887) and the next (26888), Figure 3.4 plots the relative attractiveness defined by $\alpha_i^r = \alpha_i/\alpha_p$ of top 20 best CHUZR candidates in these iterations, marked by triangle and square respectively. It can be clearly observed from the figure that in the next iteration, all of the top choices marked by squares are associated with hyper-sparse BTRAN results.

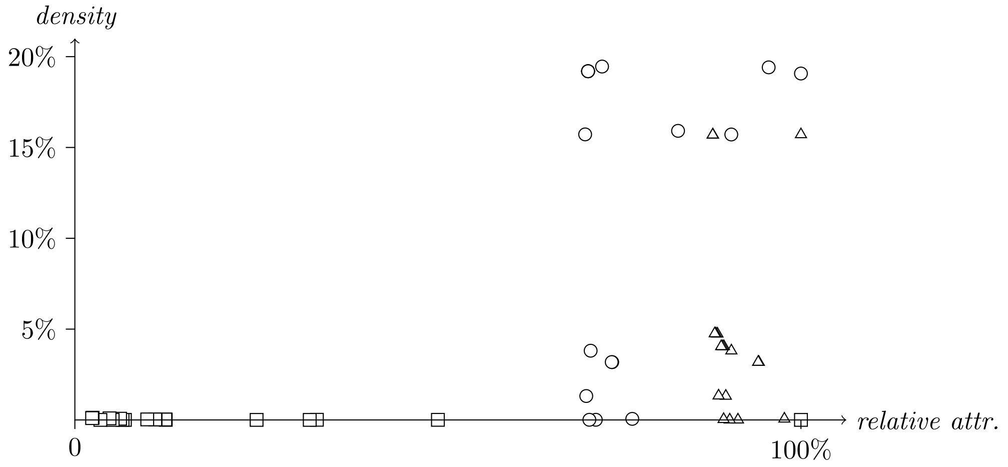


Figure 3.4: The density and top attractiveness. The statistics for the current iteration, the next iteration and the next iteration without permutation are marked by triangles, squares and circles respectively.

On the other hand, if the cost perturbation is removed before the CHUZC of iteration 26887, then all the 461 potential choices, would collapse at the zero ratio. Out of these 461 candidates, the one with biggest non-zero entry will be chosen as the entering variable and no bound flips can be identified. No matter which one is chosen, it will enter the basic partition with a relatively large primal infeasibility and thus it will still be one of the most attractive candidates in the next CHUZR, or in the worst case, *the* most attractive candidate. The attractiveness statistics for the next iteration of this situation are also provided in Figure 3.4, marked by circles, where the it seems become even worse. Of course, it is possible to handle this situation by specially or randomly choosing a set of variables to flip to reduce the primal infeasibility. However, spreading break points as a result of cost perturbation is obviously an elegant way.

One interesting aspect to consider is the scale of the perturbation. It is reasonable to suggest that, when the perturbation scale is extremely small ($\xi \ll \epsilon_d$), then all these potential choices will still be condensed near the zero, so that BFRT cannot be effectively performed. On the other hand, if a slightly larger perturbation is used, the break points may become even better spread, ideally each in one group. This is probably the reason why Bixby [7] suggests using "more aggressive perturbation in the dual" as "an algorithmic technique".

In summary, computational experiences suggests that cost perturbation encourages the BFRT, which reduces the attractiveness of the dense tableau rows and thus reduces the appearance of them in CHUZR, leading to overall hyper-sparsity promotion.

3.3.3 Experiments with partial optimality test

One straightforward alternative approach is partial dual CHUZR. Partial CHUZR in the dual simplex algorithm, or partial PRICE plus partial CHUZC, together called partial pricing for the primal simplex algorithm, is not a novel idea at all. Frequently, partial pricing is applied to reduce the computational cost when there are many qualified choices, where choosing from a small proportion (for example, 10%) of whole candidates is considered enough to identify one of the best. This approach has been very attractive, especially when used within the primal simplex method with the old-fashioned PRICE (3.6), where the matrix-vector product computation is also performed partially. In the dual revised simplex method, especially when hyper-sparse CHUZR is used for sparse LP problems, partial CHUZR is much less preferable than is partial pricing in the primal simplex method.

In relation to promoting hyper-sparsity, experiments with partial CHUZR are motivated by the observations relating to Figure 3.4. As illustrated by the triangles, although the most attractive choice is associated with a dense row, the others in the top ten are not. Thus by using partial CHUZR, it is hoped that, the worst choice is possibly luckily avoided.

In the following experiments, a simple version of partial CHUZR is implemented. Assuming hyper-sparse CHUZR is used, the partial CHUZR starts randomly from the hyper-sparse infeasible list $\mathcal{L}$, and stops after

$$\max(0.1 \times \text{ sizeof } (\mathcal{L}), 100)$$

infeasible variables have been examined. By using this approach, the solution time and iteration count when solving the LP problem PDS-20 is given in Table 3.6.

Experiment	Solution time	Iter. count	Iter. Time (ms)
Textbook	46.2	49913	926
Partial CHUZR	42.5	57903	734
Perturbation	7.6	38517	197
Partial & Perturb	6.8	39369	173

Table 3.6: Performance of HSOL when solve PDS-20 with/without partial CHUZR

It is not surprising that partial CHUZR leads to greater numbers of simplex iterations because, after all, the intention to use partial CHUZR was to avoid the "best" choice under the DSE framework. Despite this, the use of partial CHUZR results in a faster solution because of the increased iteration speed. Note that very little of the performance increase comes from the time saved from performing the full CHUZR, as it was originally considerably cheap because of hyper-sparsity exploitation. Figure 3.5 further illustrates the avoidance of dense rows as a consequence of partial CHUZR (gray colour) in additional to the cost perturbation.

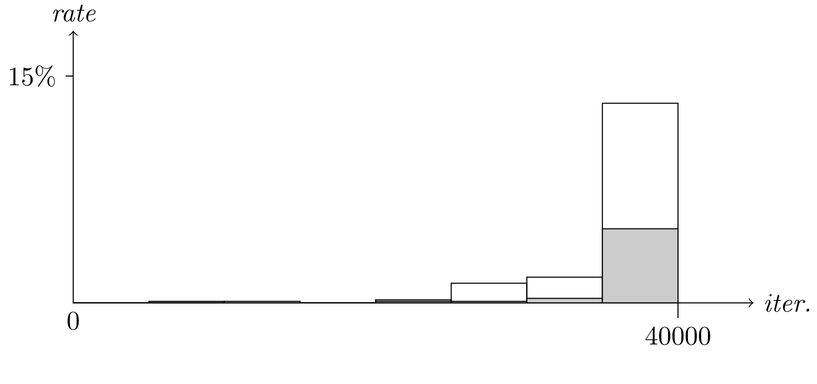


Figure 3.5: Dense BTRAN result when solving PDS-20 using the dual revised simplex method with (gray) and without (white) partial CHUZR.

The problem with partial choose row is obvious in that it does not always choose the best candidate, which may (more or less) compromise the effectiveness of DSE, resulting in a greater number simplex iterations. Also, towards the end of the solution, where the list of primal infeasibilities becomes short, partial CHUZR is disabled so does not help.

3.3.4 Experiments with less-infeasibility DSE

The second experimental approach for promoting hyper-sparsity is called *less-infeasibility* DSE or LiDSE for short. As indicated by its name, LiDSE alters the behaviour of DSE by shrinking the basic infeasibility. This approach is proposed as a response to the observation that within the DSE framework, the dense rows are more attractive than they should be.

Although the DSE weight $w_i = ||\hat{\boldsymbol{e}}_i||_2^2$ for a basic variable associated with a denser row is much larger than that of the sparser rows, its DSE attractiveness is often larger still. This is a result of the formation of the DSE framework

$$\alpha_i = |\Delta x_i|^2/||\hat{\boldsymbol{e}}_i||_2^2,$$

where the attractiveness of the denser rows is enlarged because the infeasibility Δx_i is squared. To address this issue, one idea is to use the absolute value of the infeasibility, so the attractiveness is given by

$$\alpha_i' = |\Delta x_i|/||\hat{\boldsymbol{e}}_i||_2^2.$$

This is called the *less-infeasibility* DSE approach.

Table 3.7 shows the LiDSE attractiveness for the top 20 choices identified (and sorted) by the original DSE for the illustrative iteration when solving PDS-20. Using LiDSE, the primal infeasible candidates for CHUZR associated with hyper-sparse BTRAN results are naturally highlighted.

Density	Infeasibility	L1-norm	DSE weight	DSE attr.	LiDSE attr.
15.71	1760.0	6197	8185	378.4	0.28
0.06	86.0	20	20	369.8	**4.30**
3.18	622.5	1081	1087	356.5	0.58
3.19	622.5	1082	1088	356.2	0.58
0.01	32.2	3	3	345.6	**10.73**
3.81	668.5	1296	1306	342.2	0.52
0.01	32.0	3	3	341.3	**10.67**
1.32	389.0	446	446	339.3	0.87
4.05	699.4	1396	1446	338.3	0.50
0.04	63.7	12	12	338.1	**5.31**
4.05	699.4	1397	1449	337.6	0.50
4.05	699.4	1398	1452	336.9	0.50
1.33	389.0	451	451	335.5	0.86
4.74	733.4	1606	1606	334.9	0.46
4.75	733.0	1608	1608	334.1	0.46
4.76	733.0	1611	1611	333.5	0.45
15.69	1649.0	6188	8176	332.6	0.27
15.69	1649.0	6189	8177	332.5	0.27
15.69	1649.0	6190	8178	332.5	0.27
15.70	1649.0	6191	8179	332.5	0.27

Table 3.7: LiDSE attractiveness for top attractive candidates identified using DSE

The efficiency of LiDSE is again examined by solving PDS-20. As shown in Table 3.8, even without perturbation, the use of LiDSE leads to a halving of the solution time. When used in combination with perturbation, it reduces the solution time by 47%. This performance improvement is partially because of the reduced iteration count due by LiDSE (note that it does not often lead to a shorter iteration). In terms of occurrence of dense BTRAN results, all of them are delayed to appear in the last 5000 iterations.

Experiment	Solution time	Iter. count	Iter. time (ms)
Textbook	46.2	49913	926
LiDSE	17.6	40211	438
Perturbation & DSE	7.6	38517	197
Perturbation & LiDSE	4.0	29475	136

Table 3.8: Performance of HSOL when solve PDS-20 using LiDSE

As theoretical support for the LiDSE framework, the DSE weight w_i can be viewed as an approximation to the L1-norm of $\hat{\mathbf{e}}_i$, the logical part of the tableau row i. As shown in Table 3.7, the difference between the DSE weight (square of L2-norm) and the L1-norm is relatively small. This is particularly so in the case of the sparse candidates where these two values are the same or nearly the same. For these denser candidates, the difference is less than 25%. It happens for PDS problems because the non-zero entries in the updated tableau are often $\pm 1, \pm 2$, with a large proportion of ± 1. Thus the approximation is unsurprising. In this sense, the LiDSE approach using $\alpha_i' = |\Delta x_i| / ||\hat{\mathbf{e}}_i||_2^2$ becomes an approximation of L1-norm based approach.

An issue with the LiDSE approach is that, with more general LP problems where the magnitude of updated tableau entries are often radically different, simply using $|\Delta x_i|$ may severely damage the efficiency of the normalized framework. However, it is not uncommon to have an LP problem exhibiting the same feature as the PDS problems. As they are easily identifiable, the over-wighted approach can always be activated accordingly.

3.3.5 Discussion and additional results

Additional computational results on solving larger PDS problems are given in Table 3.9. It is clear that the LiDSE (plus perturbation) approach outperforms all the other approaches. The performance of LiDSE is even comparable to Cplex (HSOL is 6% slower in average), despite the fact that it takes about 60% more iterations, as shown in Table 3.10. So it is safe to conclude the iteration speed is improved remarkably.

				Perturbation and		
MODEL	Cplex	Plain	Perturb	LiDSE	Partial	LiDSE+Partial
PDS-30	10.1	181.3	16.6	9.0	15.9	9.8
PDS-40	15.6	443.9	27.9	16.3	35.3	21.1
PDS-50	21.6	1102.8	39.8	22.1	54.7	28.7
PDS-60	30.1	1500.6	48.1	30.9	64.0	34.2
PDS-70	35.9	2638.4	64.4	41.9	83.1	42.7
PDS-80	43.0	3396.5	69.5	45.5	87.6	55.2
PDS-90	52.1	7087.2	83.8	74.0	104.1	68.6
PDS-100	56.1	4272.2	83.4	59.2	108.6	72.0
GEOMEAN	28.7	1565.8	47.9	30.9	59.7	35.2

Table 3.9: Solution time of larger PDS problems for hyper-sparsity promotion experiments

Partial CHUZR is found to be unsuitable for larger PDS problems. Although the iteration speed is generally (slightly) improved, the overall performance compared to the pure perturbation approach is poor. Using partial CHUZR in addition to LiDSE also slows it down.

				Perturbation and		
Model	Cplex	Plain	Perturb	LiDSE	Partial	LiDSE+Partial
PDS-30	38019	109748	66278	55855	74989	61136
PDS-40	56126	170271	94528	86065	113500	94217
PDS-50	68769	277235	117370	105291	154614	121695
PDS-60	97089	337941	144285	140166	188056	158702
PDS-70	113677	472672	173681	181504	226000	183177
PDS-80	130016	556205	199042	210103	259465	245280
PDS-90	152364	753988	224664	305091	290819	311842
PDS-100	163076	602172	231179	274495	303622	312270
GEOMEAN	92222	346747	144539	147707	183425	162521

Table 3.10: Iteration count of larger PDS problems for hyper-sparsity promotion experiments

The LiDSE approach can be generalized by using values $|\Delta x_i|^t$ for $1 \leq t \leq 2$. Table 3.11 shows experimental results with the generalized LiDSE (plus perturbation) approach. The column headed $|\Delta x_i|/w_i$ and $|\Delta x_i|^2/w_i$ are the LiDSE and the original DSE approaches respectively. The alternative LiDSE approach with $|\Delta x_i|^{1.2}/w_i$ solves each problem faster than Cplex.

Model	Cplex	$\lvert\Delta x_i\rvert/w_i$	$\lvert\Delta x_i\rvert^{1.2}/w_i$	$\lvert\Delta x_i\rvert^{1.5}/w_i$	$\lvert\Delta x_i\rvert^2/w_i$
PDS-30	10.1	9.0	7.8	10.5	16.6
PDS-40	15.6	16.3	14.5	19.7	27.9
PDS-50	21.6	22.1	19.1	29.4	39.8
PDS-60	30.1	30.9	25.6	35.4	48.1
PDS-70	35.9	41.9	32.3	54.3	64.4
PDS-80	43.0	45.5	35.8	56.4	69.5
PDS-90	52.1	74.0	42.0	62.7	83.8
PDS-100	56.1	59.2	50.3	51.6	83.4
GEOMEAN	28.7	30.9	24.6	34.8	47.9

Table 3.11: Performance of generalized LiDSE approach when solve PDS problems

Although the LiDSE approach is very helpful for the PDS problems, its scope for further application is fairly limited. This is partially because the L1-norm approximation assumption only holds for a limited selection of LP problems. Moreover, for lots of hyper-sparse problems, FTRAN and BTRAN are always hyper-sparse and thus additional promotion is not really required. For problems which demonstrate little or no hyper-sparsity otherwise, the techniques of cost perturbation, partial CHUZR and LiDSE are not advantageous: there is simply no hyper-sparsity to be promoted.

3.4 Summary

This chapter has described a sophisticated implementation of the advanced dual simplex method and an investigation into the hyper-sparsity promotion phenomenon. The performance of the achieved dual simplex solver, HSOL is comparable with Clp as demonstrated by Figure 3.6. This provides a solid foundation for the parallelization.

The hyper-sparsity promotion phenomenon is initially identified when using cost perturbation as an enhancement to the dual ratio rest. It has been demonstrated with reference to PDS problems that cost perturbation encourages the BFRT, which reduces the attractiveness of the dense tableau rows and thus reduces the appearance of them in CHUZR, leading to overall

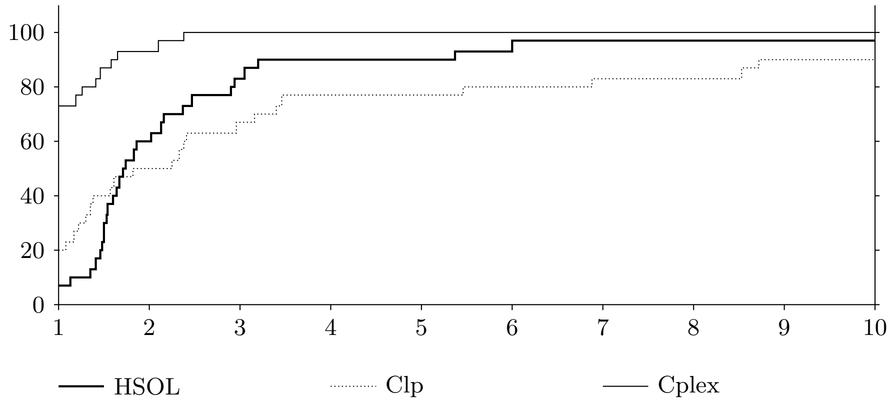


Figure 3.6: Timing profile of HSOL, Clp and Cplex

hyper-sparsity promotion. By following this explanation, experiments have been performed using two alternative approaches for promoting hyper-sparsity and the LiDSE approach has been found to be very successful for one class of problems.

Chapter 4

Parallel simplex methods

This chapter discusses parallel simplex implementations. Section 4.1 provides a brief review of previous simplex parallelization attempts. Section 4.2 analyses previous work and discusses the scope for a practical simplex parallelization. Then two novel parallel simplex implementations based on the dual revised simplex method are introduced: one by exploiting parallelism across multiple iterations (PAMI), one by exploiting single iteration parallelism (SIP), in Sections 4.3 and 4.4 respectively. Overall, PAMI achieves an average speedup of 1.5 and SIP is frequently complementary to PAMI when PAMI results in slowdown. Preliminary design and computational results of PAMI have been published as a conference paper [26].

4.1 Previous simplex parallelization attempts

The simplex method has been parallelized many times on various hardware and software platforms. To facilitate the introduction of novel parallelization implementations, a brief review and analysis of previous simplex parallelization attempts is provided in this and the next section (Section 4.2) respectively. The selection of implementations reviewed in this section is based on that of a more comprehensive report by Hall [28].

Previous parallel simplex implementations can be organized into three categories: (1) those using dense matrix algebra, (2) those using sparse matrix algebra, (3) and those using special simplex method variants.

4.1.1 Using dense matrix algebra

Parallelizing the simplex method by using dense matrix algebra refers to parallelizing the tableau simplex method in dense or the revised simplex method with dense basis inverse. Table 4.1 provides a representative list of parallel simplex implementations using dense matrix algebra.

Generally speaking, parallel simplex implementations with dense matrix algebra achieved excellent speedup, ranging from ten [54, 11] to thousand [56]. One most important reason for this success is that the major underlying operation is easily parallelizable and known to be scalable. For the tableau simplex method, the main operation is the elimination of the whole tableau by using the pivotal row (or alternatively using the pivotal column). For the revised simplex method with dense basis inverse, solving linear systems can be easily arranged as parallel matrix vector multiplication.

Date	Author	Speedup, approaches, and comments
1988	Stunkel and Reed [54]	Speedup between 8 and 12 on 16 processor Intel hyper cube. Textbook tableau simplex method.
1991	Cvetanovic et al. [11]	Best speedup is 12 on a 16 processor shared memory machine, when experimented with 2 large `NETLIB` LP problems. Textbook tableau simplex method.
1995	Eckstein et al. [14]	Iteration speed is superior to Minos 5.4 on some `NETLIB` LP. Implemented on Connection Machine, CM-2 and CM-5 with thousands of processors. Both tableau simplex method and revised simplex method with dense explicit inverse, including steepest-edge algorithm and EXPAND techniques.
1996	Thomadakis and Liu [56]	Up to 1000 speedup on 128 × 128 cores MasPar machine. Tableau simplex method with steepest-edge algorithm. Random LP.
2006	Badr et al. [2]	Up to 5 speedup on 8 processors SMP. Developed using C and MPI. Textbook tableau simplex method. Random small dense LP.
2009	Yarmish and Slyke [61]	Nearly 7 times faster with 7 processors, in terms of iteration speed. Textbook tableau simplex method. Iteration speed is comparable to Minos when solving dense `NETLIB` LP problems.
2009	Spampinato and Elster [53]	Speedup between 2 and 2.5 on GPU platform compared to corresponding CPU implementation. Textbook revised simplex method with dense PF representation of the basis inverse. Random LP.
2010	Bieling et al. [5]	Up to 18 times faster than the sequential GLPK solver. Revised simplex method with dense PF basis inverse and steepest-edge algorithm. Random LP.
2011	Lalami et al. [40]	Up to 24.5 speedup with two GPUs. Random LP.

Table 4.1: Simplex parallelization using dense matrix algebra

However, besides the excellent speedup, there are also crucial limitations with dense matrix algebra. LP problems are generally known to be large and sparse, and exploiting the sparsity properly is a key factor to the performance of the simplex algorithm. By using dense matrix algebra, exploiting the sparsity becomes impractical or even impossible. Thus, even though a dense parallel implementation achieves good speedup, it can hardly be comparable to a good sequential sparsity exploiting simplex solver.

Due to this limitation, although reports [2, 61] of parallel simplex method using dense matrix algebra continue to appear, often corresponding to novel hardware platform and software techniques (most recently, GPU and GPGPU [53, 5, 40]), none is of real novelty.

4.1.2 Using sparse matrix algebra

The real challenge and most valuable results of the simplex method parallelization are those using sparse matrix algebra. Compared to parallel implementations using dense algebra, previous work concerning sparse algebra is relatively rare. A comprehensive list is given in Table 4.2.

Previous work on parallel simplex method using sparse matrix algebra can be organized

Date	Author	Speedup, approaches, and comments
1995	Lentini et al. [42]	General speedup is between 0.5 and 2.7 when solving medium sized NETLIB problems. Best speedup is 5.2 when solving a large column-row ratio problem. Tableau simplex method in sparse.
1976	Pfefferkorn and Tomlin [50]	Discussed the parallelization scope of each revised simplex method computational component, including FTRAN and BTRAN.
1988	Helgason et al. [34]	Discussion of the parallelization scope of FTRAN and BTRAN.
1997	McKinnon and Plab [47]	Advanced investigation of data parallelism of FTRAN operation only.
1994	Ho and Sundarraj [35]	Average 1.5 speedup. Partial task parallelism by overlapping INVERT and other simplex iterations.
1995	Shu [52]	Up to 17 (8) speedup when solving small NETLIB (large and sparse Kennington set) LP problems. Full data parallelism.
1996	Wunderling [60]	No known speedup data. Full task parallelism by multiple choosing column and forming several tableau columns (FTRAN operations) in parallel.
1996	Hall and McKinnon [30]	Speedup between 1.7 and 1.9. Sophisticated full task parallelism. Up to 2.4 faster per iteration.
1998	Hall and McKinnon [31]	Speedup between 2.4 and 4.8. Asynchronous full task parallelism. Up to 5 faster per iteration.
2000	Bixby and Martin [8]	Between 1 and 3 speedup. Dual revised simplex method. Partial data parallelism on PRICE and related operations, with discussion of possible task parallelization of FTRAN and FTRAN-DSE. Based on commerical simplex solver Cplex.

Table 4.2: Simplex parallelization using sparse matrix algebra

into four groups as shown in the table: (1) the tableau simplex method parallelization using sparse algebra, (2) theoretical discussion of parallelism in FTRAN and BTRAN operations, (3) implementation of parallel primal revised simplex method and (4) implementation of parallel dual revised simplex method.

Parallelizing tableau simplex method in sparse

In the first group, the unique work of Lentini [42] is so far the only attempt to parallelize the tableau simplex method using sparse algebra. It was motivated by the observation that for certain families of practical LP problems, for example the hyper-sparse LP problems, the tableau is so sparse that maintaining each tableau column in a sparse data structure would reduce elimination work and storage requirement considerably. Though speedup of 2 is achieved, because of the limitation of the tableau simplex method, it can hardly be comparable to a revised simplex solver.

Theoretical data parallelism discussion of FTRAN and BTRAN

The second group consists of discussions of the data parallelization scope of mainly FTRAN and BTRAN operations. The early work by Pfefferkorn and Tomlin [50] provided a comprehensive discussion on each computational components of the simplex method, including the FTRAN and BTRAN operations. Later, the work of Helgason et al. [34] focused on FTRAN and BTRAN only. The work by McKinnon and Plab [47] concentrated on the FTRAN operations only, as the BTRAN operation is more efficiently achieved by using row-wise basis inverse representation with FTRAN procedure. None of these discussions reported computational results, probably because the FTRAN operation had a large number of relative cheap sparse vector additions that can hardly overlap with each other, so the performance gained by parallelization is less than the overhead associated with it.

Parallelizing the primal revised simplex method

The third group provides a list of published parallel primal revised simplex method implementations. They all appeared in a short time, when the sequential simplex method had (then) been considered developed into a mature status, and parallel machines became widely used.

Except for the relative explicit data parallelization scope of CHUZC, CHUZR, PRICE and UPDATE, the major focus of these work is exploiting parallelism further in other nontrivial computational components, mainly FTRAN, BTRAN and INVERT.

Out of these five implementations, the work of Shu [52] is distinctive because it explored data parallelism fully on FTRAN and BTRAN by maintaining an explicit sparse basis inverse. Good speedup (up to 17) was reported on NETLIB problems and larger and sparser problems from the Kennington set. However, as Hall indicated [28], the good speedup is related to a large percentage ($65-96\%$) attributable to an old-fashioned and inefficient column-wise (but easily parallelizable and scalable) PRICE component, where the reduced cost is computed rather than updated. When modern PRICE is applied, it would be a lower percentage of overall time, so the scope for good speedup may no longer exist.

Besides the work of Shu, all of the other four implementations exploited task parallelism.

The work of Ho and Sundarraj [35] overlaps the INVERT with simplex iterations, and switches to the newly formed basis inverse once it is ready, resulting in 1.5 speedup on average. The same idea had also been experimented with by Bixby and Schwabacher, with a speedup of 1.3 as described in an unpublished report [6]. In both of these overlapping schemes, the PF update *eta* matrices obtained with old basis inverse $\dot{B}_0^{-1}$ are reused to bring the newly formed basis inverse B_0^{-1} up-to-date by

$$B_t = B_0\dot{E}_{k+1}\dots\dot{E}_{k+t} \text{ and } B_t^{-1} = \dot{E}_{k+t}^{-1}\dots\dot{E}_{k+1}^{-1}B_0^{-1},$$

where $\dot{E}_{k+1},\dots,\dot{E}_{k+t}$ are last t PF update transformations associated with $\dot{B}_{k+t}$, assuming that the reinversion started when working with $\dot{B}_k$. Though relative good speedup is achieved by only parallelizing one component, as identified by Hall and McKinnon [29], the reuse of PF update often leads to numerical instability.

The work of Wunderling [60] explores parallelism fully with a variant of the primal simplex method. The variant can be described in a major-minor structure. It starts from a major initialisation, by performing multiple CHUZC to choose a set $\mathcal{Q}$ of entering variable candidates and then, using task parallelism, forming tableau columns $\hat{\boldsymbol{a}}_q$ for each $q \in \mathcal{Q}$ accordingly. Then, in each of the subsequent minor iterations, a variable q is chosen and removed from $\mathcal{Q}$ as the entering variable, until $\mathcal{Q}$ becomes empty or non of $q \in \mathcal{Q}$ remains attractive. The core component of minor iterations is primal CHUZR, where data parallelism is exploited. After the minor iterations, a major update is performed to update the reduced cost and basis inverse representation, where the key components PRICE and PRICE-SE are performed as data parallel operations Though an interesting approach, it is unfortunate that the only associated publication is his PhD thesis written in German. A little further detail of Wunderling's work is revealed by Hall [28], that generally it achieves little performance improvement. The few instances showing good speedup, correspond to large column-row ratios, where it is easy to gain performance improvement on parallel PRICE and PRICE-SE. The same idea is exploited with the dual revised simplex method during this research. Details of a similar major-minor structure are introduced in Section 4.3.

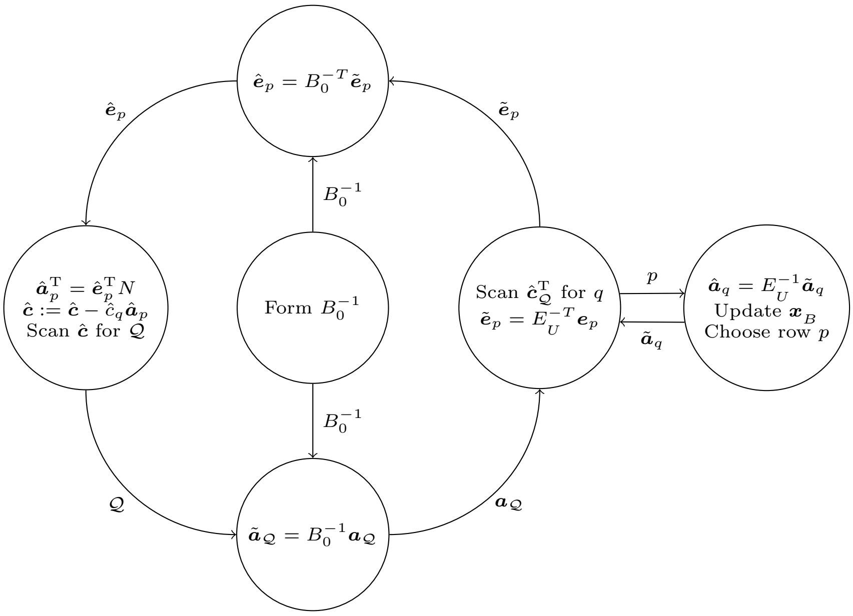

$\hat{\boldsymbol{e}}_p = B_0^{-T}\tilde{\boldsymbol{e}}_p$
$\hat{\boldsymbol{e}}_p$
$\tilde{\boldsymbol{e}}_p$
B_0^{-1}
$\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} N$
$\hat{\boldsymbol{c}} := \hat{\boldsymbol{c}} - \hat{c}_q\hat{\boldsymbol{a}}_p$
Scan $\hat{\boldsymbol{c}}$ for $\mathcal{Q}$
Form B_0^{-1}
Scan $\hat{\boldsymbol{c}}_{\mathcal{Q}}^{\mathrm{T}}$ for q
$\tilde{\boldsymbol{e}}_p = E_U^{-T}\boldsymbol{e}_p$
p
$\tilde{\boldsymbol{a}}_q$
$\hat{\boldsymbol{a}}_q = E_U^{-1}\tilde{\boldsymbol{a}}_q$
Update $\boldsymbol{x}_B$
Choose row p
B_0^{-1}
$\mathcal{Q}$
$\boldsymbol{a}_{\mathcal{Q}}$
$\tilde{\boldsymbol{a}}_{\mathcal{Q}} = B_0^{-1}\boldsymbol{a}_{\mathcal{Q}}$

Figure 4.1: Parallelization scheme of PARSMI

Besides this work, the author knows of only two other full parallel implementations of the the revised simplex method, are all due to Hall and McKinnon. These two implementations, the preliminary version ASYNPLEX [31] and its mature development PARSMI [30], are all based on an asynchronous variant of the revised simplex method. The logic of their simplex variant is illustrated by Figure 4.1. The design and implementation of PARSMI is a combination of overlapped INVERT and multiple CHUZC. Although compared to the speedup of order ten obtained when using dense matrix algebra, PARSMI achieves only a speedup of between 1.7 and 1.9. This was a great achievement because PARSMI is based on a sophisticated implementation of the revised simplex method.

Parallelizing the dual revised simplex method

The work of Bixby and Martin [8] is, so far as the author knows, the only publication on the parallel dual revised simplex method for general LP problems. Although only data parallelism on PRICE and UPDATE-DUAL is explored, their work is extremely attractive because it was based on and compared to the one of best commercial simplex solvers Cplex.

Besides data parallel PRICE and UPDATE-DUAL the possibility of a task parallel arrangement of FTRAN and FTRAN-DSE was also discussed. However, this potential task parallelization was not included in their implementation, because its performance was found to be inferior to solving two linear systems together sequentially.

However, the argument of the inferior performance may no longer hold because of the exploitation of hyper-sparsity. By exploiting hyper-sparsity, FTRAN and FTRAN-DSE are often achieved totally differently, where solving two linear systems together may severely slow down one of them (often the FTRAN). Besides, with the novel dual revised simplex method, there is a third forward linear system to solve as a consequence of the bound flipping ratio test (BFRT). By considering FTRAN operations with the novel hyper-sparsity and BFRT techniques, exploiting task parallelism may become valuable again. This is discussed in detail in Section 4.4.

4.1.3 Other approaches

Besides parallelizing the tableau and revised simplex method or its variant in relatively regular ways, there are other interesting approaches to exploit parallelism when using simplex method for solving LP problems.

Maros and Mitra [46] provided a strategy which solves an LP problem with different processors. Each processor follows a different path, given by different pivot selection approaches, including Devex, steepest edge algorithm and some partial pricing variants. Although Devex is generally considered as the best choice for the primal simplex method, it is not always the best for a single LP problem. By following different paths, an LP problem is naturally solved by the best pivoting strategy. In more sophisticated implementation, this approach can be further enhanced by periodically selecting the best progress, and then resetting all the others processors to run from the best point reached. By using the mixed strategy, a modest speedup of around 1.6 is achieved.

Another distinct parallelization approach related to one of the earliest simplex method variants uses the Dantzig-Wolfe decomposition [12] for block-angular problems with linking rows as illustrated in Figure 4.2. An LP problem in such form can be solved in a decompose-solve-join-solve manner: the original problem is firstly decomposed into many smaller independent LP subproblems and solved to optimality separately (in parallel), and afterwards, optimal solutions of subproblems are joined together to solve the original problem. For block-angular problems with linking columns, Benders decomposition [3] may be used. For large and sparse LP problems which are not originally in block-angular form, it may be permuted into such form by hyper-graph partitioning [1]. Due to its good performance and (relative) simplicity, this approach has been implemented many times.

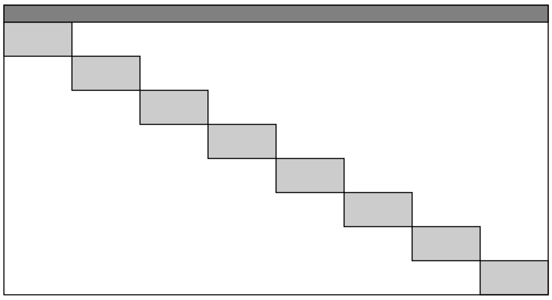


Figure 4.2: LP coefficient matrix in block angular form with linking rows on the top

A recent development for solving LP problems in the block-angular form is reported by Lubin et al. [43]. Instead of using the decompose-solve-join-solve approach, the dual revised simplex method itself is applied directly to block-angular problems with linking columns, with its operations parallelized according to the special structure. A speedup of more than 100 with 256 processors is achieved compared with the sequential Clp solver. Though extremely efficient, the application and scalability of this approach is highly limited to a specific family of LP problems, which either appear in (or can be easily permuted to) block-angular form.

4.2 Limitation and scope of simplex parallelization

This section analyses the relative poor speedup of previous simplex parallelization implementations and, based on this, discusses the scope for future practical parallel simplex implementations.

4.2.1 Analysis of previous work

Previous research into parallelizing the simplex method provides much valuable experience for future simplex parallelization.

Using dense matrix algebra

If dense matrix algebra is used, even a simple implementation yields good speedup (between ten and a thousand). However, in the real world, exploiting sparsity and hyper-sparsity in the revised simplex method keeps enlarging the gap between the performance of it and the tableau simplex method or revised simplex method using dense matrix algebra. Although it has been demonstrated many times that simplex parallelization using dense matrix algebra achieves good speedup, even with the best speedup, it can hardly be comparable to a revised simplex implementation with modern computational enhancements for general LP problems. Therefore, this path is excluded from future consideration.

Using sparse matrix algebra

The simplex parallelization based on the revised simplex method, using sparse matrix algebra, has been considered relatively little and very much less successfully. The relatively poor speedup of these parallelizations can be analysed with reference to Amdahl's Law and detailed profiling data of major computational components of the revised simplex method.

Amdahl's Law states that the speedup of parallelizing an algorithm is limited by the percentage of the sequential (non-parallelized) proportion (p_{SEQ}) of that code by

$$\text{speedup } = \frac{1}{p_{SEQ} + (1 - p_{SEQ})/N}$$

where N is number of cores used. For example, when $p_{SEQ} = 50\%$, no matter how many cores are used, the best speedup is limited to 2. An illustration of Amdahl's Law is given in Figure 4.3.

The profiling data of computational components of the advanced dual revised simplex method is provided in Table 4.3, including the iteration time (in microseconds) and the percent-

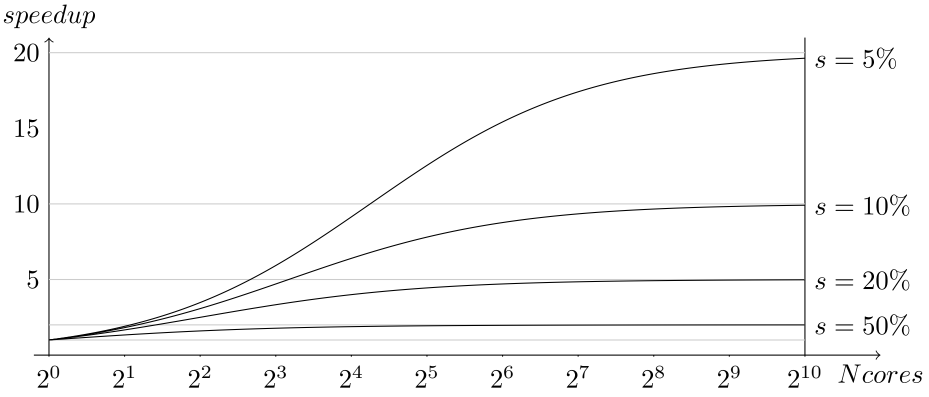


Figure 4.3: Illustration of Amdahl's Law

ages (of overall solution time) of major computational components, organized in four groups (vertically), roughly according to ascending difficulty of exploiting parallelization.

With reference to Amdahl's Law and profiling data, the relatively poor speedup of the revised simplex method parallelization can be explained from three aspects.

1. Limited parallelization. One reason for the poor speedup is the partial exploitation of parallelism. A typical example is the dual revised simplex method parallelization of Bixby and Martin [8]. In their work, only standard parallelizable computational components, namely CHUZC, CHUZR, PRICE and UPDATE are parallelized. According to the profiling data, the average percentage of these components are less than 35%, so that by Amdahl's Law, the best possible speedup is limited to 1.5 on average.

2. Inferior pivoting rule and memory bound. Even when exploiting parallelism fully by arranging FTRAN and BTRAN as task parallel operations, as reported by Hall and McKinnon [30], the achieved speedup is still less than or around 2. This is partially because the simplex variant using multiple pricing, which PARSMI was based on, frequently results in inferior pivots and leads to a greater number of simplex iterations; partially because when many FTRAN (or BTRAN) operations are performed in parallel, the competition for memory access will highly limit the parallelization efficiency.

3. Cheap computational components. Last but not least, another reason for the poor speedup is the wide availability of cheap computational components in the revised simplex method. Although it may take up to a few hours to solve an LP problem, the iteration time of the simplex method is often around hundreds of microseconds. Thus, if the overall percentage of a component is relatively small, then it only takes about tens of microseconds or even less per iteration, which is comparable to (or even smaller than) the overhead of starting a parallel environment and spreading data across parallel processors. In this case, either going parallel or leaving the components performed sequentially will not actually improve the performance. Therefore, the cheap component essentially adds up to the sequential proportion, which further limits the speedup.

4.2.2 Towards a practical simplex parallelization

Because of these limitations, the simplex method is frequently considered unsuitable for parallelization.

However, the context for simplex parallelization has changed. Since the introduction of

		Group 1			Group 2		Group 3				Group 4	
MODEL	Iter. Time	CHUZR	CHUZC1	CHUZC2	PRICE	UPDATE	BTRAN	FTRAN	F-DSE	F-BFRT	INVERT	OTHER
CRE-B	565	0.8	20.1	4.4	42.9	6.9	4.7	1.7	11.3	1.5	4.3	1.4
DANO3MIP_LP	885	1.8	21.2	3.0	35.5	5.3	6.4	6.9	11.7	0.3	6.2	1.7
DBIC1	2209	0.5	22.5	3.1	33.6	5.8	5.7	6.5	14.8	3.2	3.1	1.2
DCP2	509	6.5	3.9	1.7	8.7	7.3	5.4	18.1	28.4	10.4	7.4	2.2
DFL001	595	4.1	8.1	1.0	17.9	11.2	10.8	13.0	20.7	6.2	5.2	1.8
FOME12	971	7.9	5.1	0.6	12.4	6.8	12.3	14.5	24.0	7.1	7.9	1.4
FOME13	1225	10.1	4.2	0.5	10.6	5.6	11.4	13.5	26.4	6.7	9.6	1.4
KEN-18	126	5.3	2.9	0.6	5.2	2.2	7.9	11.0	24.4	3.8	32.4	4.3
L30	1081	0.8	14.1	9.9	24.0	6.3	8.6	9.0	12.9	4.1	8.5	1.8
LINF_520C	26168	1.5	2.3	0.1	11.8	4.0	16.6	19.7	23.2	0.0	19.2	1.6
LP22	888	2.0	10.9	2.0	23.3	8.4	9.4	10.4	14.9	6.8	10.0	1.9
MAROS-R7	1890	0.8	2.8	0.2	10.2	2.7	17.5	15.3	20.6	0.0	27.4	2.5
MOD2	1214	4.2	7.5	1.0	9.9	8.5	11.5	17.4	29.1	5.4	4.0	1.5
NS1688926	1806	2.0	0.1	0.0	2.9	4.8	3.3	31.4	44.1	0.0	6.5	4.9
NUG12	1157	1.6	7.4	1.1	16.3	6.9	11.6	12.4	16.7	5.8	18.1	2.1
PDS-40	302	3.4	7.5	1.9	19.2	5.1	10.8	10.3	23.2	4.4	12.0	2.2
PDS-80	337	3.7	6.6	1.8	19.8	3.9	10.5	9.1	23.7	3.9	15.0	2.0
PDS-100	360	3.5	7.0	1.8	18.6	3.7	10.4	9.0	24.1	3.8	16.0	2.1
PILOT87	918	1.2	5.1	0.8	17.9	4.4	12.0	12.9	17.4	7.6	17.9	2.8
QAP12	1229	1.5	7.5	1.0	16.2	6.6	12.1	12.3	16.7	5.9	18.4	1.8
SELF	8350	0.0	1.4	0.2	39.6	0.2	7.0	6.5	7.0	0.0	33.9	4.2
SGPF5Y6	491	1.3	0.3	0.1	0.2	0.1	5.0	2.3	80.7	0.0	8.4	1.6
STAT96V4	2160	0.4	12.4	4.9	67.6	1.7	2.4	1.7	4.3	0.6	2.2	1.8
STORMG2-125	115	5.2	0.8	0.2	1.7	0.9	4.4	8.3	48.7	0.1	26.7	3.0
STORMG2-1000	650	1.5	0.1	0.0	0.3	1.3	3.5	6.1	70.6	0.0	14.6	2.0
STP3D	4325	1.6	10.7	0.9	19.2	7.6	13.5	12.0	27.0	3.9	2.4	1.2
TRUSS	415	1.1	17.1	2.0	53.8	5.0	5.0	3.7	7.1	0.0	3.5	1.7
WATSON_1	210	4.3	0.7	0.2	1.0	1.2	5.7	6.0	54.4	3.5	19.6	3.4
WATSON_2	161	5.5	0.3	0.0	0.4	0.8	4.6	7.7	35.2	5.0	34.5	6.0
WORLD	1383	3.8	8.7	1.3	10.9	8.6	11.6	16.5	28.0	5.5	3.7	1.4
AVERAGE	867	2.9	7.3	1.5	18.4	4.8	8.7	10.8	26.4	3.5	13.3	2.3

Table 4.3: Iteration time (ms) and computational components profiling (the percentages of overall solution time) when solving LP problems with an advanced dual revised simplex method implementation

DSE and BFRT in the 1990s, the preferred simplex variant to use has changed from the primal simplex algorithm to the dual, but the only published work on the dual simplex parallelization is due to Bixby and Martin [8]. Although it appeared in the early 2000s, their dual simplex parallelization neither included the BFRT nor the hyper-sparse linear system solution technique. In terms of the application scope, in the past (the 1990s), parallelization was aimed at dedicated high performance computers to achieve the best performance; nowadays, when every desktop computer is a multi-core machine, any speedup is desirable in terms of solution time reduction for daily usage. Therefore, the simplex method, especially, the dual revised simplex method deserves renewed parallelization effort.

Besides the impact on the theoretical speedup, the limitations also suggest a practical number of parallel processors to use for revised simplex parallelization. Assuming the sequential (non parallelizable) part of the dual simplex method is 30% when solving general LP problems, and that other parts can be perfectly parallelized, then by using 8, 16, 32 and 64 cores, the best possible speedup are 2.6, 2.9, 3.1 and 3.2 respectively. When the overhead of using many cores is considered, the better expected speedup may never be achieved. Therefore, in this research, a choice of 8 cores is assumed.

4.3 Exploiting parallelism across multiple iterations

This section introduces a dual simplex variant based on multiple CHUZR and how to exploit parallelism across multiple iterations (PAMI) with this variant.

4.3.1 Dual simplex variant with multiple CHUZR

When solving LP problems with the dual revised simplex method, it is observed that for many problems, especially sparse ones, the second-best s pivot candidates identified by the DSE framework at iteration k, are often still among the most attractive choices in the next few iterations $k+t, t \geq 1$. This motivates a dual simplex variant which chooses a small set $\mathcal{P}$ of attractive candidates, and keeps choosing pivotal rows from it, yielding task scope for parallelization and allowing parallelism to be exploited across multiple iterations (PAMI). Specifically, this dual simplex variant and the potential parallelism associated with each computational component, can be organized as a major-minor framework.

1. Major initialization:
   (a) Choose up to s best choices, yielding the set $\mathcal{P}$ (Data parallelism)
   (b) Multiple BTRAN: form $\hat{\boldsymbol{e}}_p = B^{-T}\boldsymbol{e}_p$ for each $p \in \mathcal{P}$ (Task parallelism)
2. Minor iterations:
   (a) Minor choose row: find most attractive $p' \in \mathcal{P}$ or goto major update (Trivial)
   (b) Minor choose column: form $\hat{\boldsymbol{a}}_{p'}^{\mathrm{T}} = \hat{\boldsymbol{e}}_{p'}^{\mathrm{T}} A$ and perform ratio test (Data parallelism)
   (c) Minor update: update the dual variables $\hat{\boldsymbol{c}}$ (Data parallelism)
   (d) Minor update: remove p' from $\mathcal{P}$ and update $\hat{\boldsymbol{e}}_p$ for $p \in \mathcal{P}$ (Data parallelism)
3. Major update:
   (a) Perform FTRAN, FTRAN-DSE, FTRAN-BFRT for chosen pivots (Task parallelism)
   (b) Update primal variables and DSE weights (Data parallelism)

Choosing multiple candidates in the dual simplex method is not a novel idea, the earliest related work is reported by Rosander [51] in 1970s. In this pioneering but less known work, the (then) novel multiple PRICE technique [49] for the primal simplex method was transplanted into the dual framework and called dual multiple PRICE. In the (primal and dual) multiple PRICE framework, performing minor iterations is called *suboptimization.* There are fundamental differences between suboptimization and the simplex variant for PAMI. Suboptimization was aiming at resolving degeneracy to achieve better objective increment per step, where it was mainly regarded as a pivoting scheme. After the appearance of suboptimization, the Devex [33] and steepest-edge [23] algorithm became the best accepted pivoting scheme, and idea of suboptimization faded away. In this research, the simplex variant with multiple CHUZR it is purely utilised to provide more task parallelism scope for solving linear systems. The simplex variant and the PAMI based on it is more related to the work of Wunderling [60] and Hall and McKinnon [30] for the primal simplex method in 1990s.

Details of PAMI are introduced in the remaining parts of this section. The design and implementation of data parallel CHUZC and PRICE and task parallel BTRAN and FTRAN operations are given in Sections 4.3.2 and 4.3.3 respectively. The relatively trivial operations, for example UPDATE-DUAL, UPDATE-PRIMAL and UPDATE-WEIGHT are mentioned when corresponding nontrivial computational components are discussed. Section 4.3.4 talks about consideration for INVERT and the update of the basis inverse. Section 4.3.5 discusses both major and minor CHUZR in PAMI and experiments with the candidate attractiveness. Computational results and illustration of real time behaviour of PAMI is provided in Section 4.3.6.

In the subsequent sections, $s = 4$ is used when illustrating the designs, and $s = 8$ is used in actual implementation and computational experiments. Additionally, it is always assumed that the number of cores used for the parallel simplex method is s.

4.3.2 Data parallel PRICE and CHUZC

PRICE and CHUZC are often considered as the easiest and most profitable components when parallelizing the simplex method. Essentially, PRICE is a sparse matrix vector (SpMV) multiplication, and CHUZC is finding minimal ratio of two vectors, both of them are readily parallelizable. However, when considered within an advanced dual simplex implementation which uses advanced variant of PRICE and CHUZC and exploits sparsity, exploiting data parallelism becomes more complicated.

Coefficient matrix partition and permutation for PRICE

To exploit data parallelism of PRICE

$$\hat{\boldsymbol{a}}_p^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} A,$$

it is standard practice to partition the coefficient matrix column-wise $A = [A_{(1)}|A_{(2)}|\dots|A_{(s)}]$ with balanced numbers of non-zero entries in each part $A_{(i)}$. By partitioning the tableau row accordingly, a data parallel PRICE operation can be achieved by computing each segment

$$\hat{\boldsymbol{a}}_{p(i)}^{\mathrm{T}} = \hat{\boldsymbol{e}}_p^{\mathrm{T}} A_{(i)},\ i = 1, 2, \dots, s.$$

In the simplex method, $\boldsymbol{e}_p^{\mathrm{T}}$ is often extremely sparse so PRICE is more efficiently arranged row-wise by scanning non-zero entries of $\hat{\boldsymbol{e}}_p^{\mathrm{T}}$ and adding up the corresponding rows of A. To exploit the sparsity with the partitioned matrix, a row-wise copy of each partition $A_{(i)}$ is formed.

For certain LP problems, the coefficient matrix appears in a special format which is, though well formed, very inefficient when exploiting data parallelism because of load balance. One such case (LP problem FOME13, though formed by duplicating the same LP problems, is an interesting extreme case for discussion) is illustrated by Figure 4.4.

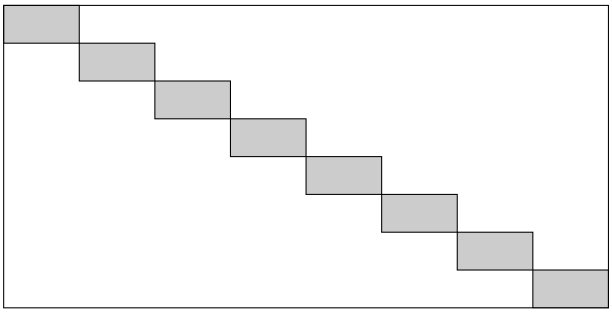


Figure 4.4: Well formed LP coefficient matrix, poor load balance if partitioned directly.

At first glance, the coefficient matrix is well formed and well partitioned, and it seems require negligible effort if the coefficient matrix is partitioned into 8 blocks. However, because of the nature of the simplex method, where the coefficient matrix is updated by Gaussian elimination, the operation involved in one block will not affect others. Therefore, for the result of PRICE operation, the p^{th} row of the updated row, its length, and thus the PRICE operation is limited to one of the block only. Therefore, directly partitioning the coefficient matrix of this extreme example or other similar formats will lead to poor load balances. Thus, to promote a good load balance for the simplex method, the coefficient matrix and related bound and cost vectors are randomly shuffled column-wise before partitioning in PAMI.

Different stages of advanced CHUZC

The textbook version of dual CHUZC only chooses variable q associated with the smallest ratio of dual variables $\hat{\boldsymbol{c}}$ and the updated tableau row $\hat{\boldsymbol{a}}_p$ by

$$q = \arg\min \hat{a}_{pj}/\hat{c}_j, \text{ for } \hat{a}_{pj} > 0,$$

which can be easily parallelized. For the most advanced variant of dual CHUZC, which involves both BFRT and Harris two-pass ratio test, and consists of multi stages, exploiting parallelism is more complicated.

As discussed in Section 3.2.3, the advance dual CHUZC can be split into two major stages of operations

1. CHUZC1: Merging various qualification situations by choosing $\bar{a}_{pj} = \hat{a}_{pj} \times s_p \times \delta_j > 0$.
2. CHUZC2: Performing bound flipping ratio test (BFRT) with the qualified variables.

The first stage (CHUZC1) is a simple one pass operation and thus can be easily parallelized. Additionally, the CHUZC1 on one segment can be performed right after the PRICE operation associated with that partition, sharing a parallelization initialization with the PRICE operation.

The second stage (CHUZC2) is a multi-pass operation where, at the end of each pass, a global minimization (for ratio) and sum (for primal infeasibility reduction) is computed and used to decide whether to perform the next pass. The multiple-pass behaviour with global minimization and sum of CHUZC2 would require the same number of synchronization (costs microsecond level of time) operations if it is parallelized. Fortunately, as shown in Table 4.3, a detailed profiling with the reference set indicates that CHUZC2 turns out to be a relative cheap component which, in average, takes 1.5% of overall solution time and costs only tens of microseconds per iteration. Therefore, CHUZC2 is not parallelized.

Logical part of the LP coefficient matrix

So far, the identity component of the coefficient matrix has not been discussed. When an LP problem is solved by the revised simplex method, the coefficient matrix of the LP is always augmented with an identity matrix as $A = [\bar{A}|I]$, where $\bar{A}$ is the coefficient matrix associated with the LP model. $\bar{A}$ and I, and the associated updated tableau row $\hat{\boldsymbol{a}}_{p(\bar{A})}$ and $\hat{\boldsymbol{a}}_{p(I)}$ are often called the structural part and the logical part respectively.

Obviously, it is not necessary to perform the PRICE operation for the logical part because the result of that part is already known as $\hat{\boldsymbol{a}}_{p(I)} = \hat{\boldsymbol{e}}_p$. Avoiding computing $\hat{\boldsymbol{a}}_{p(I)}$ explicitly also avoids any potential permutation of I or load imbalance without permutation.

One issue associated with avoiding computations of $\hat{\boldsymbol{a}}_{p(I)}$ is how to parallelize CHUZC1 associated with the $\hat{\boldsymbol{e}}_p$ segment, especially considering data affinity when $\hat{\boldsymbol{e}}_p$ is sparse and in indexed form. Again, inspecting the performance profile gives a simple solution. As shown in Table 4.3, PRICE and CHUZC together cost 27% of overall solution time, while CHUZC1 costs 7%. It is reasonable to assume the CHUZC1 associated with $\hat{\boldsymbol{e}}_p$ only would cost less. Therefore, it follows a mixed parallelism scheme which performs PRICE and CHUZC1 for the structural part in data parallel (with $s-1$ processors), while performs CHUZC1 for the logical part on a single processors in task parallel with the data parallel part.

Full parallel scheme of PRICE and CHUZC

By incorporating all the considerations above, the full parallel scheme of PRICE and CHUZC is given in Figure 4.5.

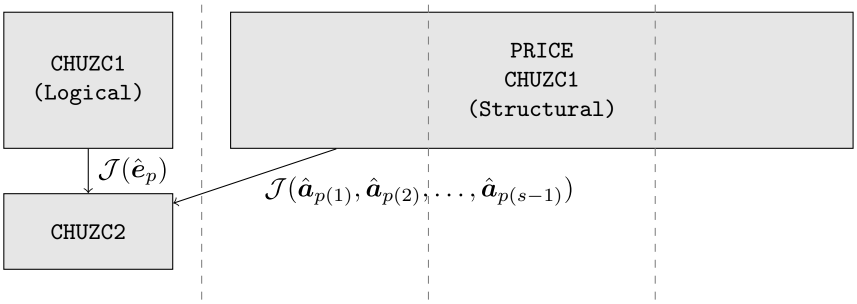

CHUZC1
(Logical)
PRICE
CHUZC1
(Structural)
$\mathcal{J}(\hat{\boldsymbol{e}}_p)$
$\mathcal{J}(\hat{\boldsymbol{a}}_{p(1)}, \hat{\boldsymbol{a}}_{p(2)}, \ldots, \hat{\boldsymbol{a}}_{p(s-1)})$
CHUZC2

Figure 4.5: The parallel scheme of PRICE and CHUZC in PAMI

In the final parallel scheme, the structural part of the coefficient matrix is partitioned to $s-1$ segments, leaving one processor performing CHUZC1 for the logical segment. The parallelization finishes after CHUZC1, when all the sub lists $\mathcal{J}(\hat{\boldsymbol{e}}_p)$ and $\mathcal{J}(\hat{\boldsymbol{a}}_{p(i)})$, $i = 1, 2, \ldots, s-1$ are joined together before CHUZC2.

In addition to the parallel scheme, the parallelization for a certain (minor) simplex iteration is dynamically switched off when the work is known to be trivial. For example, when the updated $\hat{\boldsymbol{e}}_p$ has only one or tens of non-zero entries, then parallel computation may actually result in worse performance because of the overhead of starting the parallelization environment. In the actual implementation, the density of $\hat{\boldsymbol{e}}_p$ (threshold 1%) is used decide whether or not to switch off parallel PRICE and CHUZC. The same switching off consideration is applied throughout the design and implementation of PAMI, especially for the closely related UPDATE-DUAL operation.

4.3.3 Task parallel BTRAN and FTRAN

The more important part of PAMI is the scope for task parallelization of multiple BTRAN and FTRAN. Within PAMI, there are s regular transposed linear systems associated with the initial s candidates of set $\mathcal{P}$, and up to $3 \times t, t \leq s$ forward linear systems corresponding to the final pivoting sequences $\{p_i, q_i\}_{i=0}^{t-1}$ identified by minor iterations.

Parallel BTRAN operations

In PAMI, the s transposed systems are all solved with B_k before the start of minor iterations.

$$\hat{\boldsymbol{e}}_p^{\mathrm{T}} = \boldsymbol{e}_p^{\mathrm{T}} B^{-1}, \text{ for each } p \in \mathcal{P},$$

The BTRAN operations are easily arranged as s parallel tasks in the implementation.

At the beginning of a minor iteration, the most attractive $p' \in \mathcal{P}$ is identified and removed from $\mathcal{P}$ as the leaving variable. During the minor iteration, an entering variable q' is identified by PRICE and subsequent CHUZC. At the end of the minor iteration, for each p remaining in $\mathcal{P}$, the corresponding BTRAN result $\hat{\boldsymbol{e}}_p^{\mathrm{T}}$ is updated by one APF update (essentially the same as the update of the p^{th} tableau row by the pivotal row in the tableau simplex method)

$$\hat{\boldsymbol{e}}_p^{\mathrm{T}} := \hat{\boldsymbol{e}}_p^{\mathrm{T}} - \frac{\hat{\boldsymbol{e}}_p^{\mathrm{T}} \boldsymbol{a}_{q'}}{\hat{a}_{p'q'}} \hat{\boldsymbol{e}}_{p'}^{\mathrm{T}}, \text{ for each } p \in \mathcal{P}.$$

In PAMI, the update of $\hat{\boldsymbol{e}}_p$ is performed sequentially if the BTRAN results are generally sparse, or by exploiting data parallelism if the BTRAN results are generally dense. The operation to update BTRAN results is called UPDATE-ROWS.

Parallel FTRAN operations

Compared to solving transposed systems, solving forward systems in PAMI is more complicated. There are three types of forward systems and three groups of FTRAN operations in PAMI: t regular FTRANs for obtaining updated tableau columns $\hat{\boldsymbol{a}}_q = B^{-1}\boldsymbol{a}_q$ associated with the entering variable identified during minor iterations; t additional FTRAN-DSE operations for obtaining the DSE updating vector $\boldsymbol{\tau} = B^{-1}\hat{\boldsymbol{e}}_p$; and up to t FTRAN-BFRT calculations for updating the primal solution required by bound flips identified in BFRT. Each system in a group is associated with a different basis matrix, $B_k, B_{k+1}, \dots, B_{k+t-1}$. For example the t regular forward systems for obtaining updated tableau columns are $\hat{\boldsymbol{a}}_{q_0} = B_k^{-1}\boldsymbol{a}_{q_0}, \hat{\boldsymbol{a}}_{q_1} = B_{k+1}^{-1}\boldsymbol{a}_{q_1}, \dots, \hat{\boldsymbol{a}}_{q_{t-1}} = B_{k+t-1}^{-1}\boldsymbol{a}_{q_{t-1}}$.

For regular FTRAN and FTRAN-DSE, the i^{th} linear system (which requires B_{k+i}^{-1}) in each group, is solved by starting from B_k^{-1} and then a few PF transformations given by $\hat{\boldsymbol{a}}_{q_j}, j < i$ bring the result up to date. The operations with B_k^{-1} and PF transformations are called the inverse part and the update part respectively. The inverse part is easily arranged as a task parallel computation. The update part of the regular FTRAN operations requires results of other forward systems in the same group and thus cannot be performed as task parallel calculations, but it is possible and valuable to exploit data parallel in PF update when $\hat{\boldsymbol{a}}_{q_i}$ is large and dense. For the group FTRAN-DSE, it is possible to fully exploit task parallelism if this group is arranged as calculations standalone after the regular FTRAN. However, when implementing PAMI, both FTRAN-DSE and regular FTRAN are arranged together as a bunch of parallel tasks, where the update part of FTRAN-DSE is also performed afterwards with potential scope for data parallelization.

The group of up to t linear systems with FTRAN-BFRT is slightly different from the other two groups of FTRAN operations. Firstly, there may be 0 to t linear systems depending how many minor iterations are associated with actual bound flips. More importantly, the result of FTRAN-BFRT is only used to update the primal solutions $\boldsymbol{x}_B$ by simple vector addition, which can be expressed as a single operation

$$\boldsymbol{x}_B := \boldsymbol{x}_B + \sum_{i=0}^{t-1} B_{k+i}^{-1}\boldsymbol{a}_{Fi} = \boldsymbol{x}_B + \sum_{i=0}^{t-1}\left(\prod_{j=i-1}^{0} E_j^{-1} B_k^{-1}\boldsymbol{a}_{Fi}\right) \tag{4.1}$$

where one or more of $\boldsymbol{a}_{Fi}$ may be a zero vector. By using the regular PF update, each FTRAN-BFRT operation starts from the same basis inverse B_k^{-1} but finishes with different numbers of PF update operations. Although these operations are highly related to each other, they cannot be combined. On the other hand, if an APF update (introduced in Section 2.5.1) is used before applying B_k^{-1}, so that B_{k+i}^{-1} can be expressed as

$$B_{k+i}^{-1} = B_k^{-1} T_0^{-1} \dots T_{i-1}^{-1},$$

then the primal update equation (4.1) can be rewritten as

$$\boldsymbol{x}_B := \boldsymbol{x}_B + \sum_{i=0}^{t-1}\left(B_k^{-1}\prod_{j=0}^{i-1} T_j^{-1}\boldsymbol{a}_{Fi}\right) \tag{4.2}$$

where the t linear systems start with a light-weight APF update part and finish with the same B_k^{-1} and thus can be combined as one. By using this approach, the forward linear systems associated with BFRT are always simplified as one (unless there was none originally). An additional benefit of this combination is that the UPDATE-PRIMAL operation is also reduced to a single operation after the combined FTRAN-BFRT.

By combining several FTRAN-BFRT operations into one, the number of forward linear systems is reduced to $2 \times t + 1$, or $2 \times t$ when no bound flips are performed. An additional benefit with this combination is that, when $t < s$, the total forward linear systems to solve is less than $2 \times s$, so that in average each of s processors will solve two linear systems. On the other hand, when $t = s$ and there is FTRAN-BFRT, one of the s processors is required to solve three linear systems, while the other processors are assigned only two, resulting in an "orphan task". To avoid this situation, the number of minor iterations is limited to $t = s - 1$ if bound flips have been performed in previous $t - 1$ iterations.

This combination is extended to include the regular FTRAN by adding $\theta_{p_i} \times \boldsymbol{a}_{q_i}$ to $\boldsymbol{a}_{Fi}$, so that there is only a single UPDATE-PRIMAL required over all. Even when there is no FTRAN-BFRT originally, unless $t < s$, the regular FTRAN operations are still combined to perform UPDATE-PRIMAL more efficiently.

However, different from the major benefit of avoiding each single FTRAN-BFRT with $\boldsymbol{a}_{Fi}$, combining regular FTRAN does not avoid solving the original forward system as the result $\hat{\boldsymbol{a}}_{q_i}$ is required together with $\boldsymbol{\tau}_i$ in UPDATE-WEIGHT operations for updating the DSE weight.

As for the possibility of using APF for other FTRAN operations to avoid a standalone update phase with regular PF updates, using it with regular FTRAN is not possible, as the intermediate result of regular FTRAN with B_k^{-1} is required to perform Forrest-Tomlin update. Using APF with FTRAN-DSE is possible, but it would require a relatively expensive dot product of each two different $\boldsymbol{e}_{p_j}, \boldsymbol{e}_{p_i}$, which may be less efficient than regular PF update.

The detail of the task parallel FTRAN operations discussed so far is summarized in Figure 4.6. In the actual implementation, the $2 \times t + 1$ FTRAN operations are all started the same time as parallel tasks, and the processors are left to decide which ones to perform.

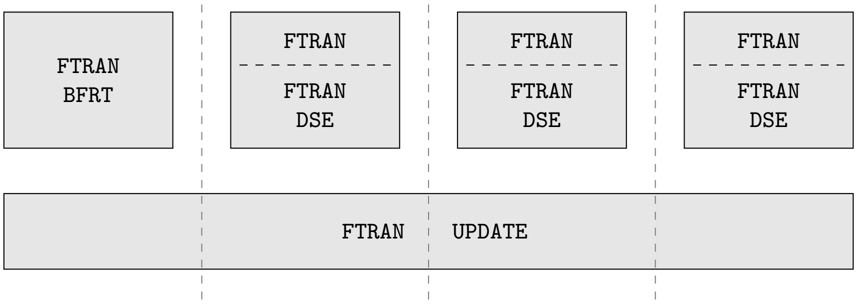


Figure 4.6: Task parallel scheme of all FTRAN operations in PAMI

4.3.4 Basis inversion and its update

Since the INVERT operation is a long sequence of (up to m) highly sparse Gaussian eliminations, it is extremely hard to exploit any parallelism.

Though it is possible to overlap INVERT with major iterations, as pointed out by Hall and McKinnon [29], the overlapping may cause numerical problems. Also because the difficulty of implementation, the overlapping scheme is not currently included in PAMI. From another aspect, a quick improvement for PAMI can be achieved by performing INVERT less frequently. As discussed in Section 2.4.4, in the sequential code, the reinversion interval is determined by comparing the accumulated time (t_U^+) of the FT update part of FTRAN (including FTRAN-DSE) and BTRAN operations since last INVERT to the time (t_I) spent on last INVERT. For the sequential simplex implementation, Once $t_U^+ > t_I$ a reinversion is performed. In the PAMI, the actual time spent on FT update part is reduced because linear systems are solved in parallel. Therefore, a slightly larger threshold is used to allow a larger reinversion interval. In implementation of PAMI, the threshold is set to $t_U^+ = 1.5 \times t_I$.

The update of basis inverse representation with PAMI is achieved by the collective Forrest-Tomlin (CFT) as introduced in Section 2.5.3. By using CFT, the basis inverse representation is updated directly from B_k^{-1} to B_{k+t}^{-1}. The CFT is always performed sequentially. The design and implementation of CFT is motivated by PAMI.

4.3.5 Major and minor CHUZR

The only computational component which has not yet been discussed is CHUZR, including major CHUZR for forming the set $\mathcal{P}$ and minor CHUZR for choosing pivots from it.

In terms of parallelization, major CHUZR exhibits scope for simple data parallelization. When the LP system is relatively dense, major CHUZR can be performed in data parallel. When the LP system is sparse, by using hyper-sparse major CHUZR (introduced in Section 3.2.2), often there are only hundreds of candidates to examine, and thus requires no parallelization.

In terms of efficiency of choosing pivots for PAMI, CHUZR is extremely important and more complicated. If lots of candidates are initially identified in a major CHUZR but are no longer qualified or attractive before used, then it results in a big waste of time on other components, especially the task parallel BTRAN operations. On the other hand, if less qualified candidates are often chosen in minor CHUZR after the first minor iteration, then it may lead to a much larger number of simplex iterations.

This section discusses how the quality of candidates can be controlled.

Controlling the quality of candidates in minor CHUZR

In PAMI, the quality of a candidate $p \in \mathcal{P}$ is measured by a relative *cutoff* ratio ψ and its attractiveness α_p. During minor iterations, if the attractiveness of a candidate p drops below its initial value α_p^i computed in major CHUZR by

$$\alpha_p < \psi \alpha_p^i,$$

then this candidate is dropped. Within the framework of DSE, the attractiveness is defined as

$$\alpha_p = \Delta x_p^2 / ||\hat{\boldsymbol{e}}_p||_2^2, \text{ where } \Delta x_i = \begin{cases} l_p - x_p & \text{if } x_p < l_p \\ x_p - u_p & \text{if } x_p > u_p \\ 0 & \text{otherwise} \end{cases}.$$

To determine the best choices of ψ, a series of experiments have been carried out for the reference set (30 LP problems) with various cutoff ratios ranging from 1.001 to 0.01. Computational results are presented in Table 4.4.

Cutoff ratio $\psi = 1.001$ corresponds to a special situation, where candidates associated with improved attractiveness are chosen. The speedup with $\psi = 1.001$ is as can be expected, poor.

Cutoff ratio $\psi = 0.999$ (the value 0.999 is used rather than 1.0 to introduce a tolerance to exclude numerical noise) corresponds to a boundary situation, where candidates associated with same or better attractiveness are chosen. Under this cutoff ratio, an average speedup of 1.52 is achieved.

With different cutoff ratio $0.9 \leq \psi \leq 0.999$, there is no really difference in the performance of PAMI. The average speedup and counts of larger speedup (columns headed "#1.6 speedup", "#1.8 speedup" and "#2 speedup") instances are relatively stable.

Starting from $\psi = 0.9$, decreasing the cutoff merit results in a clear decrease of the average speedup, though the counts of larger speedup remains stable until $\psi = 0.5$.

In summary, experiments suggest that the any value in interval $[0.9, 0.999]$ can be chosen as the cutoff ratio, and in PAMI, the median $\psi = 0.95$ is chosen.

4.3.6 Computational results and analysis

The performance of PAMI is assessed in this section by comparing with the HSOL and the serial version of PAMI. Numerical results are presented in Table 4.5.

cutoff (ψ)	speedup	#1.6 speedup	#1.8 speedup	#2.0 speedup
1.001	1.12	1	1	0
0.999	1.52	11	7	5
0.99	1.54	13	6	4
0.98	1.53	15	8	5
0.97	1.48	11	6	5
0.96	1.52	12	8	6
0.95	1.49	13	8	4
0.94	1.56	13	8	4
0.93	1.47	13	9	4
0.92	1.52	14	7	4
0.91	1.52	14	5	3
0.9	1.50	12	9	4
0.8	1.46	13	9	3
0.7	1.46	15	9	4
0.6	1.44	11	8	6
0.5	1.42	13	5	3
0.2	1.36	10	6	4
0.1	1.29	10	7	3
0.05	1.16	9	4	2
0.02	1.28	10	6	2
0.01	1.22	8	5	3

Table 4.4: Experiments with different cutoff merit for controlling candidate quality in PAMI

The solution times of HSOL, the serial version of AMI and PAMI running in parallel with 8 cores are listed in columns headed SEQ, PAMI1 and PAMI8 respectively in Table 4.5. The last two columns headed FTRAN and BTRAN, indicating the occurrence of hyper-sparse results, are included for reference.

For more than 65% of the reference set, the speedup of PAMI compared to its sequential version is more than 2, with an average of 2.23. However, when compared to the regular dual simplex method, the sequential version of PAMI is generally the less efficient (about 30% slower) implementation. The overall speedup of PAMI is thus compromised, resulting in a speedup (over HSOL) of about 1.5 on average. When solved with PAMI, 8 out of 30 instances achieved more than 1.8 speedup, and the best speedup obtained in 2.53.

It is worth noting that the instances with better speedup (greater than the average) are more associated with the relatively dense LP problems (where occurrence of hyper-sparse FTRAN and BTRAN is low). This happens partially because PAMI frequently leads to a smaller number of iterations for the dense problems. For example, the iteration counts when solving DFL001 with HSOL and PAMI are 26322 and 23668 respectively. However, the performance of PAMI1 is worse than HSOL because of wasted BTRAN operations and other inefficiencies of PAMI1.

The performance of PAMI for solving dense instances is not even. The worst cases are also associated with dense LP problems. For example, using PAMI to solve MAROS-R7 and LINF_520C, results in slowdown in both case, and almost no speedup is obtained for SELF. This can be explained with reference to the profiling of computational components provided in Table 4.3, where the proportion for INVERT of these three models is the biggest among all dense LP problems.

The performance of PAMI when solving hyper-sparse LP problems is moderate but relatively stable. Out of the three best instances of the hyper-sparse problems, namely CRE-B, SGPF5Y6 and STP3D, the large percentage (see Table 4.3 for reference) of PRICE (42.9% for CRE-B and

	Solution time			Speedup			Sparsity	
Model	seq	pami1	pami8	p1/seq	p8/p1	p8/seq	ftran	btran
cre-b	6.2	5.1	3.2	1.21	1.62	1.95	100	83
dano3mip_lp	52.6	76.9	24.8	0.68	3.10	2.12	1	6
dbic1	78.9	141.3	60.0	0.56	2.36	1.31	100	83
dcp2	12.9	16.0	8.5	0.81	1.89	1.52	100	97
dfl001	15.7	23.7	8.6	0.66	2.74	1.81	34	57
fome12	99.5	159.4	61.9	0.62	2.58	1.61	45	58
fome13	256.1	371.0	168.5	0.69	2.20	1.52	100	98
ken-18	13.5	16.0	10.4	0.84	1.54	1.30	100	100
l30	10.9	23.3	8.5	0.47	2.74	1.29	10	8
Linf_520c	3460.6	9182.9	4587.5	0.38	2.00	0.75	10	11
lp22	22.0	36.4	13.2	0.61	2.75	1.67	13	22
maros-r7	11.3	37.4	24.0	0.30	1.56	0.47	5	13
mod2	52.7	103.2	40.7	0.51	2.53	1.29	46	68
ns1688926	24.8	41.2	19.6	0.60	2.10	1.26	72	100
nug12	125.2	197.8	70.5	0.63	2.81	1.78	1	20
pds-40	28.4	42.2	21.0	0.67	2.00	1.35	100	98
pds-80	66.0	109.4	57.0	0.60	1.92	1.16	100	99
pds-100	84.3	122.8	65.5	0.69	1.88	1.29	100	99
pilot87	6.6	11.0	4.4	0.60	2.48	1.50	10	19
qap12	157.5	170.9	62.1	0.92	2.75	2.53	2	15
self	39.0	66.2	36.5	0.59	1.81	1.07	0	2
sgpf5y6	169.0	213.5	88.8	0.79	2.40	1.90	100	100
stat96v4	156.7	235.3	67.3	0.67	3.50	2.33	73	31
stormG2-125	9.4	11.0	6.5	0.85	1.70	1.44	100	100
stormG2-1000	427.0	576.6	256.7	0.74	2.25	1.66	100	100
stp3d	565.2	546.3	234.9	1.03	2.33	2.41	95	70
truss	7.9	10.8	4.0	0.73	2.67	1.94	37	2
watson_1	50.2	59.4	32.4	0.85	1.83	1.55	100	100
watson_2	53.6	57.5	33.3	0.93	1.72	1.61	100	100
world	65.1	120.7	47.5	0.54	2.54	1.37	41	61
Geomean	50.8	76.0	34.1	0.67	2.23	1.49		

Table 4.5: Performance and speedup of PAMI

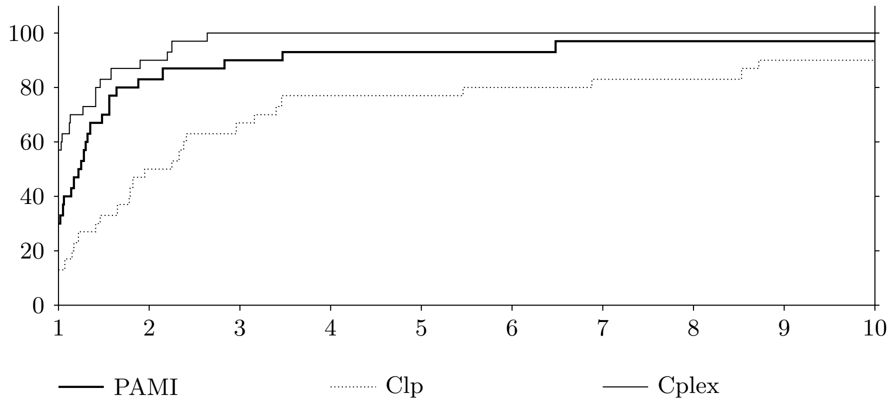


Figure 4.7: Elapsed-time profile of PAMI, Clp and Cplex

19.2% for STP3D) and FTRAN-DSE (80.7% for SGPF5Y6 and 27% for STP3D) explains the good speedup. In PAMI, the PRICE and FTRAN-DSE components, can be performed efficiently as task parallel and data parallel computations respectively, and therefore a larger percentage of these components yields a natural source of speedup.

The overall performance of PAMI is assessed via the performance profile shown in Figure 4.7. As the figure indicates, the performance of PAMI is comparable with the dual simplex implementation of Cplex, a world-leading commercial LP solver.

4.3.7 Real time behaviour

This section further analyses the performance of PAMI, especially the relative poor speedup on sparse LP problems, by investigating real time behaviour of PAMI by using Gantt charts. Table 4.6 lists the colours used for representing different computational components. Related components are grouped with a family of colours.

Colour	Component
🟩	CHUZR
🟦	BTRAN
🟥	PRICE
🟥	CHUZC1
🟥	CHUZC2
🟦	UPDATE-ROWS
🟦	UPDATE-DUAL
🟪	FTRAN
🟦	FTRAN-DSE
🟦	FTRAN-BFRT
🟩	UPDATE-PRIMAL
🟩	UPDATE-WEIGHT
🟩	UPDATE-FACTOR

Table 4.6: Colours for different components

Sparse LP problem 1

Figure 4.8 illustrates the real time behaviour at the 20000th major iteration (with 7 minor iterations) when solving the hyper-sparse LP problem WATSON_2 with PAMI.

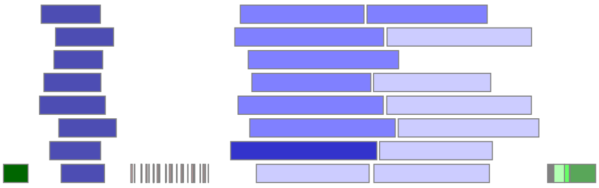


Figure 4.8: PAMI behaviour: WATSON_2, 20000th major iteration, 7 minor iterations, 275 ms

This illustration shows one major limitation of PAMI, that when the LP problem is hyper-sparse, then “cheap components” CHUZR, PRICE, CHUZC (merely visible between BTRAN 🟦 and FTRAN 🟪) and even UPDATE-PRIMAL are all performed sequentially to avoid slowdown from the parallelization overhead. This arrangement inevitably limits the overall speedup as it is obviously counted as the sequential proportion in Amdahl’s Law.

Sparse LP model 2

Figure 4.9 illustrates another typical case when applying PAMI to sparse LP problems. It shows the 5800th major iteration when solving PDS-20, which contains 6 minor iterations.

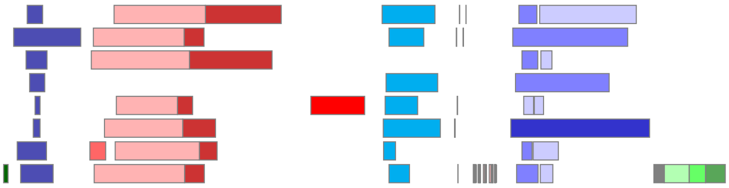


Figure 4.9: PAMI behaviour: PDS-20, 5800th major iteration, 6 minor iterations, 641 ms

In additional to the "cheap components" issue there is an obvious load imbalance in this instance. One BTRAN and the combined FTRAN-BFRT clearly dominate the whole BTRAN and FTRAN operations respectively. The reason for this load imbalance, as discussed in Section 3.3, can be explained as the occasional inefficiency of the DSE framework for certain families of LP problems (for example, PDS problems), that it tend to choose a pivot associated with dense BTRAN result. When the LiDSE approach is used, the expected real time behaviour when solving PDS-20 is expect to be similar to that of WATSON_2.

4.4 Exploiting single iteration parallelism

This section introduces a relative simple approach which explores single iteration parallelism (SIP) of the dual simplex algorithm.

4.4.1 Data dependency and parallelization scheme

Experience with PAMI shows that, the sequential PAMI (which is essentially a dual simplex variant) frequently results in a worse simplex solving path compared to the regular version. Thus, although the relative speedup of PAMI8 to PAMI1 is reasonably good, it diminishes when compared with the regular sequential version. This issue motivated another approach which exploits purely single iteration parallelism (SIP) of the regular dual simplex method.

SIP is a further development of the work [8] of Bixby and Martin in the late 1990s. In their work, the FTRAN and FTRAN-DSE are solved together sequentially rather than in parallel as the computational experience suggested so. However, the costs of FTRAN and FTRAN-DSE are typically very different because of the discovery and exploitation of hyper-sparsity since the 2000s. For many LP problems, the time required by FTRAN is generally much less than that of FTRAN-DSE. Moreover, the FTRAN-BFRT component, for the bound-flipping ratio test, whose density and cost is similar to FTRAN, is not included in their discussion. Because of these developments of the dual simplex method, the parallelization scope of FTRAN operations deserves further consideration, which jointly motivated the design and implementation of SIP.

The mixed parallelization scheme of SIP is illustrated in Figure 4.10. Data dependency of each computational component is also labelled in the figure. PRICE and CHUZC share a similar permutation and partitioning scheme as discussed in Section 4.3.2. The only difference is that for SIP, the matrix is partitioned into $s-2$ parts because one of the processor is used to perform the FTRAN-DSE, in parallel with PRICE and CHUZC. Also the subsequent UPDATE-DUAL is arranged as a single task as its cost is relatively less than other components.

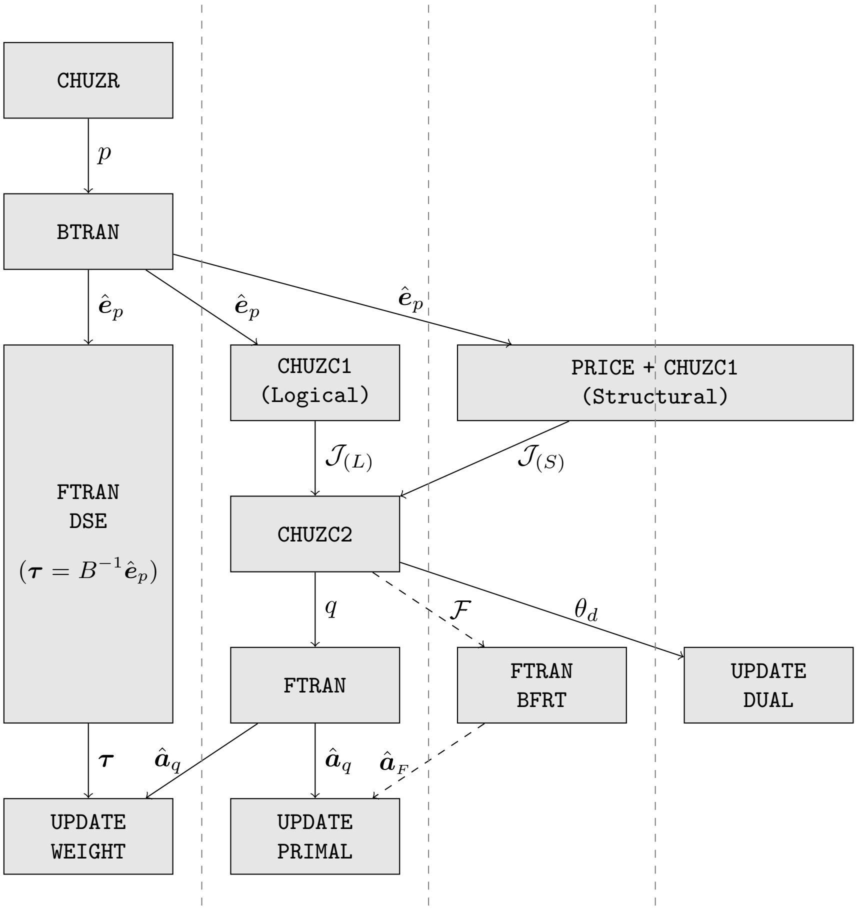


Figure 4.10: SIP data dependency and parallelization scheme

Compared to the regular sequential simplex method, the only difference besides coefficient matrix permutation is that the FTRAN-DSE is performed by firstly copying $\hat{e}_p$. In the regular dual simplex method, τ and $\hat{e}_p$ can share the same memory space if FTRAN-DSE is arranged after CHUZC.

4.4.2 Computational results and analysis

The computational performance and speedup of SIP when using 8 processors is given in Table 4.7. The data for PAMI, and occurrence of hyper-sparse results (in columns headed FTRAN and BTRAN) are included for reference.

Clearly the overall performance and average speed (1.13) of SIP is inferior to PAMI. This is because SIP, after all, exploits only limited parallelism.

The worst cases when using SIP are associated with the hyper-sparse LP problems, where applying SIP to most of them results in slowdown. A typical example is SGPF5Y6, where the proportion of FTRAN-DSE is more than 80% and the total proportion of PRICE, CHUZC, FTRAN and UPDATE-DUAL is less than 5%. Therefore, when performing FTRAN-DSE and the rest as task parallel operations, the overall performance is not only limited by FTRAN-DSE, but also the competition for memory accessing by the other components and the cost for setting up the parallel environment will slow down the FTRAN-DSE.

On the other hand, when applied to dense LP problems, the performance of SIP is moderate

	Solution time			Speedup		Sparsity	
Model	SEQ	SIP	PAMI	SIP	PAMI	ftran	btran
cre-b	6.2	5.7	3.2	1.08	1.95	100	83
dano3mip_lp	52.6	35.4	24.8	1.49	2.12	1	6
dbic1	78.9	65.1	60.0	1.21	1.31	100	83
dcp2	12.9	10.5	8.5	1.23	1.52	100	97
dfl001	15.7	12.2	8.6	1.28	1.81	34	57
fome12	99.5	81.3	61.9	1.22	1.61	45	58
fome13	256.1	207.4	168.5	1.24	1.52	100	98
ken-18	13.5	17.1	10.4	0.79	1.30	100	100
l30	10.9	8.5	8.5	1.28	1.29	10	8
Linf_520c	3460.6	2644.6	4587.5	**1.31**	0.75	10	11
lp22	22.0	15.1	13.2	1.45	1.67	13	22
maros-r7	11.3	10.1	24.0	**1.12**	0.47	5	13
mod2	52.7	42.3	40.7	1.25	1.29	46	68
ns1688926	24.8	20.1	19.6	1.23	1.26	72	100
nug12	125.2	108.3	70.5	1.16	1.78	1	20
pds-40	28.4	24.6	21.0	1.16	1.35	100	98
pds-80	66.0	63.1	57.0	1.05	1.16	100	99
pds-100	84.3	79.7	65.5	1.06	1.29	100	99
pilot87	6.6	5.1	4.4	1.31	1.50	10	19
qap12	157.5	190.8	62.1	0.83	2.53	2	15
self	39.0	28.7	36.5	**1.36**	1.07	0	2
sgpf5y6	169.0	252.2	88.8	0.67	1.90	100	100
stat96v4	156.7	76.4	67.3	2.05	2.33	73	31
stormG2-125	9.4	12.3	6.5	0.76	1.44	100	100
stormG2-1000	427.0	510.2	256.7	0.84	1.66	100	100
stp3d	565.2	492.1	234.9	1.15	2.41	95	70
truss	7.9	5.0	4.0	1.58	1.94	37	2
watson_1	50.2	62.6	32.4	0.80	1.55	100	100
watson_2	53.6	68.4	33.3	0.78	1.61	100	100
world	65.1	51.3	47.5	1.27	1.37	41	61
Geomean	50.8	44.8	34.1	1.13	1.49		

Table 4.7: Performance and speedup of SIP

and relative stable. This is especially so for these instances where PAMI results in clear slowdown: for Linf_520c, maros-r7, applying SIP achieves speedup of 1.31 and 1.12 respectively.

In summary, SIP, as a straightforward parallelization approach which exploits purely single iteration parallelism, achieves relatively poor speedup for general LP problems compared to PAMI. However, SIP is frequently complementary to PAMI in achieving speedup when PAMI results in slowdown.

4.4.3 Real time behaviour

The real time behaviour of SIP and the same iteration solved by HSOL are shown in Figure 4.11 by a Gantt chart with the LP model dfl001 at the 10000th dual simplex iteration. The same colour scheme in Table 4.6 is applied.

This illustration provides a chance to closely examine the overhead and efficiency of the parallelization scheme. Firstly, each single ftran operation (ftran in □, ftran-dse in ■ and ftran-bfrt in ■), especially the ftran-dse in SIP is longer than that in HSOL. This is inevitable since simplex parallelization involves competition for memory access. Moreover, although the price ■ operation achieves speedup and load balance, the parallelization perfor-

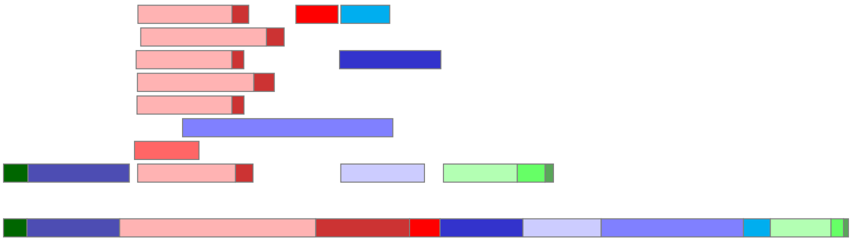


Figure 4.11: SIP behaviour: DFL001, 10000th iteration, 754 ms, and the same iteration solved by HSOL, 1144 ms

mance is highly limited by both the memory bound and the inherent inefficiency when using the partitioned matrix where, for the row-wise matrix-vector product, each row of the sub-matrix is much shorter than in the original matrix. The speedup of SIP is limited by these issues.

The same argument also holds for PAMI, especially for PRICE and CHUZC.

4.5 Summary

This chapter has introduced and analysed previous simplex parallelization attempts and described the design and development of two novel parallel dual revised simplex method implementations.

One parallelization, PAMI, achieves a speedup of 1.49 on average. Though the speedup is not extremely impressive, the overall performance of PAMI is comparable to the dual simplex implementation of Cplex, which is a world-leading commercial simplex solver.

For the relatively straightforward implementation, SIP, although it achieves merely 1.13 speedup on average, it is frequently complementary to PAMI in achieving speedup when PAMI results in slowdown.

Chapter 5

Conclusions and future work

This thesis has reported the implementation of a competitive sequential dual simplex solver and the design and development of two dual simplex parallel frameworks. The resulting sequential solver, HSOL, and one parallelization, PAMI (parallelism across multiple iterations) are comparable with the dual simplex implementations of an established public domain solver, Clp and a world-leading commercial solver, Cplex respectively. The other parallelization, SIP (single iteration parallelism) is frequently complementary to PAMI in achieving speedup when PAMI results in slowdown.

Contribution to the sequential simplex method

The research and (re)implementation of the sequential dual simplex method have resulted in many novel developments.

When implementing the linear system solution techniques, three novel simplex update variants have been developed. One of them, the MPF (middle product form) update, is comparable with the established FT (Forrest-Tomlin) update in terms of efficiency, but implementationally is much more straightforward. The other two, the CFT (collective Forrest-Tomlin) update and the APF (alternate product form) update are both key components in the PAMI parallelization framework.

When implementing the sequential dual simplex method, a phenomenon called hyper-sparsity promotion has been identified when cost perturbation is used. Further investigation has revealed that cost perturbation encourages the BFRT (bound flipping ratio test), which reduces the attractiveness of the dense tableau rows and thus reduces the appearance of them in dual CHUZR, leading to overall hyper-sparsity promotion. By following this explanation, two alternative approaches, partial CHUZR and LiDSE (less infeasibility DSE), for promoting hyper-sparsity have been experimented. The LiDSE approach has been found very successful for certain families of LP problems.

Design and development of parallel simplex solvers

Based on the efficient implementation of the sequential dual simplex method, two parallel dual simplex solvers have been designed and developed.

One relatively complicated parallelization, PAMI is based on a less-known pivoting rule called suboptimization. Suboptimization provided the scope for parallelism across multiple iterations but, as a pivoting rule, it is generally inferior to the regular dual steepest-edge algorithm. Thus, to control the qualities of the pivots, which often decline during PAMI, a *cutoff* merit is necessary. A suitable cutoff merit, 0.95, has been found via series of experiments. For the reference set, PAMI provides an average speedup of 1.49. Though not extremely impressive, the performance of PAMI is comparable to the dual simplex implementation of Cplex, which is a world-leading commercial simplex solver.

The other parallelization, SIP exploits purely single iteration parallelism. Though the average speedup, 1.13, is worse than that of PAMI, it is frequently complementary to PAMI in achieving speedup when PAMI results in slowdown. For example, when solving the LP problem LINF_520C of the reference set, PAMI solved 25% slower than the sequential solver while SIP achieved a moderate speedup of 1.31.

Future work

There are many possible future tasks.

Improving the serial implementation. The implementation of the dual revised simplex method, though it included both DSE and BFRT, is still a relatively straightforward one. For example, BFRT is not possible when none of the nonbasic variables is a `BOXED` variable, but in HSOL, a BFRT is still performed to keep using the same software procedure. In a more sophisticated implementation, handling this situation by using a regular dual ratio test (Harris ratio test) is expected to be more efficient. There are many possible improvements in this sense. In this research, it has been intended to keep the serial implementation simple so that parallelization can be relatively easily designed and implemented. Now that the parallelization framework is ready, it is desirable to polish the serial implementation to achieve better performance in both serial and parallel simplex solvers.

Promoting hyper-sparsity. Although the investigation into hyper-sparsity promotion though led to the successful LiDSE approach, it still at an initial stage. For the LiDSE approach, it is possibly to develop an "ad hoc" mechanism to activate it whenever necessary. Further study on this topic is expected to result in more alternative approaches.

Overlapping inversion. One reason for the relative poor speedup of PAMI is that the time-consuming sequential LU factorization is not overlapped with the simplex iterations. It has been reported [35, 6] that overlapping just the LU factorization with simplex iterations provides a speedup of 1.3 to 1.5 for certain LP problems. It is expected to promote the performance of PAMI further if is included in the parallelization framework. The lack of overlapping is partially because of its complicity and partially because of the potential numerical issues as pointed out by Hall and McKinnon [29]. However, previous work uses the PF update to bring the new basis inverse up-to-date. In future, it will be possible to enhance the overlapping scheme by using the FT update, in a recompute-and-catchup manner to bring the new basis inverse up-to-date. The recomputation is expected, to some extent, to resolve the numerical issue.

Bibliography

[1] 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.

[2] E.-S. Badr, M. Moussa, K. Paparrizos, N. Samaras, and A. Sifaleras. Some computational results on MPI parallel implementation of dense simplex method. *Trans Eng Comput Technol*, (17):228–231, 2006.

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

[4] M. Benichou, J. Gauthier, G. Hentges, and G. Ribiere. The efficient solution of large-scale linear programming problemssome algorithmic techniques and computational results. *Mathematical Programming*, 13:280–322, 1977.

[5] J. Bieling, P. Peschlow, and P. Martini. An efficient GPU implementation of the revised simplex method. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, 2010.

[6] R. Bixby and M. Schwabacher. Solving linear programs with two processors. Technical Report TR89-16, Department of Computational and Applied Mathematics, Rice University, 1989.

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

[8] R. E. Bixby and A. Martin. Parallelizing the dual simplex method. *INFORMS Journal on Computing*, 12(1):45–56, 2000.

[9] V. Chvátal. *Linear Programming*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1983.

[10] COIN-OR. Clp. `http://www.coin-or.org/projects/Clp.xml`. Accessed: 10/01/2013.

[11] Z. Cvetanovic, E. Freedman, and C. Nofsinger. Efficient decomposition and performance of parallel PDE, FFT, Monte Carlo simulations, simplex, and sparse solvers. *The Journal of Supercomputing*, 5:219–238, 1991.

[12] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[13] I. S. Duff. MA28 – A set of fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE R8730, AERE Harwell, 1977.

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

[15] S. K. Eldersveld and M. A. Saunders. A block-LU update for large-scale linear programming. *SIAM Journal on Matrix Analysis and Applications*, 13:191–201, 1992.

[16] R. Elwes. The algorithm that runs the world. *New Scientist*, 215(2877):32 – 37, 2012.

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

[18] J. J. H. Forrest and J. A. Tomlin. Updated triangular factors of the basis to maintain sparsity in the product form simplex method. *Mathematical Programming*, 2:263–278, 1972.

[19] R. Fourer. Notes on the dual simplex method. Technical report, Department of Industrial Engineering and Management Sciences Northwestern University, 1994. Unpublished.

[20] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 1988.

[21] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Sparse matrix methods in optimization. *SIAM Journal on Scientific and Statistical Computing*, 5:562–589, 1984.

[22] 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.

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

[24] J. Gondzio. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing*, 9(1), 1997.

[25] N. Gould and P. L. Toint. Preprocessing for quadratic programming. *Math. Program.*, 100(1):95–132, 2004.

[26] J. Hall and Q. Huangfu. A high performance dual revised simplex solver. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 143–151, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] J. A. J. Hall. *Sparse matrix algebra for active set methods in linear programming.* PhD thesis, University of Dundee Department of Mathematics and Computer Science, 1991.

[28] J. A. J. Hall. Towards a practical parallelisation of the simplex method. *Computational Management Science*, 7:139–170, 2010.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

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

[36] Q. Huangfu and J. J. Hall. Novel update techniques for the revised simplex method. Technical Report ERGO-13-001, School of Mathematics, University of Edinburgh, 2013.

[37] IBM. ILOG CPLEX Optimizer. `http://www.ibm.com/software/integration/optimization/cplex-optimizer/`.

[38] A. Koberstein. Progress in the dual simplex algorithm for solving large scale LP problems: techniques for a fast and stable implementation. *Computational Optimization and Applications*, 41(2):185–204, November 2008.

[39] A. Koberstein and U. H. Suhl. Progress in the dual simplex method for large scale LP problems: practical dual phase 1 algorithms. *Computational Optimization and Applications*, 37(1):49–65, May 2007.

[40] M. Lalami, D. El-Baz, and V. Boyer. Multi GPU implementation of the simplex algorithm. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 179 –186, 2011.

[41] C. E. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, 1(1):36–47, 1954.

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

[43] M. Lubin, J. Hall, C. Petra, and M. Anitescu. Parallel distributed-memory simplex for large-scale stochastic LP problems. *Computational Optimization and Applications*, pages 1–26, 2013.

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

[45] I. Maros. *Computational Techniques of the Simplex Method.* Kluwer Academic Publishers, Boston, 2002.

[46] I. Maros and G. Mitra. Investigating the sparse simplex algorithm on a distributed memory multiprocessor. *Parallel Comput.*, 26(1):151–170, 2000.

[47] 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.

[48] H. D. Mittelmann. Benchmarks for optimization software. `http://plato.la.asu.edu/bench.html`. Accessed: 10/01/2013.

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

[50] 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.

[51] R. R. Rosander. Multiple pricing and suboptimization in dual linear programming algorithms. *Mathematical Programming Study*, 4:108–117, 1975.

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

[53] D. Spampinato and A. Elster. Linear optimization on modern GPUs. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –8, 2009.

[54] C. B. Stunkel and D. A. Reed. Hypercube implementation of the simplex algorithm. In *Proceedings of the third conference on Hypercube concurrent computers and applications - Volume 2*, pages 1473–1482, New York, NY, USA, 1988. ACM.

[55] 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.

[56] M. E. Thomadakis and J.-C. Liu. An efficient steepest-edge simplex algorithm for SIMD computers. In *Proceedings of the 10th international conference on Supercomputing*, ICS 1996, pages 286–293, 1996.

[57] J. A. Tomlin. Pivoting for size and sparsity in linear programming inversion routines. *Journal of the Institute of Mathematics and its Applications*, 10:289–295, 1972.

[58] J. A. Tomlin. On pricing and backward transformation in linear programming. *Mathematical Programming*, 6:42–47, 1974.

[59] P. Wolfe. A technique for resolving degeneracy in linear programming. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):pp. 205–211, 1963.

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

[61] G. Yarmish and R. Slyke. A distributed, scaleable simplex method. *The Journal of Supercomputing*, 49:373–381, 2009.