Abstract

Linear programming is a form of mathematical optimisation in which one seeks to optimise a linear function subject to linear constraints on the variables. It is a very versatile tool that has many important applications, one of them being modelling of production and trade in the petroleum industry.

The Cell Broadband Engine, developed by IBM, Sony and Toshiba, is an innovative multicore architecture that has already been proven to have a great potential for high performance computing. However, developing applications for the Cell/BE is challenging, particularily due to the low-level memory management that is mandated by the architecture, and because careful optimisation by hand is often required to get the most out of the hardware.

In this thesis, we investigate the opportunities for implementing a parallel solver for sparse linear programs on the Cell/BE. A parallel version of the standard simplex method is developed, and the ASYNPLEX algorithm by Hall and McKinnon is partially implemented on the Cell/BE. We have met substantial challenges when it comes to numerical stability, and this has prevented us from spending sufficient time on Cell/BE-specific optimisation and support for large data sets. Our implementations can therefore only be regarded as proofs of concept, but we provide analyses and discussions of several aspects of the implementations, which may guide the future work on this topic.

Acknowledgements

This thesis was instigated as a collaboration between my advisor, Dr.Ing. Lasse Natvig, and Miriam AS, represented by Christophe Spaggiari, and it forms part of a larger project which also involves the Ph.D. work of Mr. Mujahed Eleyat. I would like to thank all of them for their support and input, and Dr. Anne C. Elster for feedback on the report and for giving us access to the high performance computing lab. I would also like to thank Schlumberger for funding our trip to the Supercomputing '09 conference last fall, from which we learned a lot.

A special gratitude goes to Henrik Andersson and Marielle Christiansen from the Department of Industrial Economics and Technology Management at NTNU, for their tremendously helpful response when we asked for help after having got stuck with numerical instability problems.

Thanks to my parents and my sister for thorough proofreading and moral support, particularily in the latter stages of the project, when I was working around the clock — and of course for always having kindled my interest in science throughout the years.

Thanks to my classmates at the "Ugle" computer lab for stress-relieving conversations, ping-pong matches, Rubik's cube solving and the occasional round of swing dancing. Finally, thanks to everybody in the Computer Science and Communication Technology classes of 2009, for five great years at NTNU. We made it — congratulations to everyone!

Contents

List of figures

List of tables

List of algorithms

List of source code listings

Chapter 1

Introduction

This thesis is a part of a larger project that is a cooperation between my advisor — Dr.Ing. Lasse Natvig — and the company Miriam AS. Also involved in that project is Mr. Mujahed Eleyat, whose Ph.D. thesis will be based on our work. Miriam AS develops "Regina" and "Gas", two applications for simulation of production and delivery of oil and gas products through a pipeline network (more information can be found at http://www.miriam.as/). Such situations can be modelled by a *linear program*, which is a linear function of a set of variables along with a set of constraints on the values of other linear functions of those variables. The field of linear programming revolves around the study of algorithms for finding the optimal function values of such systems, and it will be thoroughly presented in the next chapter. There are two main classes of linear programming algorithms: the simplex method (and variations thereof) and interior point methods. Although both are mentioned in the problem description, we only had the time to focus on the former.

Miriam AS has recently invested in a cluster of Playstation 3 machines containing Cell Broadband Engine processors, and is hoping to be able to utilise these to speed up their simulations. The Cell/BE is a multicore processor with two different core types: one general-purpose PowerPC core and eight cores that are specialised for high computational throughput. The computation cores do not have direct access to main memory, but they have a small cache that is controlled manually by the program that is executed on them, and a high-speed bus can be used to efficiently feed the computation cores with data. This sets the Cell/BE apart from any other computing platform, and combined with deep pipelines and vectorised instructions, it holds the promise of excellent performance if one has a problem that is suited for that platform and one invests a sufficient amount of time in the programming process.

Unfortunately, it turned out that the field of linear programming is wrought with peril, in the form of numerical instability. Neither we nor our advisor were

prepared for this (we selected this project in the belief that the primary challenge would be to program the Cell/BE), and we were not able to fully overcome the problems we encountered. Therefore, we have adjusted our aims to that of producing a code base from which further development may take place, and writing a report that is rich in background material, references and advice that we hope will prove useful to those that are to continue the project.

Outline

In Chapter 2, we give a presentation of the field of linear programming, and we describe the standard and revised simplex methods and a parallel revised simplex method called ASYNPLEX. We also explain the Cell/BE architecture and programming model.

In Chapter 3, we describe our initial plans, the progress of our work and the decisions we had to make during the project. We also present our design of a simple parallel standard simplex algorithm for Cell/BE, and our adaptations of the ASYNPLEX algorithm (which we did not get the time to fully implement). This project ended up very differently from what we had anticipated; we therefore also discuss the challenges we have encountered.

In Chapter 4, we provide a few timing analyses of the parallel standard simplex algorithm, in order to learn how the parallelisation, number of cores, and Cell/BE features such as vectorisation affect the performance. We thoroughly discuss features we did not get the time to implement, and give several pieces of advice to the researchers that will build upon our work.

Finally, in Chapter 5, we present our conclusions and summarise our suggestions for future work.

Chapter 2

Background

This chapter will give the reader the necessary theoretical background for the main subjects of this thesis: linear programming and the Cell Broadband Engine. We also give some notes on a data set collection called `netlib`, on how to represent sparse vectors, and on a formula for the maximal speedup that can be obtained when parallelising a program.

2.1 Linear programming

The term *linear programming* (LP) refers to a type of optimisation problems in which one seeks to maximise or minimise the value of a linear function of a set of variables[1]. The values of the variables are constrained by a set of linear equations and/or inequalities. Linear programming is a fairly general problem type, and many important problems can be cast as LP problems — for instance, shortest path problems and maximum flow problems (see [8]). However, the true virtue of linear programming stems from its ability to model a vast range of optimisation problems for which specialised algorithms do not exist, including many situations from economics and industry processes.

This entire section is primarily based on Vanderbei[47] and Cormen et al.[8].

2.1.1 Problem formulation. Standard and slack forms

The following framed text is an example of a simple linear programming problem. We will use this example throughout this section to illustrate how the linear programming algorithms work.

[1] Hence, LP is not (as the name would seem to suggest) a programming technique. The name originated in the 1940s, when "program" referred to military supply plans and schedules.

— Example —

A company owns a factory that makes two kinds of products based on two different raw materials. The profit the company makes per unit of product A is \$30, and the profit of product B is \$20. Producing one unit of A requires 1 unit of raw material R and 1 unit of raw material S; one unit of B requires 2 units of R and 1 unit of S. The company possesses 40 units of R and 50 units of S. We make the simplifying assumptions that all prices are constant and cannot be affected by the company, and that the company is capable of selling everything it produces. The company's goal is to maximise the profit, which can be described as $30x_1 + 20x_2$, where x_1 is the number of units of product A and x_2 is the number of units of product B. The following constraints are in effect:

- $x_1 + x_2 \leq 40$ (the production of A and B cannot consume more units of raw material R than the company possesses)
- $2x_1 + x_2 \leq 50$ (similarly for raw material S)
- $x_1, x_2 \geq 0$ (the company cannot produce negative amounts of its products)

Note that in regular LP problems, one cannot restrict the variables to be integers — in fact, adding this requirement produces a new kind of problem known as *integer linear programming* (ILP), which is *NP*-hard[2]. It is also, in general, a requirement that all variables are nonnegative. This is often the case in real-world problems that deal with physical quantities, but problems involving variables that may be negative as well as positive can still be modeled by rewriting each original variable as a difference of two nonnegative variables.

The function to be optimised is called the *objective function*. In the real world situation that gives rise to an optimisation problem, the function may contain a constant term, but it can be removed since that will affect all possible solutions in the same way. The objective function can then be written as $\zeta = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$, where the c_j are constants. The variables in the objective function are often called *decision variables*, since our task is not only to find the optimal value of the objective function, but also which variable values that yield this function value. Throughout this report, we will consistently use n to refer to the number of decision variables and m to refer to the number of equations

[2] *NP*-hardness is a term from complexity theory, which deals with the relative difficulties of solving different kinds of problems. The only known algorithms for solving *NP*-hard problems require an amount of time that is exponential in the size of the problem, which renders those algorithms useless for many real life problem sizes. For further reading on complexity theory, consult Garey and Johnson[12].

and/or inequalities. The variables will typically be labelled x_1 through x_n.

Standard form An LP problem is commonly called a *linear program*. The equations and inequalities that (together with the objective function) constitute an linear program may be represented in different forms. We shall first consider the *standard form*, in which only less-than-or-equal-to inequalities with all variables on the left hand side are allowed[3]. A problem containing equalities of the form $a_{i1}x_1 + \ldots + a_{in}x_n = b_i$ may be rewritten by splitting each equality into two inequalities[4]: $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ and $-a_{i1}x_1 - \ldots - a_{in}x_n \leq -b_i$. Also, the goal must be to maximise the objective function — if the original problem is to minimize some function f, we let our objective function be $\zeta = -f$. A linear program in standard form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.1}$$

with respect to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad \text{for } i = 1, \ldots, m. \tag{2.2}$$

$$x_1, \ldots, x_n \geq 0 \tag{2.3}$$

Slack form The other common representation is *slack form*, which only allows a set of equations (and a nonnegativity constraint for each variable). A slack form program should be produced by rewriting a standard form program. An inequality of the form $a_{i1}x_i + \ldots + a_{in}x_n \leq b_i$ is converted to an equation by adding a *slack variable* w_i. Together with the condition that $w_i \geq 0$, the equation $a_{i1}x_1 + \ldots + a_{in}x_n + w_i = b_i$ is equivalent to the original inequality (whose difference, or "slack", between the left and right hand sides is represented by w_i). When the program is constructed in this manner, each slack variable only appears in excactly one equation, which is an important property that will be utilised later. A linear program in slack form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.4}$$

[3]Note that strictly-less-than and strictly-greater-than inequalities are never allowed in LP problems, as they could easily cause situations in which it is impossible to achieve optimality — for instance, there is no optimal value for x with respect to $x < 3$; given any value for x that is less than 3, one can always find a number between x and 3.

[4]The drawback of doing this is that it increases the number of equations. See Hillier and Lieberman[20] for another approach, called *artificial variables* — with the drawback that it increases the number of variables.

with respect to

$$w_i = b_i - \sum_{j=1}^{n} a_{ij}x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.5}$$

$$x_1, \ldots, x_n, w_1, \ldots, w_m \geq 0 \tag{2.6}$$

— Example —

In standard form, our example is expressed as

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$x_1 + x_2 \leq 40$$
$$2x_1 + x_2 \leq 50$$
$$x_1, x_2 \geq 0$$

In slack form, it becomes

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$w_1 = 40 - x_1 - x_2$$
$$w_2 = 50 - 2x_1 - x_2$$
$$x_1, x_2, w_1, w_2 \geq 0$$

A proposed solution vector (that is, a specification of a value for each variable) of a linear program is called:

Feasible if it does not violate any constraints;

Infeasible if it violates one or more constraints (however, it is still called a "solution");

Basic if it consists of setting all variables except the slack variables to zero (so that $w_i = b_i$ for all i);

Optimal if it is feasible and no other feasible solutions yield a higher value for the objective function. An optimal solution vector is not necessarily unique, although the optimal objective function value obviously is.

2.1.2 The standard simplex method

The *standard simplex method*, or simply the *simplex method*[5], developed by George Dantzig[9] in 1949, was the first systematic approach for solving linear programs. It requires the linear program to be in slack form. The initial coefficients and constants are written down in a *tableau* that will change as the method progresses. The nonnegativity constraints are not represented anywhere; rather, they are implicitly maintained by the method. Because the equations will undergo extensive rewriting, it will be convenient not to distinguish the slack variables from the other variables, so we will relabel w_i to x_{n+i} for $i = 1, \ldots, m$. Thus, the total number of variables is $n + m$. Furthermore, we will use overlines over the coefficients in the tableau to denote their *current* value (which will change in each iteration of the simplex method), and the indices of the coefficients will refer to the coefficients' position within the tableau — for instance, $-\overline{a}_{ij}$ is located in row i, column j. We also introduce a constant term $\overline{\zeta}$ (initially zero) in the objective function, which will help us keep track of the best function value we have found so far. The topmost row and leftmost column are not really a part of the tableau; they are simply headers — the topmost row shows which variables correspond to which columns, and the leftmost column shows the slack variables for each row. The first actual tableau row (below the double line) contains the objective function coefficients $[\overline{c}_j]$ and is numbered as row 0; the first actual tableau column (to the right of the double line) contains the $[\overline{b}_i]$ constants and is numbered as column 0; the rest of the tableau contains the negatives of the coefficients from the equations: $[-\overline{a}_{ij}]$. Initially, $\overline{c}_j = c_j$, $\overline{b}_i = b_i$, and $\overline{a}_{ij} = a_{ij}$. For instance, with $n = 3$ and $m = 3$, the initial tableau will look like this:

		x_1	x_2	x_3
ζ	0	c_1	c_2	c_3
x_4	b_1	$-a_{11}$	$-a_{12}$	$-a_{13}$
x_5	b_2	$-a_{21}$	$-a_{22}$	$-a_{23}$
x_6	b_3	$-a_{31}$	$-a_{32}$	$-a_{33}$

Note that this is essentially just a tabular version of the standard form — for instance, the last row is interpreted as the equation $x_6 = b_3 - a_{31}x_1 - a_{32}x_2 - a_{33}x_3$.

[5]The reason for not calling it the "simplex *algorithm*" is that there exist several versions of the method, and that the general method formulation is somewhat underspecified because it does not say how to choose the pivot elements.

— Example —

In tableau form, our example becomes

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Note that w_1 and w_2 have been renamed to x_3 and x_4, respectively.

The variables are partitioned into two sets. The variables in the leftmost column (at the left side of the equations) are referred to as the *basic variables*, and the variables inside the tableau are called *nonbasic variables*. At any stage of the method, the set of the indices of the basic variables is denoted $\mathcal{B}$, and the set of nonbasic indices is denoted $\mathcal{N}$. Initially, $\mathcal{N} = \{1, \ldots, n\}$, and $\mathcal{B} = \{n+1, \ldots, n+m\}$. The sizes of the basic and nonbasic sets are constant, with $|\mathcal{N}| = n$ and $|\mathcal{B}| = m$. The tableau will generally look like this (if, for instance, $m = n = 3$):

		$\cdots$	$x_{j\in\mathcal{N}}$	$\cdots$
ζ	$\overline{\zeta}$	$\overline{c}_1$	$\overline{c}_2$	$\overline{c}_3$
$\vdots$	$\overline{b}_1$	$-\overline{a}_{11}$	$-\overline{a}_{12}$	$-\overline{a}_{13}$
$x_{i\in\mathcal{B}}$	$\overline{b}_2$	$-\overline{a}_{21}$	$-\overline{a}_{22}$	$-\overline{a}_{23}$
$\vdots$	$\overline{b}_3$	$-\overline{a}_{31}$	$-\overline{a}_{32}$	$-\overline{a}_{33}$

For now, let us assume that the solution that is obtained by setting all nonbasic variables to zero is feasible (which is the case only if all of the b_i are nonnegative); we will remove this restriction later. This trivial solution will provide a lower bound for the value of the objective function (namely, the constant term, $\overline{\zeta}$). We will now select one nonbasic variable x_j and consider what happens if we increase its value (since all nonbasic variables are currently zero, we cannot decrease any of them). Since our goal is to maximise the objective function, we should select a variable whose coefficient c_j in the objective function is positive. If no such variables exist, we cannot increase the objective function value further, and the current solution (the one obtained by setting all nonbasic variables to zero, so that $\zeta = \overline{\zeta}$) is optimal — we can be certain of this since linear functions do not have local maxima.

It seems reasonable to select the variable with the greatest coefficient (this is known as the *Dantzig criterion*; other rules are possible). Let us say that this variable is located in column e. Note that because we will soon start swapping variable positions, the indices of the leaving and entering variables will generally not correspond to their respective row and column numbers. For notational

convenience, we therefore let $x_{\tilde{i}}$ denote the basic variable that is located in row i, and we let $x_{\hat{j}}$ denote the nonbasic variable in column j. Then, our variable is labelled $x_{\hat{e}}$. How far can we increase this variable? Recall that each line in the tableau expresses one basic variable as a function of all the nonbasic variables; hence we can increase $x_{\hat{e}}$ until one of the basic variables becomes zero. Let us look at row i, which is now reduced to $x_{\tilde{i}} = \overline{b}_i - \overline{a}_{ie}x_{\hat{e}}$ since all nonbasic variables except $x_{\hat{e}}$ are zero. If $\overline{a}_{ie}$ is positive, the value of $x_{\tilde{i}}$ will decrease as $x_{\hat{e}}$ increases, so the largest allowable increase is limited by $\overline{b}_i$. Thus, by setting $x_{\hat{e}} = \frac{\overline{b}_i}{\overline{a}_{ie}}$, $x_{\tilde{i}}$ becomes zero. However, other equations may impose stricter conditions. By looking at all rows where $\overline{a}_{ie}$ is positive, we can determine an l such that $\frac{\overline{b}_l}{\overline{a}_{le}}$ is minimal and set $x_{\hat{e}} = \frac{\overline{b}_l}{\overline{a}_{le}}$. This will cause $x_{\tilde{l}}$ to become zero. If all $\overline{a}_{ie}$ are nonpositive, we can increase $x_{\hat{e}}$ indefinitely without any $x_{\tilde{i}}$ ever becoming negative, and in that case, we have determined the linear program to be *unbounded*; the method should report this to the user and terminate.

— Example —

Recall the tableau:

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Since 30 is the greatest objective function coefficient, we select x_1 to be increased. x_3 becomes zero if $x_1 = \frac{\overline{b}_1}{\overline{a}_{11}} = \frac{40}{1}$, and x_4 becomes zero if $x_1 = \frac{\overline{b}_2}{\overline{a}_{12}} = \frac{50}{2}$. The latter is the most restrictive constraint, so x_4 will become zero when we increase x_1.

The next step, called *pivoting*, is an operation that exchanges a nonbasic variable and a basic variable. The purpose of pivoting is to produce a new situation in which all nonbasic variables are zero (and no b_i is negative), so that we can repeat the previous steps all over again and find a new variable whose value we can increase. The nonbasic variable that was selected to be increased, $x_{\hat{e}}$, is called the *entering variable*, since it is about to enter the collection of basic variables. $x_{\tilde{l}}$, which becomes zero when $x_{\hat{e}}$ is increased appropriately, is called the *leaving variable*, since it is to be removed from said collection. Keep in mind that since $x_{\tilde{l}}$ is a basic variable, it only occurs in one equation, namely

$$x_{\tilde{l}} = \overline{b}_l - \sum_{j \in \mathcal{N}} \overline{a}_{lj} x_{\hat{j}}. \tag{2.7}$$

Note that we have retained all the nonbasic variables, as we want an equation that is valid at all times, not only when almost all nonbasic variables are zero.

We can eliminate the entering variable from (and introduce the leaving variable into) the set of nonbasic variables by rewriting (2.7):

$$x_{\tilde{l}} = \overline{b}_l - \overline{a}_{le} x_{\hat{e}} - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \overline{a}_{lj} x_{\hat{j}} \tag{2.8}$$

$$x_{\hat{e}} = \frac{1}{\overline{a}_{le}} \left(\overline{b}_l - x_{\tilde{l}} - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \overline{a}_{lj} x_{\hat{j}} \right). \tag{2.9}$$

Now that we have an expression for $x_{\hat{e}}$, we can substitute it into all of the other equations — this will eliminate $x_{\hat{e}}$ and introduce $x_{\tilde{l}}$ into the rest of the tableau. For all $i \in \mathcal{B} - \{\tilde{l}\}$, we have:

$$x_{\tilde{i}} = \overline{b}_i - \sum_{j \in \mathcal{N}} \overline{a}_{ij} x_{\hat{j}} \tag{2.10}$$

$$= \overline{b}_i - \overline{a}_{ie} x_{\hat{e}} - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \overline{a}_{ij} x_{\hat{j}} \tag{2.11}$$

$$= \overline{b}_i - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \left(\overline{b}_l - x_{\tilde{l}} - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \overline{a}_{lj} x_{\hat{j}} \right) - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \overline{a}_{ij} x_{\hat{j}} \tag{2.12}$$

$$= \left(\overline{b}_i - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \overline{b}_l \right) + \frac{\overline{a}_{ie}}{\overline{a}_{le}} x_{\tilde{l}} - \sum_{j \in \mathcal{N} - \{\hat{e}\}} \left(\overline{a}_{ij} - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \overline{a}_{lj} \right) x_{\hat{j}}. \tag{2.13}$$

A similar result will be achieved for the expression for the objective function. Although it might look complicated, it amounts to subtracting[6] $\frac{\overline{a}_{ie}}{\overline{a}_{le}}$ times the tableau row l from all other tableau rows i (and adding $\frac{\overline{c}_e}{\overline{a}_{le}}$ times row l to the objective function row), and then setting the tableau entries in column e to $\frac{\overline{a}_{ie}}{\overline{a}_{le}}$ (and to $-\frac{\overline{c}_e}{\overline{a}_{le}}$ in the objective function row). Note that because l was selected such that $\overline{a}_{le}$ was positive and $\frac{\overline{b}_l}{\overline{a}_{le}}$ was minimal, all $\overline{b}_i$ remain nonnegative; and because e was selected such that $\overline{c}_e$ was positive, $\overline{\zeta}$ cannot decrease (it will either retain its old value or increase, depending on whether or not $\overline{b}_l$ was zero).

(2.9) is the new form of the tableau row that originally corresponded to the basic variable $x_{\tilde{l}}$. The new row, which corresponds to $x_{\hat{e}}$, can be easily obtained from the old one by dividing the row by $\overline{a}_{le}$ and setting the coefficient of what is now $x_{\tilde{l}}$ to $-\frac{1}{\overline{a}_{le}}$.

Finally, we remove $\tilde{l}$ from $\mathcal{B}$ and add it to $\mathcal{N}$, and remove $\hat{e}$ from $\mathcal{N}$ and add it to $\mathcal{B}$, so that the leaving and entering variables swap positions in the new tableau. This completes the pivot operation — we again have a tableau in which all nonbasic variables can be set to zero and all b_i are nonnegative, and the entire process may be repeated.

[6] Keeping track of the signs here becomes somewhat cumbersome. Keep in mind that the tableau cell at row i, column j contains $-\overline{a}_{ij}$ (if $i, j \geq 1$).

A 3×3 tableau will look like this after one pivot with x_2 as the entering variable and x_5 as the leaving variable:

		x_1	x_5	x_3
ζ	$0 + b_2c_2/a_{22}$	$c_1 - a_{21}c_2/a_{22}$	$-c_2/a_{22}$	$c_3 - a_{23}c_2/a_{22}$
x_4	$b_1 - b_2a_{12}/a_{22}$	$-a_{11} + a_{21}a_{12}/a_{22}$	a_{12}/a_{22}	$-a_{13} + a_{23}a_{12}/a_{22}$
x_2	b_2/a_{22}	$-a_{21}/a_{22}$	$-1/a_{22}$	$-a_{23}/a_{22}$
x_6	$b_3 - b_2a_{32}/a_{22}$	$-a_{31} + a_{21}a_{32}/a_{22}$	a_{32}/a_{22}	$-a_{33} + a_{23}a_{32}/a_{22}$

— Example —

After one pivot with x_1 as the entering variable and x_4 as the leaving variable, we get the following tableau:

		x_4	x_2
ζ	750	-15	5
x_3	15	0.5	-0.5
x_1	25	-0.5	-0.5

For the next pivot operation, only x_2 can be selected as the entering variable, which causes x_3 to be selected as the leaving variable. After the pivot, the tableau looks like this:

		x_4	x_3
ζ	900	-10	-10
x_2	30	1	-2
x_1	10	-1	1

Since all objective function coefficients are now negative, we have reached an optimal solution with the value $\zeta = \overline{\zeta} = 900$. This solution value is obtained by setting the nonbasic variables (x_3 and x_4) to 0, in which case $x_1 = 10$ and $x_2 = 30$. We can easily verify that these variable values do not violate any constraints, and by substituting the values into the original objective function, we can verify that the optimal value is indeed $\zeta = 30x_1 + 20x_2 = 30 \cdot 10 + 20 \cdot 30 = 900$.

2.1.2.1 Degeneracy and cycling

A tableau is *degenerate* if some of the $\overline{b}_i$ are zero. Degeneracy may cause problems because a pivot on a degenerate row will not cause the objective function value to change, and we will not have got any closer to a solution. With severely bad luck, the algorithm may end up cycling through a number of degenerate states.

This, however, rarely happens — according to Vanderbei[47, p. 32], cycling "is so rare that most efficient implementations do not take precautions against it".

As mentioned in Footnote 5 on page 7, the general formulation of the simplex method is underspecified because it does not tell how to break ties between potential entering and leaving variables. There exist rules that guarantee that cycling will not happen; one of them, called *Bland's rule*[47, Sec. 3.4] is to break ties by always selecting the variable with the smallest subscript. There are $\binom{m+n}{m}$ possible dictionaries — each dictionary is uniquely determined by the set of basic variables, and the order of the variables is unimportant (if the rows and columns of a dictionary are permuted, it is still regarded as the same dictionary, since the same variables will be selected for pivoting). Since each step transforms one dictionary into another, the simplex method is guaranteed to terminate in at most $\binom{m+n}{m}$ steps if precautions are taken against cycling. In practice, however, the method is usually far more efficient, and algorithms that are guaranteed to run in polynomial time are allegedly only superior for very large data sets (this appears to be "common knowledge" in books about the subject, who tend not to give further references about this).

2.1.2.2 Duality

Duality is an interesting property that is exhibited by linear programs and gives rise to several variations of the standard simplex method.

Given a linear programming problem in standard form:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.14}$$

with respect to

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.15}$$

$$x_1, \ldots, x_{n+m} \geq 0 \tag{2.16}$$

its *dual problem* is formed by negating everything and interchanging the roles of rows and columns: the b_i become the objective function coefficients, the c_j become the right hand side, and the positions of the a_{ij} are transposed. Also, the xs are replaced by ys (to avoid confusion with the original problem, since the variables of the dual problems will attain different values in the course of the method). We still want to maximise, but we define the solution of the dual problem to be the negative of the maximal value (this is just a technicality to avoid expressing the problem as a minimisation).

−Maximise

$$\xi = -\sum_{i=1}^{m} b_i y_i \tag{2.17}$$

with respect to

$$y_{m+j} = -c_j + \sum_{i=1}^{m} a_{ij} y_i, \quad \text{for } j = 1, \ldots, n. \tag{2.18}$$

$$y_1, \ldots, y_{m+n} \geq 0 \tag{2.19}$$

This corresponds to negating and transposing the entire tableau. Note that the original problem is referred to as the *primal problem*, and that the dual of the dual problem is the primal problem. There are two highly interesting facts about the dual problem (see [47] for proofs):

The weak duality theorem states that any feasible solution of the dual problem will be greater than any feasible solution of the primal problem.

The strong duality theorem states that the optimal solution of the dual problem equals the optimal solution of the primal problem.

We will not utilise duality extensively, except for the Phase I method discussed below, so we do not give a thorough presentation of it. The concept is very interesting, however, and interested readers should consult Vanderbei[47, Chapter 5], who gives a more in-depth presentation, including an intuitive rationale for why the dual problem is formed in this way.

Duality can be exploited in many ways, one of which is the following: if one has a linear program where the right hand side contains negative numbers, but all objective function coefficients are nonpositive, one can form the dual program (whose right hand side will then contain only nonnegative numbers) and solve that one instead. This approach is called the *dual simplex method*, and it is usually performed without actually transposing the tableau — it just swaps the roles of the basic and nonbasic variables.

2.1.2.3 Initially infeasible problems

The method presented so far is capable of solving linear programs whose initial basic solution (the one obtained by setting all nonbasic variables to 0) is feasible. This is the case if and only if all of the b_i are nonnegative, which we cannot in general assume them to be. As mentioned in the preceding section, one can get around this if all the c_j are nonpositive, but this does not generally hold either. If we have one or more negative b_i, we get around this by introducing an *auxiliary problem* which is based on the original problem. The auxiliary problem

is formed such that it is guaranteed to have a basic feasible solution, and such that its optimal solution will provide us with a starting point for solving the original problem. It is created by subtracting a new variable x_0 from the left hand side of each equation of the original problem (which is assumed to be in standard form), and replacing the objective function with simply $\zeta = -x_0$. The purpose of x_0 is that by initially setting it to a sufficiently large value, we can easily satisfy all equations (even those having negative entries in the right hand side[7]). Then, we can try to change variable values (through regular pivoting) and see if it is possible to make x_0 equal to zero, in which case we can remove it from our equations and reinstate the original objective function, thereby having arrived at a problem that is equivalent to the original one. This is the purpose of our new objective function — since x_0, like all other variables, is required to be nonnegative, the goal of optimising $-x_0$ means that we are trying to make x_0 zero. Fortunately, we do not need a new algorithm for this optimisation process; we can use the simplex algorithm as it has been described above. We only need to do one pivot operation before we start that algorithm: since the idea of x_0 is to initially set it to a suitably large value, and since the algorithm requires a nonnegative right hand side, we should make x_0 a basic variable by performing one pivot operation with the row containing the most negative b_i. This will make the entire right hand side nonnegative. Solving the auxiliary problem is called *Phase I*, and solving the resulting problem (with the original objective function) is called *Phase II*. Thus, the full simplex method is a two-phase method (but of course, if the right hand side of the original problem is nonnegative, we can skip Phase I).

Another Phase I method, the one used by Vanderbei, is to first replace negative terms in the objective function by an arbitrary positive number (e.g. 1) and then run the dual simplex method as described above. The dual method will terminate when the original right hand side only consists of nonnegative numbers, in which case we can reinstate the actual coefficients of the original objective function and proceed with Phase II.

One-phase methods also exist, such as the *parametric self-dual simplex method*, as described in [47, Sec. 7.3].

2.1.2.4 Formal algorithm statement

In Algorithm 1 on the next page we present the pseudocode for an individual phase of the standard simplex method (with the first approach described in Section 2.1.2.3, the same code can be used for both Phase I and Phase II). The tableau is called T and is zero-indexed; keep in mind that row 0 is the objective function and column 0 contains the constants from the right hand sides of the inequali-

[7] Beware that "the right hand side" refers to the b_i, which are on the right hand side of the original equations — but in the tableau, they are on the *left* hand side.

ties. The current value of the objective function is always in row 0, column 0. We use row major indexing, so $T[2,3]$ is row 2, column 3.

loa 1: One phase of the standard simplex method using the Dantzig criterion

procedure STANDARDSIMPLEXPHASE($m,\ n,\ a[1..m, 1..n],\ b[1..m],\ c[1..n]$)
  $T[0,0] \leftarrow 0$
  $T[i,j] \leftarrow -a[i,j]$ for $i = 1 \ldots m$, $j = 1 \ldots n$
  $T[i,0] \leftarrow b[i]$ for $i = 1 \ldots m$
  $T[0,j] \leftarrow c[j]$ for $j = 1 \ldots n$
  $\mathcal{N} \leftarrow \{1, \ldots, n\}$
  $\mathcal{B} \leftarrow \{n+1, \ldots, n+m\}$
  loop
    Pick the smallest column number $e \geq 1$ such that $T[0,e]$ is positive and maximal
    if no e is found **then**
      return $T[0,0]$ as the optimal solution
    end if
    Pick the smallest row number $l \geq 1$ such that $T[l,e] < 0$ and $-\frac{T[l,0]}{T[l,e]}$ is minimal
    if no l is found **then**
      return "The problem is infeasible" (if this is Phase I) or "The problem is unbounded" (if this is Phase II)
    end if
    $p \leftarrow -T[l,e]$
    for $i \leftarrow 0, m$ **do**
      if $i \neq l$ **then**
        $f \leftarrow \frac{T[i,e]}{p}$
        Add f times row l of T to row i of T
        $T[i,e] \leftarrow -f$
      end if
    end for
    Divide row l of T by p
    $T[l,e] \leftarrow -\frac{1}{p}$
  end loop
end procedure

2.1.2.5 Complexity and numerical instability

The complexity classes *P* and *NP* should be familiar to anyone that has taken an algorithms course: *NP* is the class of decision problems (problems that are in the form of a yes/no question) where, if the answer is "yes" and we are given a "certificate" that demonstrates the solution, we can validate the solution in time that is polynomial in the size of the input. *P* is the subset of *NP* that consists of those decision problems where we can also *find* the solution in polynomial

time. The question of whether $P = NP$ remains one of the most important open questions in the field of computer science, and is one of the seven Clay Millenium Prize problems[8]. Most researchers believe that $P \subset NP$, and that the most difficult problems in *NP*, the so-called *NP-complete* (*NPC*) problems, cannot be solved in polynomial time. Cormen et al.[8] give a good introduction to complexity theory.

When dealing with parallel programming, another complexity class is also useful: *NC*, also known as *Nick's Class*. This is the class of all problems that can be solved in $O(\lg^{k_1} n)$ steps (so-called *polylogarithmic time*) using a polynomial ($O(n^{k_2})$) number of processors. Here, k_1 and k_2 are constants. *NC* is a subset of *P*, since any parallel algorithm requiring $f(n)$ steps using $p(n)$ processors can be simulated in $p(n)f(n)$ steps on a sequential computer. Thus, any *NC*-algorithm will require $O(n^{k_2} \lg^{k_1} n)$ steps on a sequential machine, and this is polynomial in n. However, there are problems in *P* which have not yet been proven to be in *NC*, and the most difficult problems among these are called *P-complete* (*PC*) — this is quite analoguous to the *NP*/*P*/*NPC* situation.

In some sense, *NC* captures the notion of what it means for a problem to be "parallelisable", while the *P*-complete problems can be said to be "hard to parallelise". However, it is not a perfect classification:

- A problem may be in *NC* without being efficiently solvable in practice due to a prohibitive processor requirement of the algorithm (for instance $O(n^{10})$ processors) or large constants hidden by the O-notation.
- Parallel algorithms for *P*-complete problems may still be useful because they might be faster than their sequential counterparts (just not "much faster").

Where does LP fit into this picture? The trivial upper bound of $O(\binom{m+n}{m})$ given in Section 2.1.2.1 for the number of iterations in the simplex method is absolutely horrible: $\binom{m+n}{m} \geq \left(\frac{m+n}{m}\right)^m = \left(1 + \frac{n}{m}\right)^m$, which, if $m = n$, becomes 2^m. Unfortunately, Klee and Minty[30] proved that it is possible to construct arbitrary-size data sets that make the method hit that bound when a certain pivoting rule is used (and no one has succeeded in finding a pivoting rule that can guarantee polynomial time). In spite of this, the method is said to often be surprisingly efficient in practice (this is stated without further reference in several books, among them [47] and [8]). In 1979, Khachiyan[29] discovered a different kind of algorithm that is guaranteed to run in polynomial time, and thus he proved LP to be in *P*.[9] However, LP is also *P*-complete, as proved by Dobkin

[8] http://www.claymath.org/millennium/

[9] Strictly speaking, LP is a computation problem (one in which we seek a numerical answer) rather than a decision problem and thus falls outside of the *NP*/*P*/*NC* discussion. However, like many other computation problems, LP easily can be reformulated as a decision problem that can be solved by the same algorithms; see [16, Problem A.4.3] for more references.

et al.[10]. Still, for the reasons mentioned above, this should not discourage us from seeking parallel versions of LP algorithms.

Greenlaw et al.[16] give a thorough presentation of *NC* and other aspects of parallel complexity, and a more compact survey of the field can be found in Natvig's Dr.Ing. thesis[40].

2.1.2.6 Warmstarting

If one has solved an LP problem and then wishes to solve a very similar problem (one that has been obtained by slightly altering the various coefficients of the original problem), it would seem reasonable to believe that the optimal solution to the original problem would be a great starting point in the search for the optimal solution to the new problem. This turns out to be the case, and the idea is known as *warmstarting*. It normally leads to a great reduction in the time required to solve the new problem, and it is also very easy to implement — the simplex method does not need to be changed at all; the program must simply be capable of taking a suggested starting solution as input. Note that one might have to run both phases, in case the original solution is not feasible for the new problem. Interested readers may consult Vanderbei[47, Chapter 7] for a more thorough introduction to the subject (which he refers to as *sensitivity analysis*).

Miriam AS employs Monte Carlo methods that produce a number of random variations of the current state of the oil pipeline network in order to predict what will happen if anything changes. This is an important reason that they want to focus on the simplex method rather than interior point methods (Section 2.1.5) — warmstarting *is* possible for the latter class of methods, but it is much harder to implement. Various approaches to warmstarting interior point methods are described by e.g. Gondzio and Grothey[15] (this is actually a more general approach for quadratic programming), Yildirim and Wright[52], and Benson and Shanno[6].

2.1.3 The revised simplex method

The *revised simplex method*, also due to Dantzig[9], is essentially just a linear algebra reformulation of the mathematical operations of the standard simplex method; however, it is much more numerically stable, for reasons that will be explained. We begin with expressing the slack form constraint tableau in matrix notation — note that all vectors are column vectors unless stated otherwise. An LP problem in slack form (with renaming of the slack variables) looks like the following:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.20}$$

with respect to

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.21}$$

$$x_1, \ldots, x_{n+m} \geq 0 \tag{2.22}$$

If we let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & & \\ a_{21} & a_{22} & \cdots & a_{2n} & & 1 & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & & & 1 \end{bmatrix} \tag{2.23}$$

$$\mathbf{b} = \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}^\top \tag{2.24}$$

$$\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_n & 0 & \cdots & 0 \end{bmatrix}^\top \ (m \text{ zeroes at the end}) \tag{2.25}$$

$$\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_n & x_{n+1} & \cdots & x_{n+m} \end{bmatrix}^\top \tag{2.26}$$

we can express the problem in a very compact manner:

Maximise

$$\zeta = \mathbf{c}^\top \mathbf{x} \tag{2.27}$$

with respect to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.28}$$

$$\mathbf{x} \geq 0 \tag{2.29}$$

In order to be able to handle the pivot operations, we will need to split each of our matrices and vectors into two in order to reflect which entries correspond to basic variables and which ones do not. As before, we let $\mathcal{N}$ be the collection of nonbasic variable indices (initially $\{1, \ldots, n\}$), and $\mathcal{B}$ the collection of basic variable indices (initially $\{n+1, \ldots, n+m\}$). All the basic variables are put in the vector $\mathbf{x}_\mathcal{B}$, and the nonbasic variables are put in $\mathbf{x}_\mathcal{N}$ — the order of the variables within these vectors do not matter, as long as the entries of the other matrices are arranged correspondingly. We split $\mathbf{A}$ into two matrices: an $m \times n$ matrix $\mathbf{N}$, containing all columns from $\mathbf{A}$ that correspond to nonbasic variables (initially, this will be all the columns containing the a_{ij} entries), and $\mathbf{B}$, which

is initially an $m \times m$ identity matrix. Similarly, we split $\mathbf{c}$ into one vector $\mathbf{c}_\mathcal{N}$ for the objective function coefficients belonging to nonbasic variables (initially, $\mathbf{c}_\mathcal{N} = [\ c_1 \quad \cdots \quad c_n\ ]^\top$) and one vector $\mathbf{c}_\mathcal{B}$ for the coefficients belonging to basic variables (initially an m element zero vector). After each pivot operation, entries of these matrices and vectors will swap *positions* according to how the collections of basic and nonbasic variables have changed, but the *values* themselves will never change during the course of the algorithm. This means that numerical stability may be significantly improved, since the matrices and vectors will not accumulate errors (practical implementations do, however, maintain additional vectors that will accumulate errors; see below). Note that the right hand side vector, $\mathbf{b}$, remains a single vector that will never change. Using these "split" matrices and vectors, we can express the problem as

Maximise

$$\zeta = \mathbf{c}_\mathcal{N}^\top \mathbf{x}_\mathcal{N} + \mathbf{c}_\mathcal{B}^\top \mathbf{x}_\mathcal{B} \tag{2.30}$$

with respect to

$$\mathbf{N}\mathbf{x}_\mathcal{N} + \mathbf{B}\mathbf{x}_\mathcal{B} = \mathbf{b} \tag{2.31}$$

$$\mathbf{x} \geq 0 \tag{2.32}$$

During execution of the (standard) simplex method, it is always the case that each basic variable occurs in exactly one equation, and hence each basic variable can be written as a function of the nonbasic variables. Therefore, $\mathbf{B}$ must be invertible, so we can multiply (2.31) by $\mathbf{B}^{-1}$ and rearrange it to get

$$\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_\mathcal{N}. \tag{2.33}$$

Combining this with (2.30), we get

$$\zeta = \mathbf{c}_\mathcal{N}^\top \mathbf{x}_\mathcal{N} + \mathbf{c}_\mathcal{B}^\top \mathbf{x}_\mathcal{B} \tag{2.34}$$

$$= \mathbf{c}_\mathcal{N}^\top \mathbf{x}_\mathcal{N} + \mathbf{c}_\mathcal{B}^\top (\mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_\mathcal{N}) \tag{2.35}$$

$$= \mathbf{c}_\mathcal{B}^\top \mathbf{B}^{-1}\mathbf{b} + (\mathbf{c}_\mathcal{N}^\top - \mathbf{c}_\mathcal{B}^\top \mathbf{B}^{-1}\mathbf{N})\mathbf{x}_\mathcal{N}. \tag{2.36}$$

This is very interesting, because we can use it to acquire explicit formulas for the simplex tableau at any time during the method, given the current basic/nonbasic variable configuration: From (2.33) (which can be rewritten as $\mathbf{B}^{-1}\mathbf{N}\mathbf{x}_\mathcal{N} + \mathbf{x}_\mathcal{B} = \mathbf{B}^{-1}\mathbf{b}$), we see that:

- The negative of the current body of the standard simplex tableau (the coefficients that in the previous subsection were known as $[-\overline{a}_{ij}]$) can be expressed as $\mathbf{B}^{-1}\mathbf{N}$.

- The "right hand side" (the leftmost column of the tableau, known as $[\overline{b}_i]$ — these are also the current values of the basic variables) is $\mathbf{B}^{-1}\mathbf{b}$.
- Similarly, we see from (2.36) that $\mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{b}$ corresponds to the current value of the objective function (obtained by setting $\mathbf{x}_{\mathcal{N}} = 0$), and the current objective function coefficients (also called the *reduced costs*) from the tableau (known as $[\overline{c}_j]$) are $\mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N}$.

How can we express a pivot operation? It turns out to be exceedingly simple — if the entering variable is $x_{\hat{e}}$ and the leaving variable is $x_{\tilde{l}}$, it is sufficient to swap column e of $\mathbf{N}$ with column l of $\mathbf{B}$. We do not even need to physically move the columns; we can implicitly move them by using permutation lists (based on $\mathbf{B}$ and $\mathbf{N}$) to keep track of which columns are located where. Strangely, Vanderbei[47] does not seem to contain a proof that pivots can be performed in this manner (for the most part, the book is burgeoning with useful proofs). For the sake of completeness, we provide here a (not entirely rigorous) demonstration that at least the first pivot will work. $\mathbf{B}$ was initially an identity matrix, and $\mathbf{N}$ was $[\overline{a}_{ij}]$. We now perform a pivot where $x_{\hat{e}}$ enters and $x_{\tilde{l}}$ leaves — this will cause column e from $\mathbf{N}$ to be swapped with column l from $\mathbf{B}$. Let us assume that the former column does not contain any zeroes (if it does, its inverse will look different); then we have the following (column l of $\mathbf{B}$ and $\mathbf{B}^{-1}$ and column e and row l of $\mathbf{N}$ have been emphasised):

$$\mathbf{B}^{-1} = \left[\begin{array}{ccc|c|ccc} 1 & & & a_{1e} & & & \\ & \ddots & & \vdots & & & \\ & & & a_{le} & & & \\ & & & \vdots & \ddots & & \\ & & & a_{me} & & 1 & \end{array}\right]^{-1} = \left[\begin{array}{ccc|c|ccc} 1 & & & -a_{1e}/a_{le} & & & \\ & \ddots & & \vdots & & & \\ & & & 1/a_{le} & & & \\ & & & \vdots & \ddots & & \\ & & & -a_{me}/a_{le} & & 1 & \end{array}\right]$$

and

$$\mathbf{B}^{-1}\mathbf{N} = \left[\begin{array}{ccc|c|ccc} 1 & & & -a_{1e}/a_{le} & & & \\ & \ddots & & \vdots & & & \\ & & & 1/a_{le} & & & \\ & & & \vdots & \ddots & & \\ & & & -a_{me}/a_{le} & & 1 & \end{array}\right] \left[\begin{array}{ccc|c|ccc} a_{11} & \cdots & & 0 & \cdots & a_{1n} \\ \vdots & & & \vdots & & \vdots \\ \hline a_{l1} & \cdots & & 1 & \cdots & a_{ln} \\ \hline \vdots & & & \vdots & & \vdots \\ a_{m1} & \cdots & & 0 & \cdots & a_{mn} \end{array}\right]$$

$$= \left[\begin{array}{ccc|c|ccc} a_{11} - a_{l1}a_{1e}/a_{le} & \cdots & & -a_{1e}/a_{le} & \cdots & a_{1n} - a_{ln}a_{1e}/a_{le} \\ \vdots & & & \vdots & & \vdots \\ \hline a_{l1}/a_{le} & \cdots & & 1/a_{le} & \cdots & a_{ln}/a_{le} \\ \hline \vdots & & & \vdots & & \vdots \\ a_{m1} - a_{l1}a_{me}/a_{le} & \cdots & & -a_{me}/a_{le} & \cdots & a_{mn} - a_{ln}a_{me}/a_{le} \end{array}\right]$$

Compare this to the tableau on page 11 (where $m = n = 3$ and $e = l = 2$) — its main body is the exact negative of this matrix, as expected. Similar derivations can be carried out for the right hand side and for the objective function coefficients.

Armed with this knowledge, we can formulate the revised simplex method, as shown in Algorithm 2. Note that, like the standard simplex method, it may also require two phases, and it is still necessary to specify a way of selecting the entering variable.

loa 2: One phase of the revised simplex method

procedure REVISEDSIMPLEXPHASE(m, n, $\mathbf{N}$, $\mathbf{c}_\mathcal{N}$, $\mathbf{b}$)
  Let $\mathbf{c}_\mathcal{B}$ be an m element zero vector
  Let $\mathbf{B}$ be an $m \times m$ identity matrix
  Let $\mathbf{B}^{-1}$ be an $m \times m$ identity matrix
  $\mathcal{N} \leftarrow \{1, \ldots, n\}$
  $\mathcal{B} \leftarrow \{n+1, \ldots, n+m\}$
  loop
    $\hat{\mathbf{c}}_\mathcal{N}^\top \leftarrow \mathbf{c}_\mathcal{N}^\top - \mathbf{c}_\mathcal{B}^\top \mathbf{B}^{-1}\mathbf{N}$ ▷ Compute the reduced costs
    Search $\hat{\mathbf{c}}_\mathcal{N}$ for a negative number; let e be its index (the corresponding nonbasic variable is then $x_{\hat{e}}$)
    if no negative number found in $\hat{\mathbf{c}}_\mathcal{N}$ **then**
      return $\mathbf{c}_\mathcal{B}^\top \mathbf{B}^{-1}\mathbf{b}$, $\mathbf{B}^{-1}\mathbf{b}$ ▷ Optimal value and basic variable values
    end if
    Let $\mathbf{N}_e$ be the eth column of $\mathbf{N}$ (the one corresponding to $x_{\hat{e}}$)
    $\hat{\mathbf{a}} \leftarrow \mathbf{B}^{-1}\mathbf{N}_e$ ▷ Compute the tableau coefficients of $x_{\hat{e}}$
    $\hat{\mathbf{b}} \leftarrow \mathbf{B}^{-1}\mathbf{b}$ ▷ Compute the basic variable values
    Let l be a value of i that minimises $t = \frac{\hat{\mathbf{b}}_i}{\hat{\mathbf{a}}_i}$ (only perform this calculation for those $i \in \mathcal{B}$ where $\hat{\mathbf{a}}_i$ is positive)
    if no value is found for l **then**
      return "The problem is unbounded"
    end if
    Exchange the eth column of $\mathbf{N}$ with the lth column of $\mathbf{B}$
    $\mathcal{B} \leftarrow (\mathcal{B} - \{\tilde{l}\}) \cup \{\hat{e}\}$
    $\mathcal{N} \leftarrow (\mathcal{N} - \{\hat{e}\}) \cup \{\tilde{l}\}$
    Recalculate $\mathbf{B}^{-1}$ from $\mathbf{B}$
  end loop
end procedure

This method looks problematic in that it seems to require $\mathbf{B}$ to be inverted in every single iteration. However, it turns out that since only one column of $\mathbf{B}$ changes in iteration, each $\mathbf{B}^{-1}$ can be calculated from the previous one by changing one column; furthermore, this change can be expressed as a multiplication with a sparse matrix formed in a certain way. A chain of such matrices is called an *eta file*, and this approach is described in greater detail in [47, Section

8.3]. Of course, the longer the eta file gets, the slower the calculation will become, and inaccuracies may accumulate. Therefore, with regular intervals, $\mathbf{B}^{-1}$ should be recomputed from scratch from the current version of $\mathbf{B}$. This will also eliminate the inaccuracies (unless $\mathbf{B}$ is ill-conditioned, in which case one may run into problems). Note that it is possible to update $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}_{\mathcal{N}}$ in each iteration rather than to recalculate them (this is the approach taken by [47]), but the update calculations are also time consuming.

2.1.4 ASYNPLEX

As we will describe in Section 3.3, even getting the sequential standard simplex method to work turned out to be very hard. With time becoming scarce, we realised that we most likely would not be able to develop an algorithm of our own for a parallel revised simplex method, and so we started looking for existing algorithms. We did not find many, and the most promising one (in particular because it bears a strong resemblance to the original revised simplex method) is called ASYNPLEX, and was developed by Hall and McKinnon[19]. It is an asynchronous algorithm[10] for message-passing systems, but the authors also describe a shared-memory version of the algorithm. We will now present ASYNPLEX, based on [19].

Before proceeding, we should mention that on the coarsest level, one can distinguish between two ways of achieving parallelism:

Task parallelism can be achieved when two or more different operations can be performed in parallel.

Data parallelism can be achieved when the same operation is applied to several related data elements.

The extent to which the different parts of the computation are independent will greatly affect the possibilities for speedup. Computations that can be split into parts that are entirely independent are called *embarrassingly parallel* (see Section 2.3.3 for more information on this), and such computations will benefit greatly from parallelisation (unless the computation is so simple that the time spent distributing the data to the different processors exceeds the time saved on the computation). Unfortunately, many important problems are not embarrassingly parallel because one computation may depend on an intermediate result from another computation (if, on the other hand, it depends on the *final* result, it cannot be said to be parallelisable).

[10] In a synchronous algorithm, the code contains synchronisation points where two or more processes or threads must wait for each other to reach the point before proceeding. In asynchronous algorithms, the only kind of waiting that may occur is waiting for incoming messages from other processes or threads.

ASYNPLEX can be regarded as a task parallel algorithm in which there are four different kinds of processes:

- One *invert processor*;
- One *basis change manager*;
- One *column selection manager*;
- One or more *iteration processes*.

We will interleave their descriptions with the description of the general idea behind the algorithm.

Matrix inversion

The *invert processor* is continuously performing inversions of the $\mathbf{B}$ matrix. Whenever one of the iteration processes performs a pivot operation, it sends a message to the invert processor telling which variable entered and which one left. Once the invert processor finishes the current inverse calculation, it distributes the resulting $\mathbf{B}^{-1}$ matrix to the iteration processes. Then, it receives all incoming basis change messages and begins a new inverse calculation. Most likely, the iteration processes will find that the inverse is somewhat out of date when it is received, but they will just delete the appropriate number of entries from the eta file. This approach sacrifices some numerical stability for the increase in speed that is obtained by dedicating a separate processor to the inversion operation. See Section 3.4.1 for a small discussion of what happens if this approach is used on its own, without the other elements of ASYNPLEX.

Candidate persistence

With the exception of the matrix inversion, the revised simplex method seems to be poorly suited for task parallelism, because each pivot operation seems to depend on the previous one — this, however, turns out to only be partially true. The key observation upon which ASYNPLEX is based is a phenomenon called *candidate persistence*. An *attractive candidate* is a nonbasic variable whose objective function coefficient is negative, so that it is possible to select it as the entering variable. According to [19], a variable that is attractive in one iteration (but remains nonbasic because some other variable is eventually selected as the entering variable) will often remain attractive in subsequent iterations. Furthermore, it can be observed that the pivot operation itself is usually very cheap (assuming that the implementation swaps matrix columns implicitly by using permutation lists to keep track of the current location of each column, while the columns themselves remain in one place) — the majority of the work in each iteration is associated with determining the entering and leaving variables and updating

the solution vector. This leads to the idea of having several processes (the iteration processes) speculatively computing the $\hat{\mathbf{a}}$ (see Algorithm 2) corresponding to several attractive candidates. When an iteration process has completed the calculation of $\hat{\mathbf{a}}$, it sends to the basis change manager an offer to compute the leaving variable and perform the pivot operation. Given any basis, only one iteration process should be allowed to decide how to pivot away from it (otherwise, the iteration processes would diverge in different directions), and the basis change manager handles this. If the offer is accepted, the iteration process will tell all other processes which variable left and which one entered, and the other iteration processes will update their $\mathcal{B}$ and $\mathcal{N}$ accordingly. It also computes a new set of attractive candidates. Iteration processes that have had their offers rejected will request new variables from the column selection manager, which keeps track of which variables are currently regarded as attractive.

The pseudocode uses some overly compact names (that probably stem from some old naming convention; Maros[37] uses them too) for each step of the algorithm; they are as follows (taken from [19], with some modifications):

BTRAN Compute $\pi^\top \leftarrow \mathbf{c}_\mathcal{B}^T \mathbf{B}^{-1}$ (in the process, we will use the eta file entries in reverse order, hence the name *Backwards TRANsformation*[37]).

PRICE Compute the reduced costs: $\hat{\mathbf{c}}_\mathcal{N}^\top \leftarrow \mathbf{c}_\mathcal{N}^\top - \pi^\top \mathbf{N}$.

CHUZC Choose entering variable (*Column*) by finding a negative entry in $\hat{\mathbf{c}}_\mathcal{N}$.

FTRAN Compute $\hat{\mathbf{a}} \leftarrow \mathbf{B}^{-1}\mathbf{a}_q$, where $\mathbf{a}_q$ is the column of $\mathbf{N}$ that corresponds to the entering variable (this time, the eta file will be used forwards, hence *Forwards TRANsformation*).

CHUZR Choose leaving variable (*Row*) by looking at the componentwise ratios of $\hat{\mathbf{b}}/\hat{\mathbf{a}}$, where $\hat{\mathbf{b}} \leftarrow \mathbf{B}^{-1}\mathbf{b}$. Let α be the smallest such ratio.

UPRHS Update the right-hand side by adding $\alpha\hat{\mathbf{a}}$ to $\hat{\mathbf{b}}$.

UPDATE_BASIS Add an entry to the eta file.

We now present the pseudocode for ASYNPLEX as it is given by Hall and McKinnon[19] (with a few notational adaptations), in Algorithms 3, 4, 5, and 6. It is assumed that there is a separate, sequential piece of code that handles input reading and sets up the different processes. In Section 3.4.2, we describe how we have adapted the algorithm.

A short explanation of Hall's notation may be useful. Each process has a number of points where it sends or receives data to or from the other processes. Each such communication endpoint is given a short identifying tag, both on the sending and receiving end, and each send or receive operation indicates where it wishes to send to or receive from (and the process' own tag for that operation

is indicated with a comment in the right margin — note also that each type of process has its own letter). Iteration process tags are suffixed with a colon and the index of the process, since there can be several iteration processes.

2.1.5 Interior point methods

It is possible to interpret the simplex method in a geometric fashion: with n decision variables, the space of all vectors of possible decision variable values form an n-dimensional space. Each constraint can be modelled as a plane in this space — an equality constraint requires that feasible solutions lie on the plane, and an inequality constraint requires that feasible solutions lie on or to one of the sides of the plane. Together with the planes from the implicit nonnegativity constraints, this forms a geometrical shape known as a *simplex* — hence the name of the simplex method. Each intermediate solution produced by the simplex method represents a point that is a vertex (an intersection between n or more planes). There exists another class of algorithms called *interior point methods*, whose intermediate solutions always lie in the interior of the simplex. A distinct advantage of most interior point methods over the simplex method is that they have polynomial worst-case bounds on their time consumption. The first polynomial interior point method was invented by Khachiyan[29] in 1979, and one of the most well-known methods is due to Karmarkar[27].

Interior point methods were mentioned in the problem description, but it was soon discovered that the scope of the project was already large enough even when only considering the simplex methods. Thus, interior point methods will not be taken into consideration, but we felt that no discussion of linear programming would be complete without mentioning this subject.

2.1.6 State of the art: sequential LP solvers

We now present some existing sequential solvers that we have studied.

ILOG CPLEX

CPLEX, developed by the company ILOG (`http://www.ilog.com/products/cplex/`), is a widely used mathematical optimisation package, and also the one currently used by Miriam AS. Being proprietary closed-source software, we cannot examine its inner workings (but they are probably too complex for this project).

GLPK

The *Gnu Linear Programming Kit* is an open source initiative to produce a versatile suite of mathematical optimisation tools. Unfortunately, the code base is

loa 3: ASYNPLEX — iteration process number i ($0 \leq i < p$)

```
1: procedure RunIterationProcess(i, p, N, b, c)
2:     k_i ← 0
3:     BTRAN
4:     PRICE
5:     FTRAN — let q be the ith most attractive candidate column, or -1 if that does not exist
6:     repeat
7:         if received ← V2 an LU factorisation of the inverse then ▷ I1
8:             Install new inverse
9:         end if
10:        while basis changes received ← I7 are not yet applied do ▷ I2
11:            Apply basis change; k_i ← k_i + 1
12:        end while
13:        Permute column a_q
14:        FTRAN
15:        while basis changes received ← I7 are not yet applied do ▷ I3
16:            Apply basis change
17:            FTRAN_STEP; k_i ← k_i + 1
18:        end while
19:        if q = −1 or ĉ_q > 0 then
20:            Send → C4 a message that the candidate is unattractive ▷ I4
21:        else
22:            Send → R1 an offer to perform CHUZR ▷ I5
23:            Wait ← (R2 or R3) for a reply to offer ▷ I6
24:            if Offer accepted then
25:                CHUZR
26:                Send → (I2/I3/I10 on all other iteration processes) the basis change and pivotal column ▷ I7
27:                Send → (V1 and C1) basis change ▷ I8
28:                UPDATE_BASIS; k_i ← k_i + 1
29:                BTRAN
30:                Permute π
31:                PRICE
32:                FTRAN — choose a set of the most attractive candidates
33:                Send → C2 the most attractive candidates ▷ I9
34:            else
35:                Wait ← I7 for next basis change ▷ I10
36:                goto line 15
37:            end if
38:        end if
39:        Wait ← (C3 or C5) for a new candidate column, q ▷ I11
40:    until The algorithm terminates
41: end procedure
```

loa 4: ASYNPLEX — invert processor

```
1: procedure RUNINVERTPROCESSOR(p, m, N)
2:     Let B be an m × m identity matrix
3:     k_v ← 0
4:     repeat
5:         while received ← I8 a notification that x_l has left the basis and x_e has entered do ▷ V1
6:             Swap the corresponding columns between B and N
7:             k_v ← k_v + 1
8:         end while
9:         INVERT
10:        Send → I1 on all p iteration processes the new LU factorisation of the inverse for basis k_v ▷ V2
11:    until the algorithm terminates
12: end procedure
```

loa 5: ASYNPLEX — column selection manager

```
1: procedure RUNCOLUMNSELECTIONMANAGER(m, n)
2:     k_c ← 0
3:     Mark all nonbasic variables as unselected
4:     repeat
5:         if received ← I8 basis change then ▷ C1
6:             Mark the variable which has left the basis as unselected
7:         else if received ← I9:i a set of candidates corresponding to basis k_i then ▷ C2
8:             if k_i > k_c then
9:                 Filter out the candidates already selected and those already rejected after the FTRAN at a basis ≥ k_i
10:                k_c ← k_i
11:            end if
12:            Send → I11:i the most attractive candidate to enter the basis and mark the candidate as selected ▷ C3
13:        else if received ← I4:i a message that its current candidate is now unattractive then ▷ C4
14:            Send → I11:i the most attractive candidate to enter the basis and mark the candidate as selected ▷ C5
15:        end if
16:    until the algorithm terminates
17: end procedure
```

loa 6: ASYNPLEX — basis change manager

```
1:  procedure RunBasisChangeManager
2:      k_b ← 1
3:      repeat
4:          if received ← I5:i an offer to perform CHUZR for basis k_i then   ▷ R1
5:              if k_i = k_b then
6:                  Send → I6:i an acceptance of the offer                    ▷ R2
7:                  k_b ← k_b + 1
8:              else
9:                  Send → I6:i a refusal of the offer                        ▷ R3
10:             end if
11:         end if
12:     until the algorithm terminates
13: end procedure
```

extremely large, comprising more than 75000 lines of C code distributed across nearly 100 files. While only a handful of these files contain functionality that is directly related to the simplex method, reverse engineering it still would be a daunting task — especially given that their coding convention apparently calls for very short variable names.

GLPK is released by its authors under version 3 of the GNU General Public License.

retroLP

As opposed to virtually all other LP solvers, retroLP[50] implements the original simplex method, not the revised method. The former is advantageous for dense problems, which occur in some special applications such as "wavelet decomposition, digital filter design, text categorization, image processing and relaxations of scheduling problems."[51] As compared to GLPK, the code is fairly short and readable — but it still consists of around 6000 lines.

retroLP is released by its authors under version 2 of the GNU General Public License.

Numerical Recipes

The Numerical Recipes is a well-known book containing source code for the numerical solution to problems in linear algebra, differential equations and many other fields. They also provide linear algebra solvers — in the second edition[42], they use the standard simplex method, while in the third edition[43], they use the revised simplex method. For reasons to be discussed in Section 3.3, we suspect that this is due to numerical stability problems, but we found no mention of such in the book.

Vanderbei's code

Vanderbei has published a freely available implementation of the revised simplex algorithm and three other algorithms that are presented in his book[47], at http://www.princeton.edu/~rvdb/LPbook/. While it comprises more than 20000 lines of source code, the core parts are fairly short and well separated from the rest of the code (much of which deals with different input formats). The code for the revised simplex methods alone is "only" around 7000 lines.

The code has no licence information attached to it. Anyone who wishes to commercially utilise those parts of our code that are derived from Vanderbei's code are strongly advised to contact Vanderbei.

Others

Here follows a few more solvers we are aware of. We have not had the time to study them thoroughly, but we list them here for those who might be interested in doing so.

Xpress — a commercial product, available at http://www.dashoptimization.com/home/products/products_optimizer.html;

OOPS — http://www.maths.ed.ac.uk/~gondzio/parallel/solver.html;

COIN-OR Linear Program Solver (CLP) — http://www.coin-or.org/Clp/;

SoPlex — An implementation developed as a part of Roland Wunderling's Ph.D. thesis[49], and available at http://soplex.zib.de/.

2.1.7 State of the art: parallel LP solvers

Parallel LP solvers also exist. ASYNPLEX[19] has already been discussed in greater detail, and here is a short list of some of the other solvers we have found.

Parallelisation of the revised simplex method using CUDA (Spampinato)

Compute Unified Device Architecture (CUDA) is a framework from the graphics processing unit (GPU) manufacturer nVidia. Mr. Daniele Spampinato, a student at our department, implemented the revised simplex method by using the CUBLAS linear algebra library to offload the linear algebra computations onto the GPU[46]. He reported overall speedups of 2.0–2.4 relative to a sequential implementation using ATLAS, but only for dense data sets. The only operation that (by itself) yielded the vast speedups that are theoretically possible when using GPUs (which have hundreds of cores) was the basis inversion[46, Figure

5.6 on p. 45]. Furthermore, he experienced major problems with numerical stability. Note that his implementation parallelised each linear algebra operation individually; it was not a parallel version of the simplex method itself.

SMoPlex, DoPlex (Wunderling)

These are, respectively, shared memory and distributed memory implementations of the revised simplex method, also from Wunderling's thesis[49]. Regrettably, these implementations are not available online, and since the thesis is written in German, we have not been able to study it — but it may prove useful to someone proficient in German. According to Hall[19], the implementation is "parallel for only two processors".

Parallelisation of interior point methods

Those interested in interior point methods should consult Karypis et al.[28] for an approach that allegedly scales to hundreds of processors.

retroLP

See above for a general description of retroLP, which also implements a parallel version of the standard simplex method.

Others

Again, here are some other papers and implementations we are aware of, but have not studied.

- Prior to ASYNPLEX, Hall and McKinnon developed another parallel revised simplex algorithm, called PARSMI[18];
- A distributed simplex algorithm by Ho and Sundarraj[21];
- A parallelisation of CPLEX' dual simplex method by Bixby and Martin[7];
- A parallelisation of Xpress' interior point method by Andersen and Andersen[3].

2.2 Cell Broadband Engine

The *Cell Broadband Engine* (Cell/BE) is a single chip multiprocessor architecture jointly developed by IBM, Sony and Toshiba. The initial design goals were to create an architecture that would be suitable for the demands of future gaming and multimedia applications (meaning not only high computational power, but also high responsiveness to user interaction and network communications),

with a performance of 100 times that of Sony PlayStation 2[26]. Several obstacles to such goals exist; in particular the infamous *brick walls*[5] (which are much of the reasons why the popularity of parallel computing has been increasing over the past years):

Memory wall While processor speeds have grown substantially over the past few decades, the growth in memory access times has been much more modest. Because of this, the relative cost of memory accesses is now prohibitively large, and for efficient scientifice computation, it is necessary to use caches and try to keep data cached for as long as possible once it has been loaded from memory.

Power wall Heat dissipation becomes a greater and greater obstacle as clock frequency increases.

ILP wall *Instruction-level parallelism* techniques such as pipelines and speculative execution face diminishing returns as most programs have a limited amount of exploitable parallelism, and the hardware and power cost of implementing such techniques is growing.

The Cell/BE architecture tries to solve these problems in the following ways:

- Having two different kinds of cores: one optimised for control logic and operating systems, and one optimised for computational throughput.
- Giving the programmer explicit control over data movement in the memory hierarchy, rather than having hardware-controlled caches.
- Providing an extensive instruction set for letting the programmer manually specify instruction-level parallelism.

The above lists and most of this section are based on the article by Kahle et al.[26] and on technical documentation from IBM, primarily [24]. Those interested in more architecture and programming details may also want to consult [25] and [23].

2.2.1 Architecture

2.2.1.1 Overview

The Cell/BE consists of one *PowerPC Processor Element* (PPE) and eight *Synergistic Processing Elements* (SPE), connected by a high-speed bus called the *Element Interconnect Bus* (EIB), as shown in Figure 2.1 on the next page.

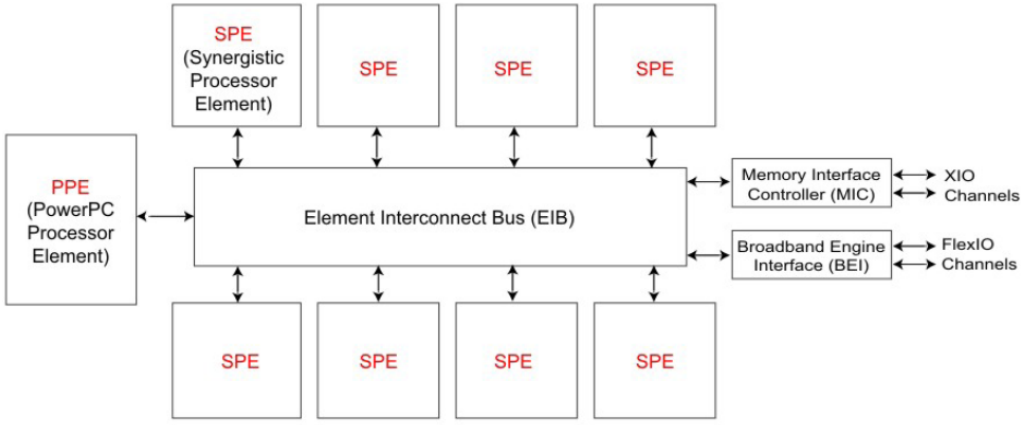

Figure 2.1: The Cell/BE architecture. Taken from [24].

2.2.1.2 PPE

The PPE (see Figure 2.2 on the facing page) is a 64 bit PowerPC, which is a general purpose RISC (reduced instruction set) architecture. Its role is that of the "control logic" core mentioned above, and it is responsible for running the operating system and controlling the rest of the Cell/BE. The PPE is again subdivided into the *PowerPC Processor Unit* (PPU[11]) (the actual PowerPC core, which supports two simultaneous threads), and the PowerPC Processor Storage Subsystem (PPSS). The latter contains the level 2 cache, arbitrates the EIB, and communicates with I/O devices. Among the most important features of the PPE is its support for the *Vector/SIMD Multimedia Extension*, an instruction set for operating on multiple values simultaneously (see Section 2.2.2.1). Further specifications can be found in [26] and [23].

2.2.1.3 SPE

Each of the eight SPEs is an independent processor that contains a *Synergistic Processor Unit* (SPU) (the actual core, with a RISC architecture and a deep pipeline) and a *Memory Flow Controller* (MFC) (see Figure 2.3 on the next page). The SPU contains a *Local Store* — 256 kB of high-speed memory. Besides the unified register file (which contains 128 128-bit registers), this is the only memory to which the code executing on the SPU has direct access — this is probably the most distinguishing feature of the entire Cell/BE architecture. Furthermore, the LS must be shared between code and data, and there is no write protection of the code area, so great care must be taken not to overwrite the code (in par-

[11] We have failed to see any system for when IBM's own documentation or any other technical documentation uses the terms PPE/SPE rather than PPU/SPU or the other way around. In this report, we have tried to stick to PPE/SPE, but several of our code files are labelled PPU/SPU.

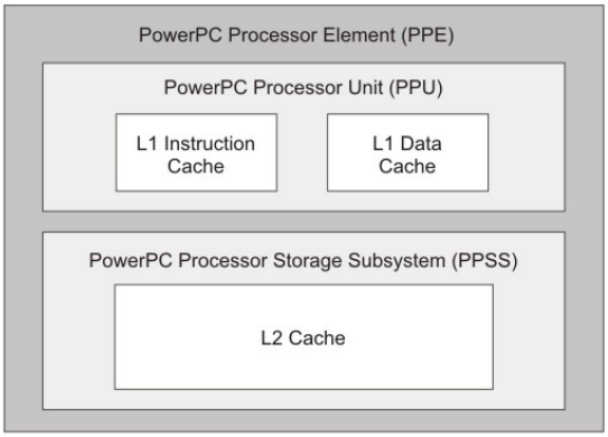

Figure 2.2: The architecture of the PPE. Taken from [24].

ticular because the address space of the LS is cyclic — any address is reduced modulo 2^{16}). Whenever the SPU needs data from system memory, that data must be transferred using DMA (Direct Memory Access — transferring data from memory to another unit attached to the bus, without going through the PPU). The MFC is responsible for handling DMA requests, and it can support several outstanding DMA requests, each with a memory area length of at most 16384 bytes. Base addresses (both in local storage and in system memory) for all DMA transfers must be aligned on a 16-byte (quadword) boundary, and the data to be transferred must be a multiple of 16 bytes. Performance is improved if entire aligned cache lines (128 bytes) are transferred at a time.

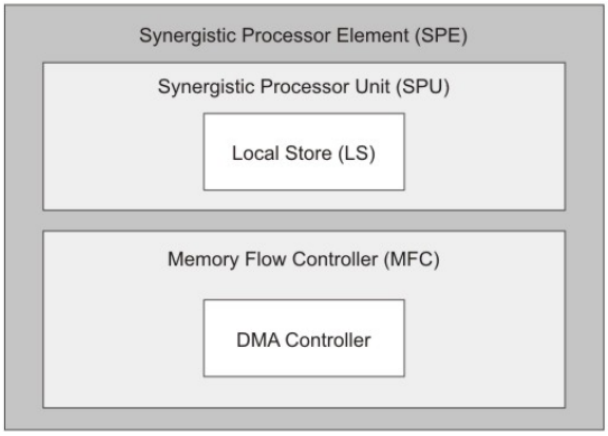

Figure 2.3: The architecture of an SPE. Taken from [24].

The idea of this architecture is that it shall be easy to overlap communication

with computation, since DMA requests can be asynchronous. It does, however, greatly add to the programming complexity. We feel that Kurzak and Dongarra expressed the spirit of the Cell/BE architecture very eloquently: "Great effort has been invested throughout the years in optimizing code performance for cache-based systems, in most cases leading to the programmers reverse engineering the memory hierarchy. By requiring explicit data motion, the memory design of the Cell takes the guesswork out of the equation and delivers predictable performance."[36].

2.2.2 Programming methods

2.2.2.1 SIMD processing

Both the PPE and the SPE support Single Instruction Multiple Data (SIMD) operations through a data type modifier called **`vector`**. A **`vector`** is 128 bits long, and can hold e.g. four **`int`**s, four **`float`**s, or two **`double`**s. **`vector`**s must start at addresses that are divisible by 16. Declarations of automatic variables (those located on the stack) may be suffixed with `__attribute__((aligned(16)))` to indicate such alignment. Memory allocated on the heap can be aligned by using `malloc_align()` — note that the second argument to this function is the base two logarithm of the byte boundary, so to get alignment on 16 byte boundaries, we would have to pass `4`.

There is a rich instruction set for operating on **`vector`**s; see [23] for a full overview. All SIMD instructions are available in C and C++ as compiler intrinsics. An example is the *fused multiply-add* operation: `d = vec_madd(a, b, c)` (where all variables are **`vector`**s), will set each component of `d` to the componentwise sum of `c` and the componentwise product of `a` and `b`. Without the SIMD instructions, this would have had to be done with arrays and a loop: **`for`** `(`**`int`** `i = 0; i < 4; ++i){ d[i] = a[i] * b[i] + c[i]; }`.

2.2.2.2 Compiler directives

There are several compiler directives that the programmer can employ to aid the compiler and the hardware in making good decisions. Two of the most interesting ones are related to branch prediction and avoidance (given the deep pipeline of the SPEs, branch mispredictions are very expensive):

1. `__builtin_expect(expression, expected)` will evaluate and return `expression` while informing the compiler that the programmer expects the result to be `expected`. This is typically placed in the condition of an **`if`**/**`else`**.

2. If the condition of an **`if`**/**`else`** is not easily predictable, but the **`if`**/**`else`** bodies are very simple, one might be better off by computing both bodies

and using a special selection instruction to determine which result will be kept. `spu_sel(a, b, condition)` will return either `a` or `b` depending on the truth value of `condition`, but is translated to instructions that do not involve branches.

2.2.2.3 PPE-SPE communication and synchronisation

It is possible for each SPE to send small (32 bit) messages to the PPE and vice versa, through an MFC-controlled mechanism called *mailboxes*. Each SPE's MFC contains an outbound queue to which the SPE can write a message. If the queue is full and the SPE tries to write another message, it will stall until the PPE has read the previous one (messages from different SPEs do not interfere with each other, however). The MFC also contains an inbound queue to which the PPE can write up to four messages, and from which the corresponding SPE can read. If the queue is full, the last message will be overwritten. It is possible to check the status of a mailbox before writing to it, thus allowing mailboxes to be used as a synchronisation mechanism.

Another method that is available for communication between the cores is *signals*, but we will not cover it here since we do not use them in our code.

2.2.2.4 Double, triple and quadruple buffering

For most applications, data sets of realistic size will not fit in the small LS. One must then employ either double, triple, or quadruple buffering. In this project, we expect to need triple or quadruple buffering, in which the data area of the LS is divided into three or four segments (there is no hardware support for this division; the program must handle the three buffers manually — note that the emphasised words are our own terms):

- The *incoming* buffer is in the process of being filled (by a DMA request) with data the SPE is about to need.

- The *work* buffer is being manipulated by the actual computation operations. This assumes that the output of the communication can be written back to the same area where the input was located; otherwise, a four-buffer scheme is needed with separate input/output buffers for the computation.

- The *outgoing* buffer contains the results of the most recently completed computation, and the data is in the process of being sent to main memory by using DMA.

When all three operations (incoming DMA, computation, and outgoing DMA) have completed, the buffer pointers are being swapped so that the computation

can continue with the recently received data, the results of the recently completed computation can be sent back to main memory, and the old outgoing buffer can be used to receive new data.

2.2.2.5 Overlays

Very large SPE programs are problematic because the small LS must be shared between code and data. *Overlays* is a mechanism where the programmer divides the code into regions of more or less independent code which will not be needed simultaneously. At execution time, the code segments will be dynamically loaded into and unloaded from the LS, based on what code is being executed. This allows programs of arbitrary size to be executed on the SPEs (unless there are extremely large functions, as each function must be fully contained within one region), at the cost of the time and bus bandwidth that is needed for the code segment transfer. One should strive to structure the segments such that segment loading will occur as infrequently as possible.

2.2.3 Tools and libraries

There exist several libraries for easing the development of scientific applications on the Cell/BE. We now provide a very quick survey of the libraries that seemed to be the most relevant to us. Only one of them, the Cell Messaging Layer, was eventually used; we discuss our choice in Section 3.4.5.1.

The Cell Messaging Layer

There are numerous articles about Cell/BE implementations of the popular Message Passing Interface (MPI), e.g. Kumar et al.[34] and Krishna et al.[33]. However, the only implementation we could find the source code for was *The Cell Messaging Layer* (CML); it is located at `http://www.ccs3.lanl.gov/~pakin/software/cellmessaging/`. The current version implements only a subset of MPI: the synchronous point-to-point communication primitives and the collective primitives (broadcast, scatter, etc.). In addition, it supports remote procedure calls (RPC) so that the SPEs can invoke code on the PPE. This library may facilitate the implementation of ASYNPLEX, which is an algorithm for message passing systems.

Others

The following are libraries that we have not used, but which may be interesting to others.

Accelerated Library Framework (ALF) is IBM's own framework for easing the development of Cell/BE applications. We believe that ALF could have

been useful to us (among other things, it provides automatic triple buffering), but it is a fairly large framework, and we never got the time to study it properly.

BlockLib [1] is a collection of *skeletons*, which are small "building blocks" (essentially functions) that are implemented in a parallel manner, and which can be combined into larger programs.

IBM Cell/BE BLAS library is a full implementation of the BLAS interface for linear algebra libraries (`http://www.netlib.org/blas/blast-forum/blas-report.pdf`), but only some functions utilise the SPEs, according to [25].

OpenMP has been implemented for Cell/BE[48]. It lets the programmer annotate the source code with **`#pragma`**s in order to indicate where there is opportunities for parallelism. A special compiler then generates PPE and SPE programs that cooperate on performing the desired computations.

RapidMind (`http://www.rapidmind.net/`) is a commercial platform for developing software that can run without modification on several different multicore platforms, including Cell/BE.

Cell Superscalar (CellSs) [41] is similar to OpenMP.

2.3 Miscellaneous topics

2.3.1 Data sets

`netlib`[13] is a collection of 98 LP problem sets, many of which stem from real life problems. This is our primary source for data sets to test our solvers with. They range in size from 28×32 to 16676×15695, and are mostly fairly sparse (the set with the biggest dimensions contains 74004 nonzeroes). Some of them have special properties, such as being extremely degenerate. All sets are represented in the MPS format; see Section 3.2.3 for a brief description and further references.

The official `netlib` site is `http://www.netlib.org/lp/index.html`, but it contains compressed files versions that must be decompressed with a fortran program. The collection is available in more convenient formats elsewhere, e.g. `ftp://ftp.numerical.rl.ac.uk/pub/cuter/netlib.tar.gz`.

2.3.2 Representation of sparse matrices

There are many ways of representing a sparse matrix efficiently, of which Shahnaz[45] provides a compact review. The one we ended up using in this project (see Section 3.4.4.1 for the reason why) is called the Compressed Column Storage

format, also known as the Harwell-Boeing Sparse Matrix Storage Format[45]. A sparse $m \times n$-matrix containing k nonzero values is represented as two numbers telling the number of rows and columns, and three arrays:

`values` contains all k nonzero values, column by column (first all nonzero values from the first column, from top to bottom, then from the second column, and so on).

`rowIndices` contains k integers, one for each element of the *values* array, telling which row that element is located in.

`columnPositions` always contains $n+1$ elements — one for each column, and one additional element. Each entry contains an index into the two other arrays, which tells where the values of the corresponding column start. The last element contains k, which in effect tells us the last valid index into the two other arrays (namely $k-1$). Thus, the indices (into `values` and `rowIndices`) of the elements of column i are `columnPositions[i]` through `columnPositions[i+1]-1`.

For instance, the matrix

$$\begin{bmatrix} 9 & 0 & 0 \\ 0 & 2 & 7 \\ 4 & 3 & 0 \end{bmatrix}$$

would be represented as follows:

Value	9	4	2	3	7	—
Row index	0	2	1	2	1	—
Column positions	0		2		4	5

Note that we use zero-based indices. A sparse (column) vector can be represented similarly, as two arrays containing the nonzeroes and the row indices, and two single variables telling the number of rows and the number of nonzeroes.

2.3.3 Amdahl's law

The following section is taken from our fall project report[11].

In 1967, Amdahl argued[2] that parallel processing was *not* a good way to improve performance, based on the following observation: If we have a computation that consists of a sequence of serial steps that take a total time of t_s, and a certain percentage f of these steps can be performed in parallel using p processors[12], then the total time for the sequential part of the calculation will

[12]These are quite optimistic calculations, since we assume that the calculation can be parallelised without incurring communication penalties or extra computation steps, and that the parallel processors are as fast as the sequential one.

be ft_s. If we can distribute the remaining workload equally over the p processors, the parallel part will take the time $(1-f)t_s/p$. Since the sequential part must presumably be completed before the parallel computations can begin (or the other way around), these times must be added together for a total time of $t_p = ft_s + (1-f)t_s/p$. We then arrive at *Amdahl's law*[13] for the speedup $S(p)$:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{1}{f + (1-f)/p} = \frac{p}{1 + f(p-1)} \tag{2.37}$$

The most significant aspect of this formula is that it highlights the importance of f as a severely limiting factor for the potential speedup of parallelising. Amdahl used this to claim that parallel programming was not a good idea. However, since parallelisation is currently regarded as the primary way of improving performance in high performance computing[5], the modern interpretation is that speedups can come arbitrarily close to p if only we can make f small enough, and thus, one should focus on reducing f when parallelising a calculation. On the other hand, since f in most situations cannot (even approximately) become equal to zero[14], Amdahl's law provides an upper bound on performance gain for a specific problem given the best nonzero f we can achieve: As p tends to infinity, the speedup converges to $1/f$. Again, it should be noted that these bounds are optimistic, as they are based on very simplifying assumptions.

[13] Amdahl did not actually state this formula in his article, but it has been derived later (in many different forms) from his article.

[14] Calculations in which $f \approx 0$ and where the assumptions about independence between the parallel parts hold are called *embarrassingly parallel*.

Chapter 3

Development

This chapter contains a more or less chronologic report of the different stages this project has gone through, and we describe the design of the solvers we have produced and discuss the decisions we have made. We also discuss the largest hurdle we encountered: numerical instability.

3.1 Overall approach

The author and his advisor agreed that we should follow a step by step approach in which we begin with solving simpler problems and gradually proceed towards more challenging problems. We decided upon the following overall plan:

1. Implement the standard simplex method on a sequential machine, in order to gain familiarity with the subject and produce a program on which we can base our later development.

2. Parallelise the standard simplex method on Cell/BE (if the Cell/BE turns out to be very hard to program, we could first parallelise it on a regular multicore machine using e.g. pthreads or OpenMP (see `http://openmp.org/wp/`) to make sure our parallelisation approach is correct).

3. Implement the revised simplex method on a regular (single core) x86 computer.

4. Parallelise the revised simplex method on Cell/BE. This is expected to be much harder than the parallelisation of the standard simplex method, and we may have to resort to implementing one or more preexisting parallel simplex methods. For each implementation, several possibilities for refinements could be investigated:

a) Investigating various ways of handling numerical instability that may occur when using single precision arithmetic rather than of double precision.

b) Experimenting with how different representations of sparse matrices and vectors affect performance.

c) Figuring out how to improve performance by utilising the Cell/BE's SIMD operations.

d) Using autotuning to find good values for e.g. data transfer block sizes.

5. Investigate interior point methods and implement them if time permits.

3.2 Standard simplex method

In order to become familiar with programming the Cell/BE, we initially implemented a few versions of the standard simplex method (which is best suited for dense problems). We began with a straightforward implementation for regular (sequential) x86 machines, then ported it to run on the Cell/BE PPE while utilising its SIMD instructions. Finally, we created a parallel version of the method which uses the SPEs. However, as will be discussed in Section 3.3, it turned out that it is extremely hard to make the standard simplex method work reliably on even medium-sized data sets.

Choice of development language The Cell/BE SDK offers three languages: assembly, C, and C++. Assembly language is of course out of the question (due to development time and risk), except perhaps for a few very performance-critical parts — but modern compilers are normally very good at optimising, and there are C and C++ intrinsics for the extended instruction sets of the Cell/BE. Although we do not need advanced object-oriented features, C++ is the language of our choice, for the following reasons:

- The author is more familiar with C++ than with C.
- It offers more high-level data structures than C, through the Standard Template Library — in particular, the `vector` class for dynamic arrays.
- The author prefers the syntactic sugar of working with classes with member methods rather than functions that take structures as parameters.
- The template mechanism may be useful.
- The exception mechanism may be useful for debugging.

3.2.1 x86 and PPE version

Our x86 solver is a more or less direct implementation of the pseudocode from Vanderbei[47] and Cormen et al.[8]. However, there are some refinements:

- Rather than a raw array, we use a class for representing the tableau, so that we can make functions for operations such as adding a multiple of a pivot row to another.
- We would like the ability to switch between different data types in order to test how much the numerical stability is affected by the use of single precision arithmetic rather than double precision. Therefore, we use the symbol `TYPE` as a data type. This symbol is expected to be defined as `float` or `double` or any other suitable data type (see Section 3.3.3) — this can be done through a compiler switch, e.g. `-DTYPE=float`.
- During development, all functions that accessed the tableau performed bounds checking, in order to facilitate debugging. This incurs a performance penalty, in particular in simple functions that just read or write one entry, so we removed the code after the development was complete. In hindsight, we should have used defines to allow for simple enabling or disabling of that feature, but it will not be hard to add such code again.

The PPE solver is very similar. As described in Section 2.2.1.2, the PPE supports SIMD instructions (also referred to as **`vector`** instructions) capable of operating on four single precision floating point values simultaneously. Since the simplex method primarily consists of row operations on the tableau, it is an excellent target for such vectorisation. The only problem might be the low arithmetic intensity of the simplex method, which may reduce performance because a lot of data needs to be loaded into the registers, and only a very simple and fast operation is being performed on each element before it is thrown out again. However, the Cell/BE has an advantage in this situation over other architectures, since the LS is faster than regular caches. On the other hand, once the tableau becomes too large to fit in the LS, one may start seeing reduced payoffs from SIMD instructions because data cannot be provided fast enough.

We only need one type of SIMD instruction, namely the fused multiply-add, as described in Section 2.2.2.1. The central part of the code is the following line: `destination_v[j] = vec_madd(source_v[j], factor_v, destination_v[j]);`, which multiplies four values from the source row by a specified factor and adds them to the destination row.

An interesting implementation detail related to SIMD is that **`vector`** instructions can only operate on 16 byte data blocks (that is, four **`float`** or **`int`** values or two **`double`** values) that are aligned on 16-byte boundaries. Therefore, we must use `malloc_align()` for memory allocation. Furthermore, one must keep in

mind that the width of the tableau may not be divisible by four, in which case the last elements will not fill an entire **`vector`**. The simplest way of getting around this (and avoiding tests for whether one has reached the end of a row) is to round the number of columns up to the nearest multiple of four and allocate the corresponding amount of memory. To the other parts of the code, the width will still appear to be the original number of columns, whereas all internal operations will utilise the fact that each row contains an integral number of **`vector`**s. The last elements in a row may be garbage, but that does not matter since they are not reachable from the rest of the code (and there are no division operations, so division by zero is not a risk). This is implemented in the `Matrix` class (see Appendix A.1) by having one variable called `cols`, which keeps track of the original number of tableau columns, and one called `physicalCols`, which keeps track of the rounded-up width.

Code The PPE code was developed by copying and rewriting the x86 code, but we were able to merge them back into one program (through the use of **`#define`**s that control which parts of the code are activated), listed in Appendix A.1. This sacrifices a little bit of readability for the sake of compactness and avoidance of code duplication. The appendix also contains information on how to compile it, and short descriptions of what each file contains.

3.2.2 SPE version

Our approach is fairly obvious[1]:

1. The PPE, which initially holds the entire tableau, distributes the tableau rows evenly among the SPEs, giving each SPE a batch of consecutive rows.

2. The first SPE analyses the objective function to determine the leaving variable and sends the column number to the PPE, which distributes this number to the other SPEs. If no leaving variable was found, the optimal solution has been found, and the SPEs are asked to send their basic variable values to the PPE and terminate.

3. Each SPE determines the strictest bound (imposed by its subset of the rows) on the value of the leaving variable and sends the bound and the corresponding row number to the PPE.

4. The PPE determines which SPE that "wins" (because it has the strictest bound) and requests this SPE to transfer the pivot row to main memory using DMA; afterwards, all the other SPEs are asked to initiate a DMA

[1] After having written the application, we found that Yarmish[50] uses a very similar approach, albeit for cluster computers with MPI.

transfer to receive this row. If no SPEs found a finite bound, the problem is unbounded, and the SPEs are asked to terminate.

5. Each SPE performs row operations on its part of the tableau, using the pivot row, and notifies the PPE upon completion. Go to step 2.

We would have liked to employ some sort of broadcast operation for distributing the pivot row in step 4, but we could not find out how to do so.

Note that we have only implemented Phase II. The code is listed in Appendix A.2. SIMD operations are utilised on the SPEs, in a manner similar to what was done in the sequential PPE solver.

3.2.3 MPS and CPLEX parsers

The `netlib` data sets are stored in a file format called *MPS* (Mathematical Programming System). The format hails from the punch card age; as such, it is fairly arcane (it employs fixed format), but all the simpler to parse. This was fortunate; since we could not find any available parsers, we had to write our own. We also needed to write a parser for the ILOG CPLEX format, since Miriam AS provided us with a few data sets in that format (these are located in the `datasets` folder of the source code attachment). Our parsers do not handle all aspects of the formats, but they are sufficient for those sets that we use. The source can be found in Appendices A.5 and A.5. Maros[37, Chapter 6] gives a fairly compact presentation of the MPS format.

3.3 Implementation problems

3.3.1 Numerical instability

Our initial plan was to begin with something we thought to be fairly straightforward and then gradually proceed towards harder problems, along the lines described in Section 3.1. Steps 1 and 2 initially seemed to have been as simple as we had assumed them to be (step 1 was based on the descriptions and pseudocode from [8] and [47]), and the Cell/BE parallelisation appeared to go well. These implementations are listed in Appendices A.1 and A.2. Unfortunately, once we started running the solvers on data sets of nontrivial size (the `netlib` sets), we started experiencing problems. For some sets, the solver gave answers that were correct only to a few decimal places, while for other sets, the answer was off by several orders of magnitude and thereby entirely useless. Yet other sets would cause the solver to go into a cycle where the same sequence of pivots was repeated over and over. Most of the sets that needed a Phase I would be declared infeasible because the solver never managed to make the objective function value reach zero. After much fiddling and debugging (we did

have some minor errors in our initial implementations), and after starting to use Bland's rule (see Section 2.1.2.1), we seemed to get rid of the cycling problems. However, the answers we got were still mostly wrong.

We were of course aware of the fact that floating point calculations are not precise, and we noticed that the Phase I problems normally were caused by the objective function reaching a value that was very close to zero, but not exactly zero. We tried various approaches to zeroing out numbers that were "small enough", as detailed in Section 3.3.2 below, but nothing (except for the GMP approach in Section 3.3.3) gave consistently good results, and we began to seriously doubt our own abilities to code anything properly — because we were assuming that the standard simplex algorithm (like all other algorithms we have encountered during our university studies) was supposed to "simply work" when implemented as specified. Our beliefs were reinforced by the fact that well-known works such as [8] and [47] make no mention of the standard simplex method being particularily *unstable* (they only say that other methods are being used in practice because they are more *efficient*). Also, [42] provided an implementation of the standard simplex method — but when we got around to try it (we had originally just consulted the book, noticed that it used the standard simplex method, and decided to implement our own solver before looking at the "solution"), it ran into the same kinds of stability problems as our code. Tellingly, in the third edition[43], it has been replaced by an implementation of the revised simplex method (but we could not quite get it to work; we have probably misunderstood the way the input data was supposed to be formatted).

Already at the beginning of the project, concerns were raised about the suitability of the Cell/BE for this kind of computations, since its double precision arithmetic is very slow compared to its single precision arithmetic. We had initially developed our code using single precision, but switching to double precision did not help much — the solver gave better answers for several sets, but the problems persisted, in particular for larger sets.

We eventually resigned and contacted a group of mathematicians with which Natvig is acquainted, describing our problems and asking them for help on how to make the standard simplex method work stably. The response, from Andersson and Christiansen[4], was highly useful. First, they indicated that we were not alone about having such problems: "Implementing the standard simplex method is unfortunately considered both highly inefficient and very numerically unstable. From your email I see that you have encountered many of the "famous" problems with this implementation." Second, they suggested a direction in which we could continue, namely implementing the revised simplex method and a parallel algorithm called ASYNPLEX. Third, they suggested a splendid book that would have been perfect to us if we had been aware of it from the start: *Computational techniques of the simplex method* by Maros[37]. This

is the closest we have come to a book that is detailing everything one needs to do in order to make the (revised) simplex method stable. At this point, we had lost enough confidence in our own abilities in numerical mathematics that we did not dare to start implementing the revised simplex method from scratch (in particular because more than half the project time had passed at this point, and because an entire book apparently was necessary for the implementation of one algorithm). Therefore, we have only briefly studied [37], but we strongly recommend this book to anyone who intends to develop a serious LP solver.

Agreeing with the words of Donald Knuth, "premature optimization is the root of all evil"[31], we decided not to optimise our Cell/BE standard simplex method beyond the simple step of using SIMD instructions for the matrix manipulations. Time spent optimising an incorrect program is most likely going to be wasted time, as the optimised parts will probably be rewritten (or the whole program is discarded and another one is developed from scratch). Instead, we decided to focus on implementing ASYNPLEX.

3.3.2 Attempts to handle roundoff errors

As mentioned above, we tried different approaches to handling the roundoff errors that will necessarily occur when using floating point numbers. They include:

- Scanning the tableau in-between each iteration and setting every number whose absolute value is below a certain threshold to zero.
- Terminating Phase I when x_0 has reached a value that is "sufficiently close" to zero.
- Whenever two numbers are added to or subtracted from each other, the result is compared to the two original numbers. If the ratios between the result and each of the original numbers are sufficiently small, the tresult is set to zero. We had some success with this rule; if the ratio is e.g. 0.00001, the right result for AFIRO is produced.

Although for some small data sets, we were able to find values that made the program produce the right answer, none of these approaches yielded consistently improved results.

See comments in the source code of `Matrix.cpp` in Appendix A.1 for instructions on how to enable some of these roundoff techniques.

3.3.3 An exact LP solver using rational numbers

In order to demonstrate that the stability problems are not caused by errors in our implementation, we have made our code support usage of the *GNU multiple*

precision arithmetic library (GMP — see `http://gmplib.org/`), which among other features has a data type for representing arbitrary-size rational numbers exactly. Since the simplex methods only apply the four basic arithmetic operations throughout their operation, all numbers in the tableau will remain rational[2]. Compile the code by running the `buildgmp.sh` script; this will link to GMP (which must first have been downloaded, compiled and installed on the system) and tell our code to use the `mpq_class` data type for all arithmetic operations and to output results in fraction form. When using GMP, the code obviously slows down by a significant factor and the memory consumption increases (which is why this approach is useless in practice unless it is absolutely essential to obtain exact results). Table 3.3.3 shows the results for some of the small-to-medium `netlib` sets. Note that our solver performs maximisation, while the `netlib` sets are supposed to be minimised (but for some reason, the MPS format does not specify whether to maximise or minimise) — therefore, our MPS parser negates the objective function, so that the answers will have correct absolute value but wrong sign. According to the `README` file, the "official" `netlib` results have been obtained using the MINOS solver, version 5.3. All digits of the `netlib` results agree with our exact results.

Data set	Netlib result	Our result	Iterations	Time
AFIRO	$-4.6475314286 \cdot 10^2$	$\frac{406659}{875}$	16	0.044 s
BRANDY	$1.5185098965 \cdot 10^3$	$\frac{-160658773925981637045452922983525576384594628005783164820957774809004110966339863688910580028111607217135047501508720411569323127506371426417345909327662918125000000000}$	605	491.402 s
LOTFI	$-2.5264706062 \cdot 10^1$	$\frac{631617651547}{25000000000}$	537	40.362 s
SCFXM2	$3.6660261565 \cdot 10^4$	$\frac{-4874671419119861011078305839244653390630042031652016001773580110200073242301193326104545913210105870694071773019150478354800551049955591329688117603047126754336400784888771958942099169754747473929704674848158506258498441472830720462613814446552258600000000000000000000}$	1299	2363.2 s
STOCFOR1	$-4.1131976219 \cdot 10^4$	$\frac{7368963026860358678147059812142062686879894069612494322055836783}{17915412056905368048974617968750000000000000000000}$	135	3.381 s

Table 3.1: Some results of our exact standard simplex implementation

[2]Assuming, of course, that they were initially rational — but all data formats for representating of LP problems are based on floating point numbers, which are inherently rational.

3.4 Revised simplex method

3.4.1 Performing the matrix inversion in parallel

The revised simplex method as described in Section 2.1.3 must occasionally spend some time reinverting the basis matrix. A simple yet attractive idea is to offload the matrix inversion onto a separate processor, which may then spend all of its time performing inversions. Then, the main processor can spend all of its time on the remaining steps of the method (while occasionally being provided with a reinverted basis matrix from the inversion processor), and one gets the added benefit of the matrix being reinverted more often (which should be good for numerical stability). Unfortunately, as reported by Ho and Sundarraj[21, Table 2], the inversion consumes less than 20% of the total time of the revised simplex method, and as such, speedups are limited as per Amdahl's law (see Section 2.3.3). Furthermore, this approach does not scale to more than two processors. Therefore, we have chosen not to pursue this direction. Note, however, that ASYNPLEX incorporates the same idea of having a separate inversion processor.

3.4.2 Decision to adapt ASYNPLEX and Vanderbei's code

Considering the problems discussed in Section 3.3, we realised that we had too little experience with numerical computation in general and the simplex method in particular. We could perhaps have been able to implement a stable sequential solver from scratch by following Maros' book[37], but this would most likely consume the rest of the project time. Therefore, we decided to find an existing sequential implementation of the revised simplex method and rewrite it as per some existing parallel revised simplex method. Finding a suitable implementation was not easy, because one apparantly needs to make a trade-off between small code size and ease of understanding on one hand and numerical stability on the other hand. As noted in Section 2.1.6, all the major, well-known implementations have exceedingly large code bases. After unsuccessful attempts at understanding GLPK and retroLP, we chose Vanderbei's implementation. The choice of parallelisation approach fell on the ASYNPLEX algorithm, described in Section 2.1.4, mostly due to its strong resemblance to the sequential revised simplex method, and because it was recommended to us by Christiansen and Andersson[4].

Basing ourselves on an existing sequential implementation also has the advantage of allowing a direct comparison between the sequential and parallel versions of the same code, allowing us to better gauge the speedup that is offered by ASYNPLEX itself (Hall et al. compared their performance to that of an entirely different sequential implementation), and the further speedup that is obtained on Cell/BE.

The disadvantage, of course, is that retrofitting an existing sequential implementation may require a vast effort in case parts of the code does not lend itself well to parallelisation (this easily happens when the code relies on global variables or internal, static variables, because such variables will need to be duplicated so that each thread has its own instance of it.)

At this point in the project (early May), we had a meeting with Mr. Spaggiari from Miriam AS where we discussed the problems we had encountered, and it was agreed that we should proceed with first implementing a simple ASYNPLEX prototype and then try to rewrite Vanderbei's code as per the ASYNPLEX design.

3.4.3 ASYNPLEX prototype in C#

In order to make sure we actually understood the ASYNPLEX algorithm, we first implemented a prototype in C#, using dense linear algebra (because sparse linear algebra operations are much more complicated to implement, and we only wanted a proof of concept implementation that could run on small datasets). The reason for using C# is that it is a more high-level language (than C++) in which development is quicker and thread handling is simpler than with C++ and `pthreads`. Furthermore, the Visual Studio integrated development environment provides an excellent debugger, which would be highly useful for debugging the many threading mistakes we suspected (rightfully) that we would make.

We succeeded in implementing a mostly functional prototype, albeit with some remaining threading glitches, and therefore decided to go ahead with ASYNPLEX. The code is not particularily useful, but for the sake of completeness, it is included in the source code attachment.

3.4.4 Restructuring Vanderbei's code

Vanderbei's code is available at `http://www.princeton.edu/~rvdb/LPbook/`. It is written in C, and we initially chose to continue the development in that language, since we felt that gaining more practice in pure C coding would be useful. While we still agree to that sentiment, we later regretted the choice, since it forced us to use a number of constructs that are more cumbersome in C than in C++ (such as passing struct pointers to functions rather than calling member methods on objects), and we were also bothered by the lack of templates.

3.4.4.1 Sparse vector and matrix representations

Vanderbei's implementation uses the Compressed Column Storage format (as described in Section 2.3.2) for sparse matrices and a similar scheme for sparse vectors. Unfortunately, he did not have a structure or class that contained the

arrays and variables for each sparse matrix or vector. For instance, the matrix **A** would be represented with the arrays `a` (values), `ia` (row indices), `ka` (column positions) and the variable `nza` (number of nonzeroes) — a naming scheme that we found to be very impractical (all variables must be passed as parameters to functions that are to manipulate sparse vectors and matrices), and which slowed down our process of understanding his code. Therefore, we introduced structures that combined these related arrays and variables, and we refactored the code to use these strucures throughout. Our structure for sparse matrices looks like this:

```
struct SparseMatrix {
  int rows;
  int cols;
  int numNonzeroes;
  int * rowIndices;
  int * colPos;
  TYPE * values;
};
```

Note that `TYPE` is a preprocessor symbol which facilitates experimentation with different precisions — it should be defined as either **`float`** or **`double`**.

Due to the vast amounts of vector manipulation (and also in order to track down some bugs we believed were related to reading/writing outside of the array bounds, but turned out to be caused by wrong memory management), we made a more elaborate sparse vector structure, which uses the **`vector`** class from the C++ Standard Template Library. The `at()` function performs boundary access checking on each access — this is inefficient, but highly helpful during development. The compiler will most likely inline the simple accessor functions and operators, so that the usage of high-level classes such as `std::vector` will not incur any performance penalty (if the boundary checking is turned off). The structure can be found in the file `sparse.h`.

Beware that in order to save time, Vanderbei preallocates the arrays for any sparse vector with r rows to have size r, but only the first k entries are used at any time (where k is the number of nonzeroes). Whenever the contents (and the number of nonzeroes) of the vector changes, one can simply fill the arrays with as many entries as necessary, since each individual vector has a constant size throughout the program and the number of nonzeroes obviously will never exceed the full vector size. This, in combination with our lack of unit tests, caused a rather insidious bug: our `copySparseVector()` function only allocated as much space for the new vector as the current amount of nonzeroes in the source vector — and when other parts of the code proceeded to add more nonzeroes to the new vector, data in other vectors would be corrupted. This also demonstrates why the use of `std::vector` is useful (at least during development), as it would have caught such "index out of bounds" errors.

Also, Vanderbei did not explicitly store the sizes of the vectors and matrices, as they could always be deduced from context (normally as having m or n rows). We feel that this practice obscures the relationship between a loop header and its body — if `v` is a sparse matrix with `n` columns and we want to write a loop that manipulates `v`, we prefer e.g. `for (int j = 0; j < v.cols; ++j)` to `for (int j = 0; j < n; ++j)`. Therefore, we have included the size information into our structures and have tried to use them instead of `m` and `n` (this also makes the linear algebra functions slightly more general, and it would facilitate unit testing). Note that such preallocation is not done for matrices, since this would require too much space, and because the main part of the algorithm never changes the matrices directly (it uses permutation lists to keep track of how columns are swapped).

3.4.4.2 Overview of changed files

Here, we describe the files we have created ourselves and those of Vanderbei's files we have modified in a nontrivial manner.

`tree.c|h` contains a binary search tree structure. It only supported one active tree at any time (through the use of static variables). Because it is used by some of the linear algebra operations in the iteration processes, we needed to create a `struct` for the internal tree information so that we could have several tree instances.

`sparse.c|h` contains our `struct`s and supporting functions for Vanderbei's sparse vectors and arrays.

`print.c|h` is a utility for making sure that outputs from different threads do not collide with each other (often, a line that is output from one thread gets cut in two by a line from another thread). It is implemented with mutexes (making sure that only one thread is allowed to print at a time), so excessive printing may hurt performance.

`2phase.c` was the core of Vanderbei's original revised simplex solver, and `iterationprocess.c` is strongly based on this file. We have chosen the `solver()` function in this file as the "entry point" of our code, because the input parsing and processing has been completed at this point. If the `useAsynplex` variable is true, we skip Vanderbei's solver and instead launch the ASYNPLEX threads and wait for their completion.

`columnselectionmanager.c|h` contains the ASYNPLEX column selection manager. We had problems implementing it because we feel that [19] is unclear on how the statuses of the variables are supposed to change, in particular when new candidates arrive. Our current interpretation is that

a new candidate should be accepted into the pool of attractive candidates unless its status is "selected" or "rejected" *and* it obtained that status at a basis that is more recent than the basis where the candidate was formed.

basischangemanager.c|h contains the ASYNPLEX basis change manager, whose functionality is so simple that the code probably speaks for itself.

communication.c|h is a simple communication layer strongly inspired by MPI. A message has a sender (string), a receiver (string), tag (string) and payload (generic memory buffer). The communication primitives are secured with mutexes. When a thread requests to receive a message, it may choose whether or not to specify a sender (passing `NULL` as the `sender` parameter indicates "any sender") and whether or not to specify a tag (passing `NULL` as the `tag` parameter indicates "any tag"). If no matching message is available, an empty message is returned. The implementation is somewhat inefficient in that sequential search is used to locate matching messages. Also, we should have used `std::queue` instead of a vector (but as noted, the project started out in C, where STL is not available). However, in ASYNPLEX, the message queue does not grow particularily long, so this is not a big problem in practice. Still, a real MPI implementation, for instance `mpich`, might have served us better.

invertprocessor.c|h is based on `lueta.c|h`. This process is continuously recomputing the inverse of the basis matrix, and is informed of basis changes by the iteration processes. The LU factored representation of the inverse is sent to the iteration processes upon completion of each inverse calculation.

iterationprocess.c|h is the only thread which may exist in several instances; therefore, we must use a `struct` to store all the internal data for each iteration process, and pass pointers to instances of the struct to the different functions. This code is based on `2phase.c` and `lueta.c`.

genericvectors.c and the similarly-named files are our attempt at simulating C++ templates in C. The approach is to write the code with lots of macro symbols as placeholders for function and type names, and then **#include**ing the code repeatedly while **#define**ing the symbols appropriately. This leads to rather unreadable code, and was one of the most important reason that we eventually switched to C++.

timer.h is the timing utility described in Section 4.1.3.

The functions in `iterationprocess.c` have been named in accordance with the pseudocode given for ASYNPLEX. Vanderbei's original comments detail the mathematical operation that is performed by each function.

3.4.4.3 Threading

`pthreads` is the de facto threading library for Unix and Linux, and since we have some prior experience with it, the choice was simple. There is no need for advanced threading features; beyond the functions for starting the threads and waiting for them to finish, we only employ the *mutex* (mutual exclusion) mechanism: a `pthread_mutex_t` variable can be declared and then initialised with `pthread_mutex_init()`. Any thread may then call `pthread_mutex_lock()` on the mutex in order to request a lock on it. The lock is granted if no other thread is holding the lock; otherwise, the thread is queued. When a thread releases the mutex with `pthread_mutex_unlock()`, an arbitrary thread among the queued threads (if any) is granted the mutex.

As usual with threading, the hard part is not the underlying concepts, but all the problematic situations that can occur when the threads start interacting. We have had many small threading bugs that were not too hard to find, but we also had one that was a bit harder and was quite interesting. Consider the following race condition: An iteration process, say, I0, has performed a pivot and sends messages about this to all other iteration processes. If the I0 thread gets preempted after sending only some of the messages, it could be that e.g. I1 receives the message and goes on to perform another pivot and tells everyone else about it. Then, I2 might receive the message from I1 before the message from I0, in which case it will fail an internal consistency check for the sequence of pivot operations. This situation can be prevented by either implementing a function that can send multiple messages at once without the risk of other messages getting interleaved with them, or letting the iteration processes keep a queue of premature pivot messages. We did the former, but that required internal support from the message system, and we are not sure if such functionality can be achieved with MPI.

3.4.5 Cell/BE implementation of ASYNPLEX

3.4.5.1 Communication approach

ASYNPLEX is an algorithm for message-passing distributed memory systems, but its authors describe how to adapt it to shared memory systems. While the Cell/BE architecture resembles both shared memory and distributed memory architectures, we chose to go with the message-passing approach because this is what we are the most familiar with. On x86, we implemented the simple message passing system described above. On Cell/BE, we used the Cell Messaging Layer (CML) (see Section 2.2.3) — we were in a hurry and therefore started using the first MPI implementation we could find for Cell/BE. Unfortunately, CML turns out to have several disadvantages:

- It only supports messaging between the SPEs, not between an SPE and the

PPE.

- Like MPI, CML employs the Single Program Multiple Data (SPMD) model, which means that all processors must run the same program. This means that even if different SPEs are to perform different tasks, they must each contain the code both for its own functionalify and the code for the functionality of all other SPEs. Still, we chose to run both the column selection manager and the basis change manager on the SPEs, because their code is fairly short, their operations are fast and simple, and it is vital that they are able to respond quickly to messages from the iteration processes. For the same reasons, one should merge them into one SPE thread so that the other seven SPEs (rather than six) would be available for iteration processes, but we did not get the time to do this.
- CML only supports synchronous point-to-point primitives, and we did not realise soon enough that this is not sufficient for ASYNPLEX. Therefore, we resorted to implementing our own message passing system (again) on the PPE and using CML's remote procedure call (RPC) functionality to send and receive messages to and from a message queue on the PPE. It is cumbersome, but it works. Unfortunately, we cannot gauge the efficiency (or lack thereof) of this approach, since we did not get the time to complete the implementation. The reason we stick to CML is that we believe that using the RPC system is safer than using DMA operations directly (at this point in the project, we did not have time to debug obscure memory corruption errors).

3.4.5.2 Overview of changed files

Cell/BE programs must be split into a PPE program and an SPE program (which must reside in different Eclipse projects). Several of our source files are common to both projects, but Eclipse does not seem to support the inclusion of files that lie outside of the project directory. Therefore, we have resorted to using symlinks to put the same file into both projects without actually duplicating it. However, in the zip file attachment to this thesis, the files are physically duplicated.

The Cell/BE ASYNPLEX solver was made by rewriting the x86 ASYNPLEX solver. In the process, we utilised the opportunity to switch to a more object-oriented structure — each process now has its own class. We also needed some wrapper code for supporting CML, but apart from this, there are fairly few changes in the code base. The most important files (aside from the ASYNPLEX processes, whose code has not changed much) are:

`asynplexcontrol.c|h` The PPE code that provides RPC functions and initiates the MPI programs on the SPEs.

`communication.c|h` The CML RPC-based messaging system, which resides on the PPE. It maintains a message queue similar to the one from the x86 solver. The SPEs can send and receive messages through RPC calls that will transfer a buffer containing the sender id, receiver id, tag, and payload.

`rpc.c|h` SPE convenience functions for message passing, which initiate RPC calls on the PPE.

Note that the invert processor still resides on the PPE, while the other three processes are run on the SPEs. All eight SPEs are used; thus, there are six iteration processes.

Chapter 4

Evaluation

Due to all the challenges we have faced, we have not been able to produce sufficiently stable solvers, and our Cell/BE implementations only handle very small data sets. Still, a number of interesting questions can be posed, and their answers might serve as a guidance to those that will continue the project. Finally, we discuss opportunities for future work.

4.1 Performance measurements

4.1.1 Testing environments

The x86 experiments were run on a machine containing an Intel Core2 Quad Q9550 with four cores at 2.83 GHz, with 4 GB of system memory. The compiler is `gcc` version 4.2.4. The system is running Ubuntu version 9.04 "jaunty" with Linux kernel version 2.6.28-11-generic.

The Cell/BE experiments were run using the IBM Full-System Simulator, version 3.1-8.f9, on a computer running Fedora 9. Being a simulator, the timing results obtained on it are independent of the physical hardware of the host computer. The simulated PPE cores have a frequency of 3.2 GHz.

4.1.2 What to measure

- How well a vectorised PPE standard simplex implementation performs relative to a non-vectorised version;
- How the time consumption of the Cell/BE parallel standard simplex implementation depends on the number of SPEs used;
- How well the Cell/BE parallel standard simplex implementation performs relative to the sequential PPE implementation;

- Time spent by the SPEs waiting for data to be moved to the local store;
- How well the x86 ASYNPLEX implementation (on a multicore) performs relative to Vanderbei's original solver.

4.1.3 Measurement methods

All Linux distributions incorporate the `time` utility, which reports the amount of time spent by a process: `real` (wall time), `user` (time spent in the process' own code), and `sys` (time spent in system calls on behalf of the process). For multithreaded programs, the two latter values will be the sums of the time spent by all threads and may therefore exceed the first value. The precision is at most one millisecond, and it is limited to timing an entire program. In order to time only parts of a program, the standard C++ function `clock()` is commonly used. Unfortunately, its resolution is system-dependent and often too coarse. Therefore, we opted for a much more high-precision timer that is available on Intel processors and PowerPC (and thereby on the PPE), called the Time Stamp Counter. This counter is incremented on each clock cycle (on PowerPC, it might be controlled by a separate clock[53]), and it may be read by using the `rdtsc` instruction. Timing utilities have been implemented in `timer.h`, and they employ the function `rdtsc()`, taken from [53], that uses the aforementioned instruction. The drawback is that in order to get the time in seconds, we must empirically determine the `rdtsc` frequency. Using the following simple code fragment (which was run with `time` in order to verify that it actually slept for that long; we also tried several different delays), it was found to be 25 MHz on the PPE (we later discovered that this could also be found with the command `cat /proc/cpuinfo`) and 2.83 GHz (as expected) on the machine described in Section 4.1.1:

```
#include <unistd.h>
#include <cstdio>
#include "timer.h" // See the source listings in the appendix
int main() {
  unsigned long long start = rdtsc();
  usleep(1000000); // Sleep for one second
  printf("%llu\n", rdtsc() - start);
  return 0;
}
```

All x86 and PPE programs were compiled with the `-O3` switch (maximal optimisation level). The x86 programs were run run using `nice -n -20` in order to force the operating system to ensure a favorable thread scheduling priority for the programs (this does not seem to have any effect on the PPE, probably because it supports two simultaneous threads (one for the operating system and

Set name	Rows	Columns	Nonzeroes
80BAU3B	2263	9799	29063
ADLITTLE	57	97	465
AFIRO	28	32	88
BRANDY	221	249	2150
ISRAEL	175	142	2358
SC105	106	103	281
SC205	206	203	552
SC50A	51	48	131
SC50B	51	48	119
SCTAP1	301	480	2052

Table 4.1: Dimensions of selected `netlib` sets

one for the user program) and we do not have any multithreaded PPE programs except the incomplete Cell/BE ASYNPLEX implementation).

Note that during timing, we disabled or commented out most of the `couts` and `printf()` calls in order to minimise the output processing's impact on the run time. On the SPEs, console output is particularily expensive because the output must be DMA'ed to the PPE.

The Cell/BE simulator can gather detailed statistics on each of the SPEs. Before invoking the program one wishes to analyse, all SPEs must be set to "pipeline mode", and after the program has been run and the simulator has been stopped, the simulator command `mysim spu X stats print` can be issued for each SPE by replacing the X by a number between 0 and 7 to indicate which SPE one wants the statistics for.

The `netlib` sets vary greatly in size, so it may be useful to know the dimensions of the sets we have used. We list them in Table 4.1, as a reference for those who do not want to acquire the entire collection.

4.2 Results

Note that the timings do not include the reading and parsing of the MPS input file, as this must necessarily be done by any implementation, and is not of interest when one desires to gauge the efficiency of the core algorithm. We do, however, include the time required for starting and stopping the SPE threads, as this is a Cell/BE specific feature, and we felt that excluding it would give the Cell/BE an unfair apparent advantage in comparison to other architectures.

4.2.1 Standard simplex method

As discussed in Section 3.3, the standard simplex method is highly susceptible to numerical instability, and our implementation is no exception to this. It is essentially useless in practice because for most sets of realistic size, it produces answers that are off by orders of magnitude. Still, we might be able to learn something about the computation to communication ratio of the algorithm, and how much time vector operations are capable of saving. Also, Miriam AS stated that they are interested in such measurements.

It should be noted that these timings were obtained while running the simulator in "fast" mode, rather than in "cycle" mode (which is what one should ideally use for benchmarking), because we had problems getting the latter mode to work[1]. These modes control how closely the simulator mimicks the real hardware, and it could be that the results will be different if the experiments are repeated in "cycle" mode.

Again, please take note that our solvers did not produce the right answer on most of the sets used here (and some sets may only have been solved to the end of Phase I, because the solver never finds a feasible solution because of the numerical instability). These analyses are merely for evaluating the performance benefits of our design and implementation approaches, without regard for numerical stability — in the hope that more stable implementations may benefit from our observations.

Speedup of sequential PPE version by using SIMD

The standard simplex method has very low *arithmetic intensity* (number of arithmetic instructions per load from memory) — so much time is spent moving data from main memory into the cache and from there into the registers that the SIMD speedup of the simple operation that is executed on the data once it is in the registers may not have much impact. In order to find out how much impact the SIMD operations have in this situation, we compared the run times of the sequential standard simplex method on the PPE with and without SIMD operations. The results are seen in Figure 4.1 on the next page. As expected, they are far away from the fourfold speedup that should in theory be possible. Interestingly, the speedup increases with the size of the data set. We did not test larger sets than these because the simulator is terribly slow, but we suspect that once the data set is too large to fit in the cache, it will start slowing down again (but this, of course, will apply whether or not SIMD is used; it will just further diminish the gains from SIMD).

[1] In general, we have struggled a lot with the Cell SDK and simulator, whose installation procedures, user interfaces, and documentation are often somewhat obscure and lacking in detail.

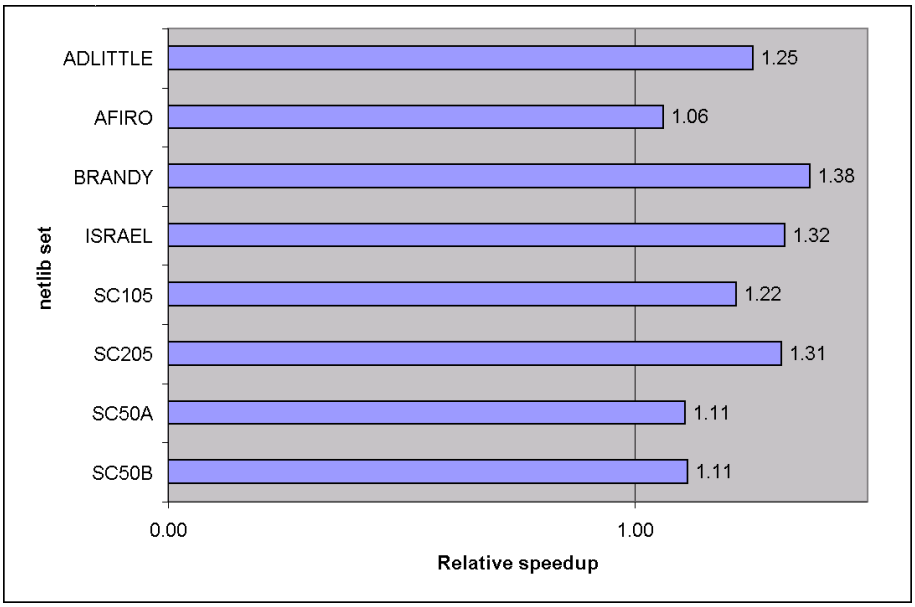

Figure 4.1: Speedup obtained in the sequential PPE standard simplex solver by using SIMD, relative to not using SIMD. Higher is better; 1.0 means no speedup.

Time consumption depending on number of SPEs

For our parallel standard simplex solver, it will be interesting to see how the number of SPEs (which can be anything between 1 and 8) affect the run time. Since that solver only implements Phase II, we must limit ourselves to `netlib` sets whose right hand sides are nonnegative, such as the `SC` sets. The SPEs do not currently track the locations of each variable, and therefore they cannot employ Bland's rule. Because of this, the solver cycles on `SC205`, so we have omitted that set.

The results, which are averaged over five runs, are presented in Figure 4.2 on the following page, and they are not very encouraging: going from 1 to 2 SPEs gives a minor speedup (but only for the largest set), but increasing the number of SPEs beyond 2 actually causes a *slowdown*. By looking at the detailed SPE timing statistics (as described in Section 4.1.3), in particular the overview of what the SPE spends its cycles on, we get the explanation (note that these are SPE cycles and are not supposed to match the cycle counts in the graph):

```
Single cycle                          507683 (  4.9%)
Dual cycle                             89774 (  0.9%)
Nop cycle                              62087 (  0.6%)
Stall due to branch miss              183518 (  1.8%)
Stall due to prefetch miss                 0 (  0.0%)
Stall due to dependency               733134 (  7.1%)
```

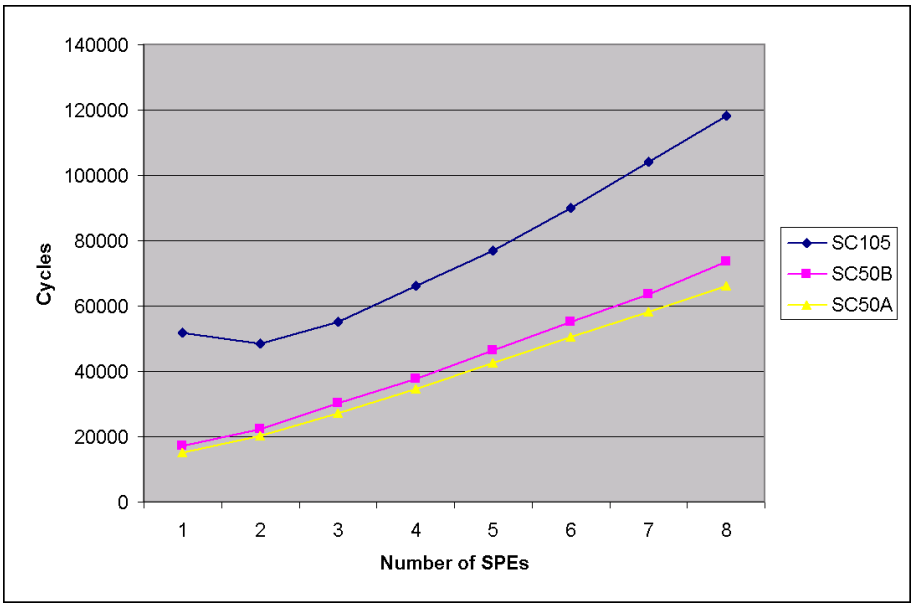

Figure 4.2: Time consumption (in PPE timer cycles) of parallel SPE standard simplex solver, depending on the number of SPE threads used. Lower is better.

```
Stall due to fp resource conflict          3024 (  0.0%)
Stall due to waiting for hint target       1324 (  0.0%)
Issue stalls due to pipe hazards              6 (  0.0%)
Channel stall cycle                     8812670 ( 84.8%)
SPU Initialization cycle                      9 (  0.0%)
-----------------------------------------------------
Total cycle                            10393229 (100.0%)
```

The SPE spends the majority of its time waiting for data, and only 5.8% actually performing computations! This highlights the importance of overlapping communication and computation (which we have essentially ignored in the development since there is not much use in optimising a program that does not produce correct answers). Furthermore, this algorithm will most likely perform better for sets that have much more rows than columns, since the pivot row to be transferred will be narrower. Most `netlib` sets are either approximately square or have more columns than rows, but such sets could be handled by using the dual simplex method instead (which in effect solves the transpose of the original problem). Also, if it is possible to perform some sort of DMA broadcasting operation so that the pivot row could be distributed to all SPEs simultaneously, performance could be improved (but it could be that the MFC is advanced enough to be able to notice that all SPEs ask for the same memory region and then performs an efficient broadcast).

Given that the SPE is stalled for 94.2% of its time, we see no point in per-

forming experiments to compare single precision performance versus double precision performance, a speedup or slowdown which affects only 5.8% of the run time will hardly be noticeable.

Performance of parallel SPE version vs. sequential PPE version It should not come as a surprise that the SPE version is slower than the PPE version, due to the massive amount of stalling. Therefore, we have not bothered to perform detailed timing experiments on this matter.

4.2.2 ASYNPLEX

Performance of x86 ASYNPLEX vs. original Vanderbei solver

Even with a quad core processor, our ASYNPLEX implementation turns out to be much slower than the original Vanderbei code — the results are displayed in Figure 4.3 on the next page. We suspect that this in part is due to the extensive data copying that takes place; we could have done a better job of letting the iteration processes share common data (such as the inverse basis matrix) and reusing message buffers rather than allocating and freeing them over and over. Also, keep in mind that Hall and McKinnon's implementation was run on a Cray machine, which we presume is much more optimised for high performance message passing programs than a regular quad core x86 machine is — and our self-written message passing system may not be particularily optimal. The fact that using more iteration processes only helped for the largest data set also suggests that there is a substantial message passing overhead.

On the bright side, this implementation is much more stable than our standard simplex implementations — it handles `80BAU3B`, which is a fairly large set, as seen in 4.1. It fails for many other large sets, but we believe that it might succeed for several of them by adjusting the feasibility tolerances (the different `EPS` values in `iterationprocess.c`). Our reasons for believing so is that for some of the sets that are reported as infeasible, the objective function value that is reached at that point is fairly close to the optimal value.

Note that we do have (at least) one unresolved threading bug or array boundary violation problem. The more iteration processes we use, the more frequently it crashes in the following manner: it runs normally (and produces correct intermediate values for the objective function) for a while, and then it starts to gradually produce more and more extreme values for the step length, and finally, one of the linear algebra operations crashes because one of the vectors does not contain a value at an expected index. We have tried to figure out if this is caused by an incorrect pivot operation or not, by recording all pivot operations performed by our solver and then forcing Vanderbei's original solver to follow the same path (this is easily done by hardcoding lists of leaving and

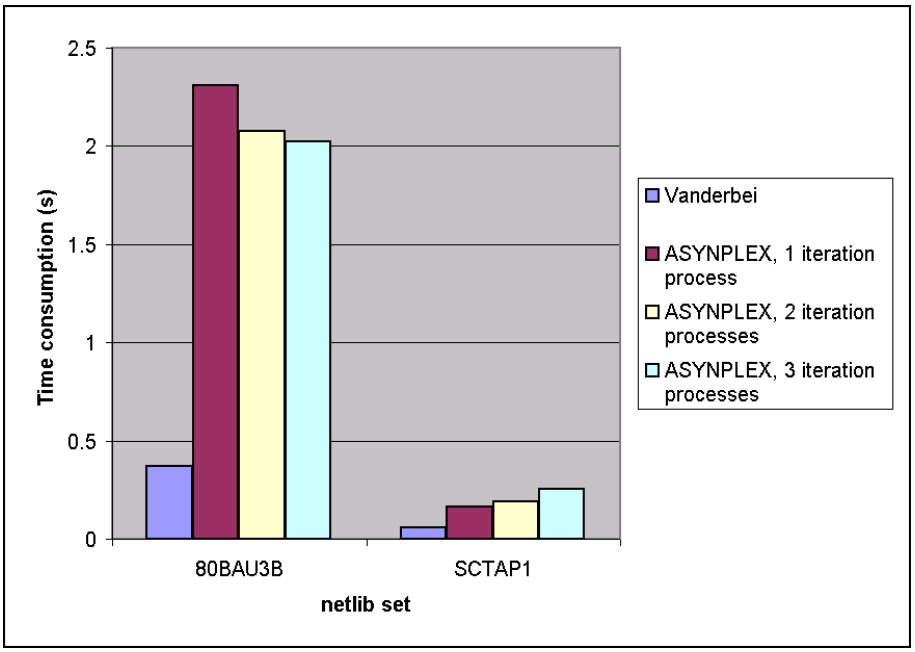

Figure 4.3: Performance of our x86 ASYNPLEX implementation relative to the Vanderbei code on which it is based. Lower is better.

entering variables into the program, and changing the pivot variable selection functions to simply return values from these lists), but we did not get the time to fully investigate it. The problem appears to be triggered by the presence of more than one iteration process; the solver works fine when only one iteration process is used.

Some comments on our incomplete Cell/BE ASYNPLEX implementation

We are very close to having a proof of concept implementation of ASYNPLEX on the Cell/BE. All ASYNPLEX processes are up and running (the invert processor on the PPE; the three others on the SPEs) and are capable of communicating with each other. The only thing we did not get the time to do was to fully rewrite the code in all communication endpoints such that it packs the data properly (which is a little tricky at some points where several arrays containing different kinds of data must be sent). However, a lot of work is required to produce a solver that can handle large data sets — for this, triple or quadruple buffering is required, and it must be integrated with all of the linear algebra functions.

4.2.3 Other aspects

Code size

Vanderbei's Phase I/II scheme causes the code size to be larger than necessary, because much code must be duplicated and changed slightly in order to work with both $\mathbf{A}$ and $\mathbf{A}^\top$. It is vital that the executable (compiled and linked) SPE program does not take up too much space in the LS, so that there is still room for storing data (and programs that take up more than 256 kB will not be possible to run at all). We will briefly discuss some of our experiences with trying to minimise the program size.

The debug flags, in particular `-g3`, dramatically increase the program size. Optimisation flags, in particular `-O3`, reduce object file size greatly (strangely, `-Os` does not seem to have much effect).

As often happens in optimisation, there are tradeoffs that must be considered — for instance, while loop unrolling gives a good speedup of tight loops, it increases the object file size, which one cannot always afford on Cell/BE. Thus, neither manual unrolling nor `-funroll-loops` should be done if (like us) one has a large program.

When optimising for size, one would normally want to consider using `-fno-inline` in order to disable function inlining (replacing calls to short functions by the actual function code). However, this flag actually *increased* the size of the object files. We suspect that this is due to heavy use of `std::vector` and its `[]` operator — the operator code itself can probably be translated into one instruction (load using memory address and offset), while a function call would require several instructions for parameter passing, stack management, etc.

The option that by far had the greatest impact on the final code size was `-s`, which tells the linker not to include symbol information (a debugging and linking aid) in the executable. Using the flag on our ASYNPLEX code reduces the size of the executable by a factor of almost four.

Another way of reducing code size is to use as few external libraries as possible.

Memory leaks

`valgrind` with `MemCheck` (`http://valgrind.org`) is an invaluble tool for detecting memory leaks (forgetting to release memory segments that are no longer in use, such that the program will continuously consume more and more memory) and illegal use of the memory allocation system (such as calling `free()` on the same pointer twice, which may easily cause corruption of the memory allocator's internal data structures). It can also detect corruption caused by directly overwriting the memory allocator's internal data, but *not* corruption of user data caused by writing outside of the boundaries of an array. We have used

this tool on several occasions during this project, and we believe that we have removed all memory leaks caused by our own code. The only leaks that remain are caused by initialisation code, so that each "leak" occurs only once (not inside loops) and the data that is being allocated is needed throughout the entire program (and is automatically freed when the program terminates), so that we chose not to spend time on removing them.

4.3 Reflections on unimplemented features. Ideas for future work

We will now discuss features we believe will be useful, but that we did not get the time to implement. One may also want to study a fairly recent survey article by Hall, which discusses the current state of research on parallel simplex methods[17].

4.3.1 Interior point methods

The most time consuming step of many interior point methods is a Cholesky factorisation. Monien and Schulze[39] discuss approaches to parallelising this operation for sparse matrices, and one of those methods, called the *multifrontal method*, is elaborated by Schulze[44].

Andersen and Andersen[3] present a parallel shared memory version of the interior point method that is (or was at the time) underlying the Xpress solver (see Section 2.1.6). Yet another parallel interior point method is presented by Karypis et al.[28].

The opportunities for implementing any of these methods on the Cell/BE should be investigated, but, given our experiences, it will probably be necessary to dedicate at least an entire master's thesis to that subject.

For dense matrices, Cholesky factorisation has already been implemented on the Cell/BE by Kurzak et al.[35], whose work should be studied by those who wish to implement the same operation for sparse matrices.

4.3.2 Mixed precision

It may be possible to overcome the limited precision that is offered by the Cell/BE without rewriting the computation to use double precision (which will incur a massive slowdown). Some linear algebra problems can be solved by using a technique called *iterative refinement*. Kurzak and Dongarra[36] describe a successful implementation of a Cell/BE program for solving equations of the form $\mathbf{Ax} = \mathbf{b}$, which meets the LINPACK benchmark's requirements for the precision of the solution. Similar techniques may be investigated for the linear algebra operations that underlie the revised simplex methods.

4.3.3 Stabilisation techniques

According to Hall and McKinnon, their own ASYNPLEX implementation uses a technique developed by Gill et al.[14] called EXPAND, which allegedly prevents cycling and improves numerical stability. Koberstein's thesis [32] also includes a survey of a number of ways to handle stability. Such techniques should be studied.

4.3.4 Parallel linear algebra operations

It is possible to utilise parallel BLAS libraries to parallelise the linear algebra operations themselves. Such an approach was tried on the CUDA architecture by Spampinato[46] (see Section 2.1.7), but the speedups were limited. Furthermore, as long as the number of threads is limited (such as it is on Cell/BE), it may be difficult to combine with an algorithm that itself is parallel (and therefore requires several threads of its own), such as ASYNPLEX.

4.3.5 Loop unrolling

Loop unrolling consists of duplicating the body of a loop such that several iterations are performed sequentially inside the loop body. This reduces the number of jumps and may yield good speedups in short, tight loops. Manual loop unrolling is usually not necessary, as compilers can be instructed to perform this operation automatically, but they may not always succeed. Unfortunately, the price that necessarily must be paid when loops are unrolled is an increase in code size, which is undesirable on Cell/BE.

4.3.6 Unit testing

While one might argue that testing an LP solver by running it against a collection of large data sets provides sufficient evidence that the implementation is correct, one will gain even more confidence in the implementation by creating unit tests. Any decent programmer knows how to structure a program by breaking it down into functions, each performing a limited, well-defined part of the overall task. Unit testing, on the other hand, is often neglected, even though it is highly beneficial during development[2] There is a lot of literature on the subject (a good introductory book is [22], and [38] is a more thorough work), but the basic idea is simple: write code that tests other code. This is fairly straightforward to do as long as the code is partitioned into functions in a reasonable manner. Code should be written to test each nontrivial function for a number of different parameter combinations.

[2] As with many other good practices, unit testing is easier to preach than to practice, as evidenced by the lack of unit testing in this project.

Another important aspect is that unit testing gives *regression testing* for free. If one discovers a bug, one should immediately add a test that demonstrates the bug *before* one fixes the code. That way, one can easily demonstrate that the bug has been fixed, and since this test is now a part of the test suite (*all* of which should be run after each change to *any* code) it will immediately discover the bug if it resurfaces — in large applications, bugs in one part of the code can often be triggered by a change in distante part of the code.

While some of these considerations are most relevant for commercial software companies, researchers might also find that unit testing provides a useful safety net.

4.3.7 Overlays

By using overlays (see Section 2.2.2.5) wisely, it may be possible to significantly reduce the size of the program code that at any time resides in the local store, thereby freeing up space for data. For instance, because of the SPMD approach mandated by CML, the code for all four ASYNPLEX processes must reside in the same program. With overlays, each SPE will only need to load the code for the process type that it is responsible for. Furthermore, since the code for each phase is slightly different, but the phase change only occurs once, it should be beneficial to only load the code for one phase at a time. We doubt that even more fine-grained use of overlays will help, unless the linear program is extremely large, in which case it may also be possible to use overlays for the code of the individual linear algebra operations that are executed in each iteration.

4.3.8 Representation of sparse matrices

Sparse matrices and vectors can be represented in numerous ways; Shahnaz et al.[45] give a good review of different storage schemes. Several operations in a linear solver will depend on the choice of such a representation. If one takes care to place the code for each such operation in a separate function, only a modest amount of work will be required to create implementations of several storage schemes (with the added benefit that these implementations can be tested separately, and as long as they work, the entire solver will still work). Then, one can measure how performance is impacted by the choice of storage scheme. The first alternative representation to try might be the *jagged diagonal storage*, which, according to [45], is "specially tailored for sparse matrix-vector multiplications", and its variation *transposed jagged diagonal storage*, which is "suitable for parallel and distributed processing".

It should be noted that some formats are intended for general matrices, while others make assumptions about the distribution of nonzeroes — the latter category may be risky to use internally in the solver, since one cannot tell in advance

what kind of patterns might emerge in the intermediate matrices produced in the course of the algorithm. Vanderbei's implementation uses the Compressed Column Storage format described in Section 2.3.2.

4.3.9 Vectorisation

As mentioned in Section 2.2.2.1, utilising vector operations is essential in order to obtain the high computational throughput that is promised by the Cell/BE. While vectorisation of dense matrix-vector operations is fairly trivial (as seen in our parallel standard simplex solver), putting vectors to good use in sparse operations is much harder. For instance, vectorisation of a simple addition of (mathematical) vectors will require the opportunity to add four adjacent numbers to four other adjacent numbers simultaneously, but with sparse representations, adjacent numbers in one vector may not correspond to adjacent numbers (or any numbers at all) in the other vector.

One approach may be the following: for each nonzero number, store all four elements of that column that would be located in the same `vector` in a dense column major representation (even if the other three are zeroes) — if the element at row i is nonzero, we would store all elements from $\lfloor \frac{i}{4} \rfloor \cdot 4$ through $\lfloor \frac{i}{4} \rfloor \cdot 4 + 3$. This would permit operations on four adjacent numbers — but only if there is a matching `vector` in the other vector. Thus, the gains from this approach may be rather limited. Furthermore, it would come at the cost of an increase in the storage requirements, which may be detrimental since it would increase the traffic on the Cell/BE bus. When using the compressed column storage format as described in Section 2.3.2, the required space would increase from $2k + 1$ elements to $5k + 1$ elements (it is sufficient to store the row index of each `vector` , so only the value array would quadruple its size) — but only in the worst case of a vector having k nonzeroes with none of them spaced closer than four elements apart. For an $m \times n$-matrix containing k nonzeroes it would increase from $2k + n + 3$ elements to at most $5k + n + 3$ elements.

4.3.10 Autotuning

Autotuning is the process of experimentally finding good values for compile-time constants such as block sizes for data transfer and matrix multiplication[3]. As implied by the word "auto", the experimentation is performed by another program, which repeatedly recompiles the target program with new parameters and runs benchmarks; the best parameter combination is then used for the final compilation.

[3] A typical matrix multiplication is to divide each matrix into blocks and perform the multiplication blockwise, with the goal of having each block stay in the cache for as long as possible.

A well-known software product that utilises autotuning is ATLAS, which is a BLAS library which can be automatically optimised for any architecture.

However, the benefits from autotuning may be smaller on Cell/BE than on regular computers, since one of the points of the Cell/BE architecture is that it should be possible to produce code that is predictably good because it utilises the manually-controlled memory hierarchy well — but this, of course, is complicated, and using autotuning may be simpler in some situations. A good Cell/BE-specific target for autotuning may be buffer sizes for triple buffering.

4.3.11 Triple buffering

The double/triple/quadruple buffering technique is described in Section 2.2.2.4. We believe this to be the by far most important optimisation to consider — we might even go as far as to state that it is a *necessity* rather than an optimisation. Triple buffering is necessary in order to support data sets of realistic sizes — a solver that can only handle data sets of less than a hundred kilobytes (which is what remains when the large solver code has taken its share of the local store) is of little practical use.

Chapter 5

Conclusion

The purpose of this project was to explore how linear programming algorithms, primarily variations of the simplex method, might be parallelised and implemented on the Cell Broadband Engine, a multicore processor with an innovative architecture. To the surprise of both the author and his advisor, the various simplex methods turned out to be exceedingly difficult to implement, even on a regular computer and without parallelisation — a fact which we later learned is well-known within the mathematical optimisation community. We are astonished that well-known books on the subject of linear programming do not present this simple fact ("the standard simplex method is virtually useless in practice") more clearly; knowing this would have saved us a considerable amount of time. However, we ourselves are to blame for not having contacted professionals in the field much earlier than we did. One of the most important lessons we have drawn from the project is that building an industrial-strength LP solver is a vast amount of work and must only be undertaken by someone who has extensive knowledge of both programming *and* numerics. The author only possessed skills in the former field and selected this project in the belief that the major challenge would be the Cell/BE devlopment alone. Most likely, someone more skilled in numerics would have been able to produce better results, and we believe that the investigation of the Cell/BE as a platform for linear programming should continue in spite of the poor results we have achieved.

The products of this project, besides this report, include a number of LP solvers (for both x86, PPE only, and PPE with SPEs) employing the standard simplex method and an algorithm by Hall and McKinnon called ASYNPLEX. We did not get the time to finish the Cell/BE ASYNPLEX implementation, and only the x86 ASYNPLEX implementation displays a reasonable amount of numerical stability. We have performed a few experiments, from which we drew the following conclusions:

- SIMD instructions give little speedup in the standard simplex method (at

most 38%), most likely due to the low arithmetic intensity of the method.

- Overlapping communication and computation is absolutely necessary in order to gain even acceptable performance on the SPEs — in our implementation, the SPEs were stalled waiting for data for around 94% of the time.
- ASYNPLEX on x86 will require careful optimisation and possibly a better messaging system in order to outperform the sequential solver on which our implementation is based. Currently, it is 3–6 times slower, and adding iteration processes only helps for large data sets.

5.1 Future work

The following is a summary of our discussion in Section 4.3. We recommend those who are going to continue this project to investigate the following areas:

- Triple (or quadruple) buffering for supporting data sets of nontrivial size — more of a necessity than an optimisation.
- Implementing existing techniques for anti-cycling and numerical stability.
- Unit testing.
- Parallelisation of the individual linear algebra operations.
- Mixed precision computations for obtaining double precision accuracy while primarily using single precision computation.
- Unrolling tight loops to minimise pipeline stalls (while paying attention not to increase code size too much).
- Using overlays to free up more space for data buffers in the SPE local stores, and to allow for bigger and more complex solvers.
- Experimenting with different representations of sparse matrices (this will require a lot of coding, since the linear algebra operations are dependent on the matrix representation).
- Vectorisation of the matrix operations; the opportunities for this will depend on the matrix representation.
- Autotuning, for determining good values for e.g. triple buffer sizes.

In addition, interior point methods can be investigated. However, looking at our experiences from this project, we believe this to be enough subject matter for at least one full Ph.D. thesis.

Bibliography

[1] M. ÅLIND, M. V. ERIKSSON, AND C. W. KESSLER, *BlockLib: A Skeleton Library for Cell Broadband Engine*, in IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering, New York, NY, USA, 2008, ACM, pp. 7–14. [cited at p. 37]

[2] G. AMDAHL, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, in Proceedings of the AFIPS spring joint computer conference, 1967, pp. 483–485. [cited at p. 38]

[3] E. D. ANDERSEN AND K. D. ANDERSEN, *A parallel interior-point algorithm for linear programming on a shared memory machine*, Tech. Rep. 1998008, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), January 1998. [cited at p. 30, 66]

[4] H. ANDERSSON AND M. CHRISTIANSEN, *(Private e-mail correspondence)*, April 2009. [cited at p. 46, 49]

[5] K. ASANOVÍC, R. BODIK, B. CATANZARO, J. GEBIS, P. HUSBANDS, K. KEUTZER, D. PATTERSON, W. PLISHKER, J. SHALF, S. WILLIAMS, AND K. YELICK, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences — University of California at Berkeley, December 2006. [cited at p. 31, 39]

[6] H. Y. BENSON AND D. F. SHANNO, *An exact primal-dual penalty method approach to warmstarting interior-point methods for linear programming*, Computational Optimization and Applications, 38 (2007), pp. 371–399. [cited at p. 17]

[7] R. E. BIXBY AND A. MARTIN, *Parallelizing the Dual Simplex Method*, INFORMS Journal on Computing, 12 (2000), pp. 45–56. [cited at p. 30]

[8] T. H. CORMEN, C. R. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, McGraw-Hill Science/Engineering/Math, 2nd ed., 2003. [cited at p. 3, 16, 43, 45, 46]

[9] G. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963. [cited at p. 7, 17]

[10] D. P. DOBKIN, R. J. LIPTON, AND S. P. REISS, *Linear programming is log-space hard for P*, Information Processing Letters, 8 (1979), pp. 96–97. [cited at p. 16]

[11] Å. ELDHUSET, *Edge detection on GPUs using CUDA*. Fall project report, Norwegian University of Science and Technology, January 2009. [cited at p. 38]

[12] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979. [cited at p. 4]

[13] D. GAY, *Electronic mail distribution of linear programming test problems*, COAL Newsletter, (1985), pp. 10–12. [cited at p. 37]

[14] P. E. GILL, W. MURRAY, M. A. SAUNDERS, AND M. H. WRIGHT, *A practical anti-cycling procedure for linearly constrained optimization*, Mathematical Programming, 45 (1989), pp. 437–474. [cited at p. 67]

[15] J. GONDZIO AND A. GROTHEY, *A New Unblocking Technique to Warmstart Interior Point Methods based on Sensitivity Analysis*, SIAM Journal on Optimization, 19 (2008), pp. 1184–1210. [cited at p. 17]

[16] R. GREENLAW, H. J. HOOVER, AND W. L. RUZZO, *Limits to parallel computation: P-completeness theory*, Oxford University Press, Inc., New York, NY, USA, 1995. [cited at p. 16, 17]

[17] J. A. J. HALL, *Towards a practical parallelisation of the simplex method*, Optimization Online, (2005). [cited at p. 66]

[18] J. A. J. HALL AND K. I. M. MCKINNON, *PARSMI, a Parallel Revised Simplex Algorithm Incorporating Minor Iterations and Devex Pricing*, in PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, Springer-Verlag, 1996, pp. 359–368. [cited at p. 30]

[19] J. A. J. HALL AND K. I. M. MCKINNON, *ASYNPLEX, an asynchronous parallel revised simplex algorithm*, Annals of Operations Research, 81 (1998), pp. 27–50. [cited at p. 22, 23, 24, 29, 30, 52]

[20] F. S. HILLIER AND G. J. LIEBERMAN, *Introduction to Operations Research*, McGraw-Hill Science/Engineering/Math, July 2004. [cited at p. 5]

[21] J. K. HO AND R. P. SUNDARRAJ, *On the efficacy of distributed simplex algorithms for linear programming*, Computational Optimization and Applications, 3 (1994), pp. 349–363. [cited at p. 30, 49]

[22] A. HUNT AND D. THOMAS, *Pragmatic Unit Testing in C# with Nunit*, The Pragmatic Programmers, 2004. [cited at p. 67]

[23] IBM, *Cell Broadband Engine Programming Handbook, Version 1.1*. [cited at p. 31, 32, 34]

[24] ——, *Software Development Kit for Multicore Acceleration, Version 3.0 — Programming Tutorial*. [cited at p. 31, 32, 33]

[25] ——, *Software Development Kit for Multicore Acceleration, Version 3.1 — Programmer's Guide*. [cited at p. 31, 37]

[26] J. A. KAHLE, M. N. DAY, H. P. HOFSTEE, C. R. JOHNS, T. R. MAEURER, AND D. SHIPPY, *Introduction to the cell multiprocessor*, IBM J. Res. Dev., 49 (2005), pp. 589–604. [cited at p. 30, 31, 32]

[27] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, in STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, New York, NY, USA, 1984, ACM, pp. 302–311. [cited at p. 25]

[28] G. KARYPIS, A. GUPTA, AND V. KUMAR, *A parallel formulation of interior point algorithms*, in Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, New York, NY, USA, 1994, ACM, pp. 204–213. [cited at p. 30, 66]

[29] L. G. KHACHIYAN, *A Polynomial Algorithm in Linear Programming*, Doklady Akademiia Nauk SSSR, 224 (1979), pp. 1093–1096. (English translation in Soviet Mathematics Reports 20:1 (1979), pp. 191–194). [cited at p. 16, 25]

[30] V. KLEE AND G. J. MINTY, *How good is the simplex algorithm?*, in Inequalities, O. Shisha, ed., vol. III, Academic Press, New York, 1972, pp. 159–175. [cited at p. 16]

[31] D. E. KNUTH, *Structured Programming with go to Statements*, ACM Computing Surveys, 6 (1974), pp. 261–301. [cited at p. 47]

[32] A. KOBERSTEIN, *The Dual Simplex Method — Techniques for a fast and stable implementation*, PhD thesis, University of Paderborn, 2005. [cited at p. 67]

[33] M. KRISHNA, A. KUMAR, N. JAYAM, G. SENTHILKUMAR, P. K. BARUAH, R. SHARMA, S. KAPOOR, AND A. SRINIVASAN, *A Synchronous Mode MPI Implementation on the Cell BE Architecture*, in ISPA, 2007, pp. 982–991. [cited at p. 36]

[34] A. KUMAR, G. SENTHILKUMAR, M. KRISHNA, N. JAYAM, P. K. BARUAH, R. SHARMA, A. SRINIVASAN, AND S. KAPOOR, *A Buffered-Mode MPI Implementation for the Cell BE Processor*, in ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I, Springer-Verlag, 2007, pp. 603–610. [cited at p. 36]

[35] J. KURZAK, A. BUTTARI, AND J. DONGARRA, *Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization*, IEEE Transactions on Parallel and Distributed Systems, 19 (2008), pp. 1175–1186. [cited at p. 66]

[36] J. KURZAK AND J. DONGARRA, *Implementation of mixed precision in solving systems of linear equations on the Cell processor: Research Articles*, Concurrency and Computation: Practice & Experience, 19 (2007), pp. 1371–1385. [cited at p. 34, 66]

[37] I. MAROS, *Computational Techniques of the Simplex Method*, Kluwer Academic Publishers, Norwell, MA, USA, 2002. [cited at p. 24, 45, 46, 47, 49]

[38] S. MCCONNELL, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 2nd ed., 2004. [cited at p. 67]

[39] B. MONIEN AND J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 66]

[40] L. NATVIG, *Evaluating Parallel Algorithms: Theoretical and Practical Aspects*. Dr.Ing. thesis, The Norwegian Institute of Technology, 1990. [cited at p. 17]

[41] J. P. PEREZ, P. BELLENS, R. M. BADIA, AND J. LABARTA, *CellSs: making it easier to program the cell broadband engine processor*, IBM Journal of Research and Development, 51 (2007). [cited at p. 37]

[42] W. PRESS, S. TEUKOLSKY, W. VETTERLING, AND B. FLANNERY, *Numerical Recipes in C*, Cambridge University Press, 2nd ed., 1992. [cited at p. 28, 46]

[43] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, AND B. P. FLANNERY, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 3rd ed., August 2007. [cited at p. 28, 46]

[44] J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 66]

[45] R. SHAHNAZ, A. USMAN, AND I. CHUGHTAI, *Review of Storage Techniques for Sparse Matrices*, in 9th International Multitopic Conference, IEEE INMIC 2005, December 2005, pp. 1–7. [cited at p. 37, 68]

[46] D. G. SPAMPINATO, *Linear Optimization on Modern GPUs*, in Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), 2009. [cited at p. 29, 67]

[47] R. J. VANDERBEI, *Linear Programming: Foundations and Extensions*, Springer, 2nd ed., 2001. [cited at p. 3, 12, 13, 14, 16, 17, 20, 21, 22, 29, 43, 45, 46]

[48] H. WEI AND J. YU, *Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-core Chip*, in IWOMP '07: Proceedings of the 3rd international workshop on OpenMP, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 129–133. [cited at p. 37]

[49] R. WUNDERLING, *Paralleler und objektorientierter Simplex-Algorithmus*, PhD thesis, Technische Universtät Berlin, Fachbereich Mathematik (G. Ziegler) and ZIB (M. Grötschel), December 1996. [cited at p. 29, 30]

[50] G. YARMISH, *A Distributed Implementation of the Simplex Method*, PhD thesis, Polytechnic University, March 2001. [cited at p. 28, 44]

[51] G. YARMISH AND R. V. SLYKE, *retroLP, An Implementation of the Standard Simplex Method*, Tech. Rep. TR-CIS-2001-05, Polytechnic University, August 2001. [cited at p. 28]

[52] E. A. YILDIRIM AND S. J. WRIGHT, *Warm-Start Strategies in Interior-Point Methods for Linear Programming*, SIAM Journal on Optimization, 12 (2002), pp. 782–810. [cited at p. 17]

[53] K. YOSHII, *Time-stamp counter*. http://www.mcs.anl.gov/~kazutomo/rdtsc.html. [cited at p. 58, 92]

Appendices

Appendix A

Code

We now list the most interesting parts of our code. The full source code for all of our solvers (including `Makefile`s and, for some solvers, project files for Eclipse and Visual Studio) is located in the source code attachment uploaded to the thesis submission system of the Department of Computer and Information Science.

A.1 Sequential standard simplex method for x86 and Cell/BE

This is the full source code. It can be compiled in several versions:

- For x86, using `float`: run `g++ -O3 -Wall *.cpp -o standard_simplex_x86_float -DTYPE=float`
- For x86, using `double`: run `g++ -O3 -Wall *.cpp -o standard_simplex_x86_float -DTYPE=double`
- For x86, using GMP: run `g++ -O3 -Wall *.cpp -o standard_simplex_x86_gmp -DTYPE=mpq_class -lgmpxx -lgmp -DUSE_GMP`
- For Cell/BE (but uses only the PPE), using `float` and SIMD instructions: run `make`
- For Cell/BE, using `float` and no SIMD instructions: edit the `Makefile` and remove `-DUSE_SIMD`, and run `make`

Note that the code highlighting package we use (`listings`) erroneously highglights the `vector` class from the Standard Template Library, thinking that it is the `vector` keyword for SIMD data types.

The solver expects MPS files with no `BOUNDS` or `RANGES` sections as the first command line parameter when it is executed.

`Matrix.h` and `Matrix.cpp`

A class for representing dense matrices where the size of the physical memory buffer for each row is a multiple of 16 bytes, so that it supports SIMD operations internally. The actual matrix with

can be anything.

```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>
#include "gmpInterop.h"

class Matrix {
  friend std::ostream & operator << (std::ostream &, const Matrix &);
public:
  Matrix(int, int);
  Matrix(int rows, int cols, TYPE * data);
  Matrix(int, int, bool);
  Matrix(const Matrix &);
  ~Matrix();
  int getRows() { return rows; }
  int getCols() { return cols; }
  int getPhysicalCols() { return physicalCols; }

#ifdef VOLATILE_DATA
  // When this class is being used in the parallel standard simplex method for
       Cell, the data buffer must be marked as volatile because it will be the
       target of DMA transfers
  volatile
#endif
  TYPE * getData() { return data; }

  inline
#ifdef VOLATILE_DATA
  volatile
#endif
  TYPE & operator () (int r, int c) { return data[r * physicalCols + c]; }

  inline TYPE operator () (int r, int c) const { return data[r * physicalCols
     + c]; }
  void swapRows(int firstRow, int secondRow);
  void multiplyRow(int row, TYPE factor);
  void addRows(int sourceRow, int destinationRow, TYPE factor);
  void print(const std::vector<int> & basic, const std::vector<int> & nonbasic
     );
private:
  int rows;
  int cols;
  int physicalCols;
#ifdef VOLATILE_DATA
  volatile
```

```
#endif
  TYPE * data;
};

std::ostream & operator << (std::ostream &, const Matrix &);

// Used by addRows to increment one number by another, with support for two
   different "zeroing rules", controlled by defines:
// - set to zero if result is below ZEROING_RULE_EPSILON
// - set to zero if the ratios of each of the operands to the result is
   greater than ZEROING_RULE_RATIO
// - not defining either of these will simply result in a regular a += b
   operation
#ifdef VOLATILE_DATA
inline void incr(volatile TYPE & a, const TYPE & b) {
#else
inline void incr(TYPE & a, const TYPE & b) {
#endif
  #ifdef ZEROING_RULE_EPSILON
    a += b;
    if (ABS(a) <= ZEROING_RULE_EPSILON) {
      a = 0;
    }
  #else
    #ifdef ZEROING_RULE_RATIO
      TYPE result = a + b;
      if (result == 0 || (ABS(a / result) >= ZEROING_RULE_RATIO && ABS(b /
          result) >= ZEROING_RULE_RATIO)) {
        a = 0;
      }
      else {
        a = result;
      }
    #else
      a += b;
    #endif
  #endif
}

#endif
```

```
#include "Matrix.h"
#define VECTOR_WIDTH (16 / sizeof(TYPE))
#define ALIGN_LOG2 4
#define ROUND_UP_MULTIPLE(x, m) (((x) + (m) - 1) / m * m) // Returns x rounded
    up to the nearest multiple of m
```

```
#ifdef __powerpc__
  // On Cell, we need these includes
  // Note: __powerpc__ should perhaps be replaced by something more Cell-
     specific in case one wants to compile this on a PowerPC that is not a
     Cell PPE
  #include <altivec.h>
  #include <libmisc.h>
#else
  #include <cstdlib>
  // On x86, there is no malloc_align and no need for it either
  #define malloc_align(size, alignment) malloc(size)
  #define free_align(buffer) free(buffer)
#endif

#ifdef USE_SIMD
  #ifdef USE_GMP
    #error "Cannot use GMP with SIMD operations"
  #endif
#endif

using namespace std;

Matrix::Matrix(int rows, int cols) {
  this->rows = rows;
  this->cols = cols;
#ifdef USE_GMP
  // GMP's types are classes and require their constructors to be called, so
     we need to use new.
  // If necessary, it is possible to combine malloc_align and GMP through the
     use of 'placement new'.
  this->physicalCols = cols;
  this->data = new mpq_class[rows * this->physicalCols];
#else
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
     ALIGN_LOG2);
#endif
  for (int i = 0; i < rows * this->physicalCols; ++i)
    this->data[i] = 0;
}

Matrix::Matrix(int rows, int cols, TYPE * data) {
  this->rows = rows;
  this->cols = cols;
#ifdef USE_GMP
  this->physicalCols = cols;
  this->data = new TYPE[rows * this->physicalCols];
```

```
#else
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
      ALIGN_LOG2);
#endif
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c)
      this->data[r * this->physicalCols + c] = data[r * cols + c];
    for (int c = cols; c < this->physicalCols; ++c)
      this->data[r * this->physicalCols + c] = 0;
  }
}

Matrix::Matrix(int rows, int cols, bool identity) {
  this->rows = rows;
  this->cols = cols;
#ifdef USE_GMP
  this->physicalCols = cols;
  this->data = new TYPE[rows * this->physicalCols];
#else
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
      ALIGN_LOG2);
#endif
  for (int i = 0; i < rows * this->physicalCols; ++i)
    this->data[i] = 0;
  if (identity && rows == cols) {
    for (int i = 0; i < rows; ++i) {
      (*this)(i, i) = 1;
    }
  }
}

Matrix::Matrix(const Matrix & source) {
  this->rows = source.rows;
  this->cols = source.cols;
  this->physicalCols = source.physicalCols;
#ifdef USE_GMP
  this->data = new TYPE[source.rows * source.physicalCols];
#else
  this->data = (TYPE*)malloc_align(source.rows * source.physicalCols * sizeof(
      TYPE), ALIGN_LOG2);
#endif
  for (int i = 0; i < source.rows * source.physicalCols; ++i)
    this->data[i] = source.data[i];
}
```

```
Matrix::~Matrix() {
#ifdef USE_GMP
  delete [] data;
#else
  free_align((float*)data);
#endif
}

// Prints the entire matrix
ostream & operator << (ostream & out, const Matrix & matrix) {
  out << "=== " << matrix.rows << " x " << matrix.cols << " @ " << matrix.data
      << " ===" << endl;
  for (int r = 0; r < matrix.rows; ++r) {
    out << matrix(r, 0);
    for (int c = 1; c < matrix.cols; ++c)
      out << ' ' << matrix(r, c);
    out << endl;
  }
  out << "======" << endl;
  return out;
}

// A more advanced print operation that prints only nonzero entries, along
   with variable names
void Matrix::print(const vector<int> & basic, const vector<int> & nonbasic) {
  cout << "=== " << rows << " x " << cols << " @ " << data << " ===" << endl;
  for (int r = 0; r < rows; ++r) {
    if (r == 0)
      cout << "z = ";
    else
      cout << "x" << basic[r - 1] << " = ";
    cout << (*this)(r, 0);
    for (int c = 1; c < cols; ++c)
      if ((*this)(r, c) != 0)
        cout << " " << (*this)(r, c) << "x" << nonbasic[c - 1];
    cout << endl;
  }
  cout << "======" << endl;
}

// No point in using SIMD here; this function is used very rarely
void Matrix::swapRows(int firstRow, int secondRow) {
  if (firstRow == secondRow) return;
  for (int j = 0; j < cols; ++j) {
    TYPE tmp = (*this)(firstRow, j);
    (*this)(firstRow, j) = (*this)(secondRow, j);
    (*this)(secondRow, j) = tmp;
```

```
  }
}



#ifdef USE_SIMD

  void Matrix::multiplyRow(int row, TYPE factor) {
    if (factor == 1) return;
    vector TYPE zero_v = (vector TYPE){0.0f, 0.0f, 0.0f, 0.0f};
    vector TYPE factor_v = (vector TYPE){factor, factor, factor, factor}; //
        Wanted to use vec_splat(vec_lde(0, &factor), 0) here, but might have
        misunderstood the syntax
    vector TYPE * data_v = (vector TYPE *)(data + row * physicalCols);
    for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
      data_v[j] = vec_madd(data_v[j], factor_v, zero_v);
    }
  }

  void Matrix::addRows(int sourceRow, int destinationRow, TYPE factor) {
    if (factor == 0) return;
    vector TYPE factor_v = (vector TYPE){factor, factor, factor, factor}; //
        As above
    vector TYPE * source_v = (vector TYPE *)(data + sourceRow * physicalCols);
    vector TYPE * destination_v = (vector TYPE *)(data + destinationRow *
        physicalCols);
    for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
      destination_v[j] = vec_madd(source_v[j], factor_v, destination_v[j]);
    }
  }

#else

  void Matrix::multiplyRow(int row, TYPE factor) {
    if (factor == 1) return;
    for (int j = 0; j < cols; ++j) {
      (*this)(row, j) *= factor;
    }
  }

  void Matrix::addRows(int sourceRow, int destinationRow, TYPE factor) {
    if (factor == 0) return;
    for (int j = 0; j < cols; ++j) {
      incr((*this)(destinationRow, j), (*this)(sourceRow, j) * factor);
    }
  }
```

```
#endif
```

mps.h and mps.cpp

An MPS parser. See notes in Appendix A.5 and Appendix A.5.

```
#ifndef MPS_H
#define MPS_H

#include <iostream>
#include <vector>
#include <string>

std::vector<std::vector<TYPE> > parse(std::istream & input);
std::vector<std::vector<TYPE> > parse(const std::vector<std::string> & lines);

#endif
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <stdexcept>
#include <sstream>
#include <cctype>
#include "gmpInterop.h"

using namespace std;

enum RowType {
  LESS_THAN,
  EQUAL_TO,
  GREATER_THAN,
  OBJECTIVE
};

string rowTypeLabels[] = {"L", "E", "G", "N"};

class Equation {
public:
  string label;
  RowType type;
  map<string, TYPE> values;
  int index;
  TYPE rhs;
```

```
  Equation(string label, string type, int index) {
    this->label = label;
    if (type == "L")
      this->type = LESS_THAN;
    else if (type == "E")
      this->type = EQUAL_TO;
    else if (type == "G")
      this->type = GREATER_THAN;
    else if (type == "N")
      this->type = OBJECTIVE;
    else
      throw invalid_argument("");
    this->index = index;
    this->rhs = 0;
  }
};

string strip(string line) {
  int start = 0, end = (int)line.size() - 1;
  while (start < (int)line.size() && isspace(line[start])) ++start;
  while (end >= 0 && isspace(line[end])) --end;
  if (end < start)
    return "";
  else
    return line.substr(start, end - start + 1);
}

vector<string> split(string line) {
  stringstream ss(line);
  vector<string> items;
  string item;
  while (ss >> item) {
    items.push_back(item);
  }
  return items;
}

vector<vector<TYPE> > parse(const vector<string> & lines) {
  map<string, Equation *> equations;
  vector<string> columnLabels;
  map<string, int> columnIndices;
  vector<vector<TYPE> > tableau;
  unsigned int i = 0;
  int objectiveIndex = -1;
  while (i < lines.size()) {
    string line = lines[i];
    string header = strip(line);
```

```
i++;
if (line[0] == '*' || line[0] == ' ' || line.substr(0, 4) == "NAME") {
  continue;
}
else if (strip(line) == "ENDATA") {
  break;
}
else if (strip(line) == "ROWS") {
  int rowIndex = 0;
  while (lines[i][0] == ' ') {
    vector<string> items = split(lines[i]);
    Equation * eqn = new Equation(strip(items[1]), strip(items[0]),
        rowIndex);
    if (eqn->type == OBJECTIVE)
      objectiveIndex = rowIndex;
    equations[eqn->label] = eqn;
    rowIndex++;
    i++;
  }
}
else if (strip(line) == "COLUMNS") {
  int columnIndex = -1;
  while (lines[i][0] == ' ') {
    vector<string> items = split(lines[i]);
    int lim = (items.size() == 5 ? 2 : 1);
    string columnLabel = strip(items[0]);
    if (columnIndices.find(columnLabel) == columnIndices.end()) {
      columnIndex++;
      columnLabels.push_back(columnLabel);
      columnIndices[columnLabel] = columnIndex;
    }
    for (int j = 0; j < lim; ++j) {
      string rowLabel = strip(items[1 + j * 2]);
      stringstream ss(strip(items[2 + j * 2]));
      TYPE value;
      readNumber(ss, value);
      equations[rowLabel]->values[columnLabel] = value;
    }
    i++;
  }
}
else if (strip(line) == "RHS") {
  while (lines[i][0] == ' ') {
    vector<string> items = split(lines[i]);
    int lim = (items.size() == 5 ? 2 : 1);
    for (int j = 0; j < lim; ++j) {
      string rowLabel = strip(items[1 + j * 2]);
```

```
          stringstream ss(strip(items[2 + j * 2]));
          TYPE value;
          readNumber(ss, value);
          equations[rowLabel]->rhs = value;
        }
        i++;
      }
    }
    else {
      throw invalid_argument("");
    }
  }
  tableau.resize(1); // obj. func.
  for (map<string, Equation *>::iterator eqnIter = equations.begin(); eqnIter
      != equations.end(); ++eqnIter) {
    Equation * eqn = eqnIter->second;
    vector<TYPE> row(columnLabels.size() + 1, 0);
    bool any = false;
    row[0] = eqn->rhs;
    for (map<string, TYPE>::iterator colIter = eqn->values.begin(); colIter !=
        eqn->values.end(); ++colIter) {
      string colLabel = colIter->first;
      row[columnIndices[colLabel] + 1] = -eqn->values[colLabel];
      if (eqn->values[colLabel] != 0)
        any = true;
    }
    if (!any)
      continue;
    if (eqn->type == OBJECTIVE) {
      tableau[0] = row;
    }
    else {
      if (eqn->type == LESS_THAN || eqn->type == EQUAL_TO) {
        tableau.push_back(row);
      }
      if (eqn->type == GREATER_THAN || eqn->type == EQUAL_TO) {
        for (unsigned int j = 0; j < row.size(); ++j)
          row[j] = -row[j];
        tableau.push_back(row);
      }
    }
    delete eqn;
  }

  return tableau;
}
```

```
vector<vector<TYPE> > parse(istream & input) {
  string line;
  vector<string> lines;
  while (getline(input, line)) {
    lines.push_back(line);
  }
  return parse(lines);
}
```

gmpInterop.h and gmpInterop.cpp

Code for facilitating the use of GMP, whose classes must occasionally be treated differently from primitive C++ types.

```
#ifndef GMPINTEROP_H
#define GMPINTEROP_H

#include <iostream>
#ifdef USE_GMP
  #include <gmpxx.h>
  #define ABS(x) abs(x)
#else
  #include <cmath>
  #define ABS(x) fabs(x)
#endif

void readNumber(std::istream & in, TYPE & number);
void printNumber(const TYPE & number);
void printNumberFull(const TYPE & number);

#endif
```

```
#ifdef USE_GMP
  #include <gmpxx.h>
#endif
#include "gmpInterop.h"
using namespace std;

#ifdef USE_GMP
// Only handles floats using regular notation (no 0x, no E) and integers.
// Accepts numbers < 1 without a leading zero.
// The string cannot be empty or contain any spaces.
static void parseRational(const string & str, mpq_class & number) {
  //bool negative = (str[0] == '-');
  size_t dotIndex = str.find('.');
  if (dotIndex == string::npos) {
    number = str;
```

```
    number.canonicalize();
    return;
  }
  string noDot = str.substr(0, dotIndex) + str.substr(dotIndex + 1);
//  size_t firstNonzeroIndex = noDot.find_first_not_of("-0");
//  if (firstNonzeroIndex == string::npos) {
//    number = 0;
//    return;
//  }
//  string noLeadingZeroes = negative ? "-" : "";
//  noLeadingZeroes += noDot.substr(firstNonzeroIndex);
  mpq_class numerator(noDot, 10);
  int power = str.size() - dotIndex - 1;
  mpq_class denominator = 1;
  while (power--) {
    denominator *= 10;
  }
  number = numerator / denominator;
  number.canonicalize();
}
#endif

void readNumber(istream & in, TYPE & number) {
#ifdef USE_GMP
  string str;
  in >> str;
  parseRational(str, number);
#else
  in >> number;
#endif
}

void printNumber(const TYPE & number) {
#ifdef USE_GMP
  cout << number.get_d();
#else
  cout << number;
#endif
}

void printNumberFull(const TYPE & number) {
#ifdef USE_GMP
  cout << number.get_d() << " (approx); " << number << " (exact)";
#else
  cout << number;
#endif
}
```

```
/*
int main() {
  mpq_class a;
  parseRational("0.00001", a);
  cout << a << endl;
  parseRational("10.00001", a);
  cout << a << endl;
  parseRational("12.00000", a);
  cout << a << endl;
  parseRational(".02001", a);
  cout << a << endl;
  parseRational("-.5", a);
  cout << a << endl;
  parseRational("-0.00001", a);
  cout << a << endl;
  parseRational("-120", a);
  cout << a << endl;
  parseRational("-0.0", a);
  cout << a << endl;
  parseRational("-0.0", a);
  cout << a << endl;
  parseRational("0", a);
  cout << a << endl;
  parseRational("-0", a);
  cout << a << endl;
  parseRational("32.1234", a);
  cout << a << endl;
  return 0;
}
*/
```

timer.h

Precise timing module, taken from [53].

```
#ifndef TIMER_H
#define TIMER_H

// Thanks to Kazutomo Yoshii
// http://www.mcs.anl.gov/~kazutomo/rdtsc.html

#if defined(__i386__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int x;
```

```
    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
    return x;
}
#elif defined(__x86_64__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}

#elif defined(__powerpc__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int result=0;
  unsigned long int upper, lower,tmp;
  __asm__ volatile(
                "0:                  \n"
                "\tmftbu   %0           \n"
                "\tmftb    %1           \n"
                "\tmftbu   %2           \n"
                "\tcmpw    %2,%0        \n"
                "\tbne     0b         \n"
                : "=r"(upper),"=r"(lower),"=r"(tmp)
                );
  result = upper;
  result = result<<32;
  result = result|lower;

  return(result);
}

#else

#error "No tick counter is available!"

#endif


/*  $RCSfile:  $   $Author: kazutomo $
 *  $Revision: 1.6 $  $Date: 2005/04/13 18:49:58 $
 */

#endif
```

`TableauSimplex.h` and `TableauSimplex.cpp`

Code for performing a simplex iteration (with pivoting).

```
#ifndef TALBEAUSIMPLEX_H
#define TALBEAUSIMPLEX_H

#include "Matrix.h"
#include <vector>

enum SimplexResult {
  SUBOPTIMAL,
  OPTIMAL,
  INFEASIBLE_OR_UNBOUNDED
};

class TableauSimplex {
public:
  static SimplexResult iteration(Matrix & tableau, std::vector<int> & basic,
      std::vector<int> & nonbasic);
  static void pivot(Matrix & tableau, std::vector<int> & basic, std::vector<
      int> & nonbasic, int leaving, int entering);
};

#endif
```

```
#include "TableauSimplex.h"
#include "gmpInterop.h"
#include <cmath>
#include <vector>

using namespace std;

void TableauSimplex::pivot(Matrix & tableau, std::vector<int> & basic, std::
    vector<int> & nonbasic, int leaving, int entering) {
  cout << "Pivoting: x" << basic[leaving - 1] << " (row " << leaving << ")
      leaves, x" << nonbasic[entering - 1] << " (column " << entering << ")
      enters" << endl;
  TYPE xFactor = tableau(leaving, entering);
  int leavingLabel = basic[leaving - 1];
  basic[leaving - 1] = nonbasic[entering - 1];
  nonbasic[entering - 1] = leavingLabel;

  // Cancel out occurrences of the entering variable
  for (int i = 0; i < tableau.getRows(); ++i) {
    if (i == leaving) continue;
    TYPE factor = -tableau(i, entering) / xFactor;
    TYPE savedColVal = tableau(i, entering);
```

```
    tableau.addRows(leaving, i, factor);
    tableau(i, entering) = savedColVal / xFactor;
  }

  tableau.multiplyRow(leaving, -1 / xFactor);
  tableau(leaving, entering) = 1 / xFactor;
}



SimplexResult TableauSimplex::iteration(Matrix & tableau, vector<int> & basic,
    vector<int> & nonbasic) {
  int n = tableau.getCols() - 1, m = tableau.getRows() - 1;

  // Find entering variable by searching the objective function (row 0) for a
     positive coefficient (disregard the constant in column 0)
  int entering = -1;
  for (int j = 1; j <= n; ++j) {
    if (tableau(0, j) > 0 && (entering == -1 || tableau(0, j) > tableau(0,
       entering) || (tableau(0, j) == tableau(0, entering) && nonbasic[j - 1]
        < nonbasic[entering - 1])))
      entering = j;
  }
  if (entering == -1)
    return OPTIMAL;

  // Find leaving variable by searching the column of the entering variable
     and determine the strictest bound
  int leaving = -1;
  TYPE smallestRatio = -1; // Keep the compiler from complaining about
     uninitialised variables
  for (int i = 1; i <= m; ++i) {
    if (tableau(i, entering) >= 0)
      continue;
    TYPE ratio = -tableau(i, 0) / tableau(i, entering); // The "right hand
       side", tableau(i, 0), is always nonnegative, and we only get here if
       tableau(i, entering) is negative, so 'ratio' will be nonnegative
    if (leaving == -1 || ratio < smallestRatio || (ratio == smallestRatio &&
       basic[i - 1] < basic[leaving - 1])) {
      smallestRatio = ratio;
      leaving = i;
    }
  }
  if (leaving == -1)
    return INFEASIBLE_OR_UNBOUNDED;

  pivot(tableau, basic, nonbasic, leaving, entering);
```

```
  return SUBOPTIMAL;
}
```

simplex.cpp

Driver code — initiates input reading, handles the two phases, and initiates iterations.

```
#include "Matrix.h"
#include "TableauSimplex.h"
#include "mps.h"
#include <cmath>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <fstream>
#include <cstring>
#include <ctime>
#include "timer.h"

#define FEASIBILITY_THRESHOLD 1.0e-5 // abs(x0) must be below this value in
    order for the program to be declared feasible (only applies if Phase I is
    needed)

using namespace std;

int main(int argc, char * argv[]) {
  int rows, cols;
  bool initiallyFeasible = true;
  bool print = argc >= 3 && strcmp(argv[2], "print") == 0;
  istream * input;
  ifstream infile;

  if (argc == 1)
    input = &cin;
  else {
    infile.open(argv[1]);
    input = &infile;
  }
  vector<vector<TYPE> > parsedTableau = parse(*input);
  rows = parsedTableau.size();
  cols = parsedTableau[0].size();
  Matrix A(rows, cols + 1);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      A(r, c) = parsedTableau[r][c];
    }
```

```
  if (r > 0 && A(r, 0) < 0) {
    initiallyFeasible = false;
  }
  A(r, cols) = 1;
}

unsigned long long startTime = rdtsc();

std::vector<int> basic, nonbasic;
// Nonbasic variables are labeled 1 .. n
for (int i = 1; i < cols; ++i)
  nonbasic.push_back(i);
nonbasic.push_back(0); // Phase I variable
// Basic variables are labeled n+1 .. n+m
for (int i = cols; i < cols + rows - 1; ++i)
  basic.push_back(i);

int numIterations = 0;
SimplexResult status = SUBOPTIMAL;

Matrix obj(1, cols); // Saves the original objective function
if (!initiallyFeasible) {
  cout << "Entering Phase I" << endl;
  for (int c = 0; c < cols; ++c) {
    obj(0, c) = A(0, c);
    A(0, c) = 0;
  }
  A(0, cols) = -1; // The goal is to maximize -x0

  int leaving = 1;
  for (int i = 2; i < rows; ++i) {
    if (A(i, 0) < A(leaving, 0))
      leaving = i;
  }
  TableauSimplex::pivot(A, basic, nonbasic, leaving, cols);
  if (print) A.print(basic, nonbasic);
  while ((status = TableauSimplex::iteration(A, basic, nonbasic)) ==
      SUBOPTIMAL) {
    ++numIterations;
    if (print) A.print(basic, nonbasic);
    cout << numIterations << ": " << A(0, 0) << endl;
  }
  if (status == INFEASIBLE_OR_UNBOUNDED || ABS(A(0, 0)) >
      FEASIBILITY_THRESHOLD) {
    cout << "INFEASIBLE" << endl;
    return 0;
  }
```

```
  cout << "PHASE I COMPLETED" << endl;
  if (print) A.print(basic, nonbasic);
}

// Locate x0, which is expected to be nonbasic
int x0 = -1;
for (int i = 0; i < cols; ++i) {
  if (nonbasic[i] == 0) {
    x0 = i + 1;
    nonbasic.erase(nonbasic.begin() + i);
    break;
  }
}
if (x0 == -1) {
  for (int j = 0; j < rows - 1; ++j) {
    if (basic[j] == 0) {
      x0 = j + 1;
      break;
    }
  }
  cout << "x0 is basic, and has value " << A(x0, 0) << " - terminating" <<
     endl;
  // If x0 is basic, but has value 0, it should be possible to continue by
     pivoting it out, but we haven't spent time on this since the program
     usually does not give the right answer anyway
  return 0;
}

// Even if there was no Phase I, we still copy the tableau - this should be
   avoided
Matrix newTableau(rows, cols);
for (int i = 0; i < rows; ++i) {
  for (int j = 0; j < cols; ++j) {
    newTableau(i, j) = A(i, j < x0 ? j : j + 1);
  }
}
if (!initiallyFeasible) {
  if (print) newTableau.print(basic, nonbasic);
  newTableau(0, 0) = obj(0, 0);
  for (int j = 1; j < cols; ++j)
    if (nonbasic[j - 1] < cols)
      newTableau(0, j) = obj(0, nonbasic[j - 1]);
  for (int i = 1; i < rows; ++i) {
    if (basic[i - 1] < cols) {
      newTableau.addRows(i, 0, obj(0, basic[i - 1]));
    }
  }
```

```
  }

  if (print) newTableau.print(basic, nonbasic);
  cout << "Entering phase II" << endl;
  while ((status = TableauSimplex::iteration(newTableau, basic, nonbasic)) ==
      SUBOPTIMAL) {
    ++numIterations;
    if (print) newTableau.print(basic, nonbasic);
    cout << numIterations << ": " << newTableau(0, 0) << endl;
  }
  if (status == INFEASIBLE_OR_UNBOUNDED) {
    cout << "UNBOUNDED" << endl;
    return 0;
  }
  cout << "OPTIMAL" << endl;
  cout << "Optimal value: " << newTableau(0, 0) << endl;
  cout << "Elapsed time (minus input parsing): " << rdtsc() - startTime <<
      endl;
  return 0;
}
```

Makefile

Makefile for compiling the Cell/BE versions. The file must be renamed to simply `Makefile` (unlike what it says under "List of code listings"); the code highlighting package does not seem to handle files without extensions.

```
# Use comments to select if you want SIMD or not
#PROGRAM_ppu    := standard_simplex_ppe_float_serial
#CPPFLAGS_gcc = -DTYPE=float
PROGRAM_ppu   := standard_simplex_ppe_float_simd
CPPFLAGS_gcc  = -DTYPE=float -DUSE_SIMD



INCLUDE     := -I .

INSTALL_DIR   = /tmp
INSTALL_FILES = $(PROGRAM_ppu)

IMPORTS     = -lmisc



ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
```

```
  include ../../../../buildutils/make.footer
endif
```

A.2 Parallel standard simplex method for Cell/BE

This is the full source code, except for several files that are identical to those listed in the previous section: `gmpInterop.h`, `gmpInterop.cpp`, `Matrix.h`, `Matrix.cpp`, `mps.h`, `mps.cpp`, and `timer.h`. They (or a symlink to them, to avoid unnecessary file duplication) should be placed in the top level source code folder of this program.

The solver expects MPS files with no `BOUNDS` or `RANGES` sections and a nonnegative right hand side as the first command line parameter when it is executed. The second command line parameter is optional and can be used to control the number of SPE threads.

`spu/PartialMatrix.h` and `spu/PartialMatrix.cpp`

A class very similar to the `Matrix` class above, but with the property that it only contains a contiguous subset of the rows of the complete matrix. The idea is to give each SPE one part of the entire tableau. When addressing the `PartialMatrix`, all indices are relative to the original matrix.

```
#ifndef PARTIALMATRIX_H
#define PARTIALMATRIX_H

class PartialMatrix {
public:
  PartialMatrix(int, int, int, int, int);
  PartialMatrix(int rows, int cols, int physicalCols, int startRow, int
      containedRows, float * data);
  PartialMatrix(int rows, int cols, int physicalCols, int startRow, int
      containedRows, bool identity);
  PartialMatrix(const PartialMatrix &);
  ~PartialMatrix();
  inline int getRows() const { return rows; }
  inline int getCols() const { return cols; }
  inline int getPhysicalCols() const { return physicalCols; }
  inline int getStartRow() const { return startRow; }
  inline int getContainedRows() const { return containedRows; }
  inline int getRowLimit() const { return startRow + containedRows; }
  inline volatile float * getData() { return data; }
  inline volatile float & operator () (int r, int c) { return data[(r -
      startRow) * physicalCols + c]; }
  inline float operator () (int r, int c) const { return data[(r - startRow) *
       physicalCols + c]; }
  void multiplyRow(int row, float factor);
  void addRows(float factor, int sourceRow, int destinationRow);
```

```
  void addRows(float factor, float * sourceRow, int destinationRow);
  void swapRows(int firstRow, int secondRow);
  void print();
private:
  void init(int rows, int cols, int physicalCols, int startRow, int
     containedRows);

  int rows;
  int cols;
  int physicalCols;
  int startRow;
  int containedRows;
  volatile float * data;
};

#endif
```

```
#include "PartialMatrix.h"
#include <spu_intrinsics.h>
#include <libmisc.h>
#include "../types.h"
#include <stdio.h>

using namespace std;

void PartialMatrix::init(int rows, int cols, int physicalCols, int startRow,
    int containedRows) {
  this->rows = rows;
  this->cols = cols;
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
  this->startRow = startRow;
  this->containedRows = containedRows;
  this->data = (float*)malloc_align(containedRows * physicalCols * sizeof(
     float), ALIGN_QUAD_LOG2);
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = 0;
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows, float * data) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int r = 0; r < containedRows; ++r) {
```

```
    for (int c = 0; c < cols; ++c)
      this->data[r * physicalCols + c] = data[r * cols + c];
    for (int c = cols; c < physicalCols; ++c)
      this->data[r * physicalCols + c] = 0;
  }
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows, bool identity) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = 0;
  if (identity && rows == cols) {
    for (int i = 0; i < containedRows; ++i) {
      data[i * physicalCols + (startRow + i)] = 1;
    }
  }
}

PartialMatrix::PartialMatrix(const PartialMatrix & source) {
  init(source.rows, source.cols, source.physicalCols, source.startRow, source.
      containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = source.data[i];
}

PartialMatrix::~PartialMatrix() {
  free_align((void *)data);
}

void PartialMatrix::print() {
  printf("=== %d x %d: %d rows: [%d, %d) ===\n", rows, cols, containedRows,
      startRow, getRowLimit());
  for (int r = startRow; r < getRowLimit(); ++r) {
    printf("%f", (*this)(r, 0));
    for (int c = 1; c < cols; ++c)
      printf(" %f", (*this)(r, c));
    printf("\n");
  }
  printf("======\n");
}

void PartialMatrix::multiplyRow(int row, float factor) {
  if (factor == 1) return;
  vector float zero_v = (vector float){0.0f, 0.0f, 0.0f, 0.0f};
  vector float factor_v = (vector float){factor, factor, factor, factor};
```

```
  vector float * data_v = (vector float *)(data + (row - startRow) *
     physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    data_v[j] = spu_madd(data_v[j], factor_v, zero_v);
  }
}

void PartialMatrix::addRows(float factor, int sourceRow, int destinationRow) {
  if (factor == 0) return;
  vector float factor_v = (vector float){factor, factor, factor, factor};
  vector float * source_v = (vector float *)(data + (sourceRow - startRow) *
     physicalCols);
  vector float * destination_v = (vector float *)(data + (destinationRow -
     startRow) * physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    destination_v[j] = spu_madd(source_v[j], factor_v, destination_v[j]);
  }
}

// Adds a multiple of a row that is not a part of this PartialMatrix to one of
    the rows that are a part of it
void PartialMatrix::addRows(float factor, float * sourceRow, int
   destinationRow) {
  if (factor == 0) return;
  vector float factor_v = (vector float){factor, factor, factor, factor};
  vector float * source_v = (vector float *)sourceRow;
  vector float * destination_v = (vector float *)(data + (destinationRow -
     startRow) * physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    destination_v[j] = spu_madd(source_v[j], factor_v, destination_v[j]);
  }
}

void PartialMatrix::swapRows(int firstRow, int secondRow) {
  if (firstRow == secondRow) return;
  for (int j = 0; j < cols; ++j) {
    float tmp = (*this)(firstRow, j);
    (*this)(firstRow, j) = (*this)(secondRow, j);
    (*this)(secondRow, j) = tmp;
  }
}
```

spu/SpuTableauSimplex.h and spu/SpuTableauSimplex.cpp

Code for performing a simplex iteration (with pivoting) within an SPE and communicating with the PPE.

```
#ifndef SPUTABLEAUSIMPLEX_H
#define SPUTABLEAUSIMPLEX_H

#include "PartialMatrix.h"
#include "../types.h"

class SpuTableauSimplex {
public:
  static SimplexResult iterate(PartialMatrix & tableau, float * ppuPivotRow,
     uint tagId, int spuIndex);
  static void pivot(PartialMatrix & tableau, float * pivotRow, int leaving,
     int entering);
};

#endif
```

```
#include "SpuTableauSimplex.h"
#include "../types.h"
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include <libmisc.h>
#include <cstdio>

using namespace std;

void SpuTableauSimplex::pivot(PartialMatrix & tableau, float * pivotRow, int
    leaving, int entering) {
  float xFactor = pivotRow[entering];

  // Cancel out occurrences of the entering variable
  for (int i = tableau.getStartRow(); i < tableau.getRowLimit(); ++i) {
    if (i == leaving) continue;
    float factor = -tableau(i, entering) / xFactor;
    float savedColVal = tableau(i, entering);
    tableau.addRows(factor, pivotRow, i);
    tableau(i, entering) = savedColVal / xFactor;
  }

  if (leaving != -1) {
    tableau.multiplyRow(leaving, -1 / xFactor);
    tableau(leaving, entering) = 1 / xFactor;
  }
}

// Communications (* means "all";[^x] means "all except x"):
// Each communication end point is tagged in the code as "comm0" etc.
```

```
// 0. SPU0 -> PPU: entering variable, or optimality
// 1. PPU -> SPU[^0]: entering variable, or termination instruction
// 2. SPU* -> PPU: value and index of leaving variable, or unboundedness (both
     determined locally; PPU determines global choice - let x be the spu that
    'wins')
// 3. PPU -> SPU*: whether this spu 'won', or termination instruction
// 4. SPUx -> PPU: transfer pivot row; spu writes to (ppe reads from) mbox to
    notify
// 5. PPU -> SPU[^x]: ppe writes to (spu reads from) mbox to sync; transfer
    pivot row
SimplexResult SpuTableauSimplex::iterate(PartialMatrix & tableau, float *
    ppuPivotRow, uint tagId, int spuIndex) {
  int n = tableau.getCols() - 1;
  int entering = -1;
  volatile float * pivotRow;
  static volatile float * incomingPivotRowBuffer = (float *)malloc_align(
      tableau.getPhysicalCols() * sizeof(float), ALIGN_QUAD_LOG2);

  if (tableau.getStartRow() == 0) {
    // Find entering variable by searching the objective function (row 0) for
        a positive coefficient (disregard the constant in column 0)
    for (int j = 1; j <= n; ++j) {
      // We are not using Bland's rule here, since the SPEs do not maintain
          the basic/nonbasic lists (they could of course be arranged to do so)
      if (tableau(0, j) > 0 && (entering == -1 || tableau(0, j) > tableau(0,
          entering)))
        entering = j;
    }
    if (entering == -1) {
      spu_write_out_mbox(SIMPLEX_MBOX_OPTIMAL); //comm0
      return OPTIMAL;
    }
    else
      spu_write_out_mbox((uint)entering); //comm0
  }
  else {
    entering = (int)spu_read_in_mbox(); //comm1
    if ((uint)entering == SIMPLEX_MBOX_OPTIMAL)
      return OPTIMAL;
  }

  // Find leaving variable by searching the column of the entering variable
      and determine the strictest bound
  int localLeaving = -1;
  Value32 largestRatio;
  int i = tableau.getStartRow();
```

```
if (i == 0) // Skip objective function row - this also handles the case
    where this partial matrix only contains the objective function, in which
     case SIMPLEX_MBOX_UNBOUNDED will be returned by SPE 0
  i = 1;
// We discovered a little too late that this part of the code is a little
    outdated - we are testing the inverse of the ratio that is described in
    the report, and therefore we must look for the largest ratio rather than
     the smallest. It is more cumbersome, but entirely equivalent to what is
     described in the report.
for (; i < tableau.getRowLimit(); ++i) {
  float ratio;
  if (tableau(i, 0) == 0) {
    if (tableau(i, entering) == 0)
      ratio = 0;
    else if (tableau(i, entering) < 0)
      ratio = INFINITY;
    else
      ratio = -INFINITY;
  }
  else
    ratio = -tableau(i, entering) / tableau(i, 0);
  if (ratio <= 0) continue;
  if (localLeaving == -1 || ratio > largestRatio.floatValue) {
    largestRatio.floatValue = ratio;
    localLeaving = i;
  }
}
//comm2
if (localLeaving == -1) {
  spu_write_out_mbox(SIMPLEX_MBOX_UNBOUNDED);
  spu_write_out_mbox((uint)-1);
}
else {
  spu_write_out_mbox(largestRatio.uintValue);
  spu_write_out_mbox((uint)localLeaving);
}

uint instruction = spu_read_in_mbox(); //comm3
if (instruction == SIMPLEX_MBOX_UNBOUNDED)
  return UNBOUNDED;

if (instruction == SIMPLEX_MBOX_LEAVING_IS_HERE) {
  //comm4 - pivot->ppu
  pivotRow = &tableau(localLeaving, 0);
  spu_mfcdma32((void *)pivotRow, (uint)ppuPivotRow, tableau.getPhysicalCols
      () * sizeof(float), tagId, MFC_PUT_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

```
    spu_write_out_mbox(0U); //sync
  }
  else if (instruction == SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE) {
    //comm5 - ppu->pivot
    pivotRow = incomingPivotRowBuffer;
    spu_read_in_mbox(); //sync
    spu_mfcdma32((void *)pivotRow, (uint)ppuPivotRow, tableau.getPhysicalCols
        () * sizeof(float), tagId, MFC_GET_CMD);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  }
  else {
    printf("Illegal instruction\n");
    return FAILURE;
  }

  pivot(tableau, (float *)pivotRow, instruction ==
     SIMPLEX_MBOX_LEAVING_IS_HERE ? localLeaving : -1, entering);

  return SUBOPTIMAL;
}
```

spu/spu.cpp

Driver code for the SPE — transfers the initial tableau to the SPE, initiates the iterations, and transfers the final tableau back to the PPE.

```
#define SPE_CODE

#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include <libmisc.h>
#include <stdio.h>
# include "../types.h"
#include "PartialMatrix.h"
#include "SpuTableauSimplex.h"

#define MAX_DMA_SIZE 16384

volatile ParameterContext context ALIGNED_QUAD;

void safeDMA(volatile float * localStoreAddress, unsigned int effectiveAddress
    , unsigned int size, unsigned int tagId, unsigned int command) {
  int remainingSize = size;
  while (remainingSize > 0) {
    spu_mfcdma32(localStoreAddress, effectiveAddress, (remainingSize >
       MAX_DMA_SIZE ? MAX_DMA_SIZE : remainingSize), tagId, command);
    remainingSize -= MAX_DMA_SIZE;
```

```
    effectiveAddress += MAX_DMA_SIZE;
    localStoreAddress += MAX_DMA_SIZE / sizeof(float);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  }
}

int main(unsigned long long spuId __attribute__ ((unused)), unsigned long long
    parameter) {
  uint tagId;
  if ((tagId = mfc_tag_reserve()) == MFC_TAG_INVALID) {
    printf("ERROR (%llx): unable to reserve a tag\n", spuId);
    return 1;
  }
  spu_writech(MFC_WrTagMask, -1);

  // Fetch the context and wait
  spu_mfcdma32((void *)&context, (uint)parameter, sizeof(ParameterContext),
     tagId, MFC_GET_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

  int submatrixLength = context.numContainedRows * context.physicalCols;
  PartialMatrix matrix(context.rows, context.cols, context.physicalCols,
     context.startingRow, context.numContainedRows);
  safeDMA(matrix.getData(), (uint)(context.dataOrigin + context.startingRow *
     context.physicalCols), submatrixLength * sizeof(float), tagId,
     MFC_GETB_CMD); // MFC_GETB_CMD can be used for a barrier get, which will
      wait until previous puts have completed - http://publib.boulder.ibm.com
     /infocenter/systems/scope/syssw/index.jsp?topic=/eiccj/tutorial/
     cbet_3optimz.html

  while (SpuTableauSimplex::iterate(matrix, context.ppuPivotRow, tagId,
     context.spuIndex) == SUBOPTIMAL);

  safeDMA(matrix.getData(), (uint)(context.dataOrigin + context.startingRow *
     context.physicalCols), submatrixLength * sizeof(float), tagId,
     MFC_PUT_CMD);

  return 0;
}
```

types.h

Some structs, defines and enums that are needed by several of the other files.

```
#ifndef TYPES_H
#define TYPES_H
```

```
#define VECTOR_WIDTH 4
#define ALIGN_CACHE_WIDTH 128
#define ALIGN_CACHE_LOG2 7
#define ALIGN_QUAD_WIDTH 16
#define ALIGN_QUAD_LOG2 4
#define ALIGNED_CACHE __attribute__ ((aligned(ALIGN_CACHE_WIDTH)))
#define ALIGNED_QUAD __attribute__ ((aligned(ALIGN_QUAD_WIDTH)))
#define ROUND_UP_MULTIPLE(x, m) (((x) + (m) - 1) / (m) * (m))
#define PADDING(actualSize, alignmentWidth) char padding[ROUND_UP_MULTIPLE(
    actualSize, alignmentWidth) - (actualSize)] // Adds a char array to the
    end of a struct in order to make the struct take up a certain number of
    bytes - the first parameter is the combined size of the other struct
    memebers, and the second parameter is the byte boundary on which the
    struct size should be divisible.

// Sentinel values used when an SPE wants to notify the PPE that it did not
    find a good leaving or entering variable, and when the PPE wants to let
    the SPEs know who should perform the pivot. All of these are IEEE 754
    quiet NaNs, which means that they can't accidentally be interpreted as (or
     collide with) valid floating point numbers.
#define SIMPLEX_MBOX_OPTIMAL 0xffffffff
#define SIMPLEX_MBOX_UNBOUNDED 0xfffffffe
#define SIMPLEX_MBOX_LEAVING_IS_HERE 0xfffffffd
#define SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE 0xfffffffc

//#define INFINITY (__builtin_inff())
#ifndef INFINITY
  #define INFINITY 3.4E38f
#endif

typedef unsigned int uint;

struct ParameterContext {
  int rows;
  int cols;
  int physicalCols;
  int startingRow;
  int numContainedRows;
  int spuIndex;
  float * dataOrigin;
  float * ppuPivotRow;
  PADDING(6 * sizeof(int) + 2 * sizeof(float*), ALIGN_QUAD_WIDTH);
};

// For interpreting a 32 bit pattern in different ways.
union Value32 {
  uint uintValue;
```

```
  int intValue;
  float floatValue;
};

enum SimplexResult {
  SUBOPTIMAL,
  OPTIMAL,
  UNBOUNDED,
  CYCLING,
  FAILURE
};

#endif
```

main.cpp

Driver code — initiates input reading, starts the SPE threads, and communicates with the threads.

```
#define PPE_CODE

#include "Matrix.h"
#include "types.h"
#include "timer.h"
#include "mps.h"
#include <libspe2.h>
#include <libmisc.h>
#include <pthread.h>
#include <iostream>
#include <fstream>
using namespace std;

#define MAX_SPE_THREADS 8

extern spe_program_handle_t speProgramHandle; // Must match the name given in
    the SPE makefile

struct PpuThreadData {
  spe_context_ptr_t speContext;
  pthread_t pthread;
  void * argument;
};

SimplexResult simplexIteration(vector<int> & basic, vector<int> & nonbasic,
    int numSpeThreads, spe_context_ptr_t speContexts[]);

void * ppuPthreadFunction(void * argument) {
  PpuThreadData * data = (PpuThreadData *)argument;
```

```
  unsigned int entry = SPE_DEFAULT_ENTRY;
  if (spe_context_run(data->speContext, &entry, 0, data->argument, NULL, NULL)
      < 0) {
    perror("Failed running context");
    exit(1);
  }
  pthread_exit(NULL);
}

int main(int argc, char * argv[]) {
  int numSpeThreads;
  PpuThreadData data[MAX_SPE_THREADS];
  ParameterContext contexts[MAX_SPE_THREADS] ALIGNED_QUAD;
  spe_context_ptr_t speContexts[MAX_SPE_THREADS];

  if (argc < 2) {
    cout << "Must take MPS file name as parameter; the second argument is
       optional and specifies the number of SPEs" << endl;
    return 1;
  }
  ifstream infile(argv[1]);
  vector<vector<float> > parsedTableau = parse(infile);
  int rows = parsedTableau.size();
  int cols = parsedTableau[0].size();
  Matrix matrix(rows, cols);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      matrix(r, c) = parsedTableau[r][c];
    }
    if (r > 0 && matrix(r, 0) < 0) {
      cout << "This prototype does only handle Phase II; must be run on
         problems that are initially feasible (the entire right hand side
         must be nonnegative)" << endl;
      return 1;
    }
  }
  volatile float * pivotRow = (volatile float *)malloc_align(matrix.
     getPhysicalCols() * sizeof(float), ALIGN_QUAD_LOG2);

  unsigned long long startTime = rdtsc();

  int n = cols - 1;
  int m = rows - 1;
  vector<int> basic, nonbasic;
  // Nonbasic variables are labeled 1 .. n
  for (int i = 1; i <= n; ++i)
    nonbasic.push_back(i);
```

```
// Basic variables are labeled n+1 .. n+m
for (int i = n + 1; i <= n + m; ++i)
  basic.push_back(i);

// Determine how many SPE threads we want
if (argc >= 3) {
  numSpeThreads = atoi(argv[2]);
}
else {
  numSpeThreads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
}
if (numSpeThreads < 1)
  numSpeThreads = 1;
if (numSpeThreads > MAX_SPE_THREADS)
  numSpeThreads = MAX_SPE_THREADS;
if (numSpeThreads > matrix.getRows())
  numSpeThreads = matrix.getRows();
cout << "Using " << numSpeThreads << " SPEs" << endl;

// Create and start SPE threads
for (int i = 0; i < numSpeThreads; ++i) {
  contexts[i].rows = matrix.getRows();
  contexts[i].cols = matrix.getCols();
  contexts[i].physicalCols = matrix.getPhysicalCols();
  contexts[i].startingRow = matrix.getRows() / numSpeThreads * i;
  if (i == numSpeThreads - 1)
    contexts[i].numContainedRows = matrix.getRows() - contexts[i].
       startingRow;
  else
    contexts[i].numContainedRows = matrix.getRows() / numSpeThreads;
  contexts[i].spuIndex = i;
  contexts[i].dataOrigin = (float *)matrix.getData(); // Removing 'volatile'
     ness is ok here, since this pointer will just be used by the SPU to
     indicate the MFC target
  contexts[i].ppuPivotRow = (float *)pivotRow;

  if ((data[i].speContext = spe_context_create(0, NULL)) == NULL) {
    perror("Failed creating context");
    exit(1);
  }
  if (spe_program_load(data[i].speContext, &speProgramHandle)) {
    perror("Failed loading program");
    exit(1);
  }
  speContexts[i] = data[i].speContext;
  data[i].argument = &contexts[i];
  if (pthread_create(&data[i].pthread, NULL, &ppuPthreadFunction, &data[i]))
```

```
        {
      perror("Failed creating thread");
      return 1;
    }
  }

  int numIterations = 0;
  while (simplexIteration(basic, nonbasic, numSpeThreads, speContexts) ==
     SUBOPTIMAL) {
  //  cout << "Done with iteration " << ++numIterations << endl;
  }

  // Uncomment this if you want the list of basic/nonbasic variables to be
     printed
  /*
  cout << "Basic:";
  for (uint i = 0; i < basic.size(); ++i)
    cout << " " << basic[i];
  cout << "\nNonbasic:";
  for (uint i = 0; i < nonbasic.size(); ++i)
    cout << " " << nonbasic[i];
  cout << endl;
  */

  // Wait for SPEs to terminate
  for (int i = 0; i < numSpeThreads; ++i) {
    if (pthread_join(data[i].pthread, NULL)) {
      perror("Failed joining thread");
      exit(1);
    }
    if (spe_context_destroy(data[i].speContext) != 0) {
      perror("Failed destroying context");
      exit(1);
    }
  }

  cout << "The optimal solution is " << matrix(0, 0) << endl;
  cout << "Elapsed time (minus input parsing): " << rdtsc() - startTime <<
     endl;
  return 0;
}



// For explanation of communication tags, see spu/SpuTableauSimplex.cpp
SimplexResult simplexIteration(vector<int> & basic, vector<int> & nonbasic,
    int numSpeThreads, spe_context_ptr_t speContexts[]) {
```

```
//comm0
while (!spe_out_mbox_status(speContexts[0]));
uint entering;
spe_out_mbox_read(speContexts[0], &entering, 1);

//comm1
for (int i = 1; i < numSpeThreads; ++i) {
  spe_in_mbox_write(speContexts[i], &entering, 1, SPE_MBOX_ALL_BLOCKING); //
      entering may be SIMPLEX_MBOX_OPTIMAL, in which case SPE code will
     terminate
}
if (entering == SIMPLEX_MBOX_OPTIMAL) {
  return OPTIMAL;
}

//comm2
// The case where the first SPE only contains the objective function will
    not cause problems. That SPE will report SIMPLEX_MBOX_UNBOUNDED since it
     doesn't find a leaving variable, so it will be ignored unless no other
    SPEs find a leaving variable - in which case the entire problem is
    unbounded.
while (!spe_out_mbox_status(speContexts[0]));
float maxLeavingValue = 0; // Initialise with arbitrary value to get rid of
    warning
int maxLeavingSpu = -1;
int maxLeavingIndex = -1;
for (int i = 0; i < numSpeThreads; ++i) {
  Value32 leavingValue, leavingIndex;
  while (!spe_out_mbox_status(speContexts[i]));
  spe_out_mbox_read(speContexts[i], &leavingValue.uintValue, 1);
  while (!spe_out_mbox_status(speContexts[i]));
  spe_out_mbox_read(speContexts[i], &leavingIndex.uintValue, 1);
  //cout << i << " reports " << leavingValue.floatValue << endl;
  if (leavingValue.uintValue != SIMPLEX_MBOX_UNBOUNDED && (maxLeavingSpu ==
      -1 || leavingValue.floatValue > maxLeavingValue)) {
    maxLeavingValue = leavingValue.floatValue;
    maxLeavingIndex = leavingIndex.intValue;
    maxLeavingSpu = i;
  }
}

//comm3
for (int i = 0; i < numSpeThreads; ++i) {
  uint instruction;
  if (maxLeavingSpu == -1)
    instruction = SIMPLEX_MBOX_UNBOUNDED;
  else if (maxLeavingSpu == i)
```

```
      instruction = SIMPLEX_MBOX_LEAVING_IS_HERE;
    else
      instruction = SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE;
    spe_in_mbox_write(speContexts[i], &instruction, 1, SPE_MBOX_ALL_BLOCKING);
  }
  if (maxLeavingSpu == -1) {
    cout << "Unbounded" << endl;
    return UNBOUNDED; //TODO: cleanup
  }

//  cout << "Pivoting: variable in row " << maxLeavingIndex << " (in SPU " <<
   maxLeavingSpu << ") leaves, variable in column " << entering << " enters"
   << endl;

  //comm4 - sync
  uint dummySyncValue;
  while (!spe_out_mbox_status(speContexts[maxLeavingSpu]));
  spe_out_mbox_read(speContexts[maxLeavingSpu], &dummySyncValue, 1);

  //comm5
  for (int i = 0; i < numSpeThreads; ++i) {
    if (maxLeavingSpu != i) {
      spe_in_mbox_write(speContexts[i], &dummySyncValue, 1,
          SPE_MBOX_ALL_BLOCKING);
    }
  }

  // Swap basic and nonbasic variables
  int tmp = basic[maxLeavingIndex - 1];
  basic[maxLeavingIndex - 1] = nonbasic[entering - 1];
  nonbasic[entering - 1] = tmp;

  return SUBOPTIMAL;
}
```

Makefiles

`Makefile` for SPE code (located in the `spe` folder):

```
# Target

PROGRAM_spu   := speProgramHandle # Must match the name of the
   spe_program_handle_t in main.cpp
LIBRARY_embed := matrix_spu.a

IMPORTS     = -lmisc
```

```
# buildutils/make.footer

ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
  include ../../../../buildutils/make.footer
endif
```

`Makefile` for PPE code:

```
# Subdirectories

DIRS         := spu


# Target

PROGRAM_ppu   := standard_simplex_parallel_float_simd


# Local Defines

INCLUDE     := -I .

INSTALL_DIR   = /tmp
INSTALL_FILES = $(PROGRAM_ppu)

IMPORTS     = spu/matrix_spu.a -lspe2 -lpthread -lmisc
CPPFLAGS_gcc  = -DTYPE=float -DVOLATILE_DATA


# buildutils/make.footer

ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
  include ../../../../buildutils/make.footer
endif
```

A.3 ASYNPLEX for x86, based on Vanderbei

We only list the most relevant files here — the files we have created ourselves and those of Vanderbei's files that we have modified heavily. The rest can be found in the attachment. The files are described in Section 3.4.4.2.

Since the development started out in C and our source control system does not handle name changes very gracefully, the file suffixes remain `.c`. The compiler is instructed to compile it as C++ with the `-x c++` flag in the `Makefile`s.

common/message.h

```
#ifndef MESSAGE_H
#define MESSAGE_H

typedef struct {
  const char * sender;
  const char * receiver;
  const char * tag;
  void * payload;
} Message;

#endif
```

common/genericvector_source.h

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "../simpo/sparse.h"

void VECTOR_INIT(VECTOR_NAME * vector) {
  vector->size = 0;
  vector->totalSize = 16;
  vector->array = (VECTOR_TYPE*)malloc(vector->totalSize * sizeof(VECTOR_TYPE)
      );
}

void VECTOR_RESIZE(VECTOR_NAME * vector, int size) {
  vector->size = size;
  vector->totalSize = size;
  vector->array = (VECTOR_TYPE*)realloc(vector->array, vector->totalSize *
      sizeof(VECTOR_TYPE));
}

void VECTOR_FREE(VECTOR_NAME * vector) {
  vector->size = 0;
  vector->totalSize = 0;
  free(vector->array);
  vector->array = NULL;
}
```

```
void VECTOR_APPEND(VECTOR_NAME * vector, VECTOR_TYPE element) {
  if (vector->size == vector->totalSize) {
    vector->totalSize *= 2;
    vector->array = (VECTOR_TYPE*)realloc(vector->array, vector->totalSize *
        sizeof(VECTOR_TYPE));
  }
  vector->array[vector->size++] = element;
}

VECTOR_TYPE VECTOR_GET(VECTOR_NAME vector, int index) {
  assert(index >= 0 && index < vector.size);
  return vector.array[index];
}

void VECTOR_SET(VECTOR_NAME vector, int index, VECTOR_TYPE value) {
  assert(index >= 0 && index < vector.size);
  vector.array[index] = value;
}

void VECTOR_REMOVE_AT(VECTOR_NAME * vector, int index) {
  int i;
  assert(index >= 0 && index < vector->size);
  --vector->size;
  for (i = index; i < vector->size; ++i) {
    vector->array[i] = vector->array[i + 1];
  }
}

// destination is assumed not to point to anything that must be freed
void VECTOR_COPY(VECTOR_NAME source, VECTOR_NAME * destination) {
  destination->size = source.size;
  destination->totalSize = source.size;
  allocateAndCopyArray(source.array, (void**)&destination->array, source.size,
      sizeof(VECTOR_TYPE));
}
```

simpo/2phase.c

```
static int useAsynplex = 1;
/*****************************************************************************

 Implementation of the
 2 phase (Dual then Primal) Simplex Method
 R. Vanderbei, 3 October 1994
```

```
 Solves problems in the form:

 T
 max c x

 A x  = b
 x >= 0

 A is an m by N matrix (it is convenient to reserve n for
 the difference N-m).  Artificial variables have been
 added (hence N > m).  One may assume that the last
 m columns of A are invertible and hence can be used as
 a starting basis.

 ******************************************************************************
    */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include "print.h"
#include "timer.h"
#include "communication.h"
#include "invertprocessor.h"
#include "iterationprocess.h"
#include "basischangemanager.h"
#include "columnselectionmanager.h"

#ifdef QuadPrec
#include "Quad.h"
#define TYPE Quad
#else
#define high(x) (x)
#endif

#include "../common/linalg.h"
#include "lu.h"
#include "../common/myalloc.h"
#include "../common/macros.h"

#define EPS_MATRIX 1.0e-8
#define EPS_MU_COEFFICIENTS 1.0e-12
#define EPS_MU_OPTIMAL 1.0e-12 //3.0e-4
#define MAX_ITER 1000000

#define EPS  1.0e-8
#define EPS1 1.0e-8
```

```
#define EPS2 1.0e-12
#define EPS3 1.0e-12

#define numIterationProcesses 2

//TODO: Can we find a drand48() implementation for windows?
#ifdef WIN32
#define drand48() (((double)rand())/((double)RAND_MAX))
#endif

TYPE recalculateObjectiveValues(TYPE *c, TYPE *x_B, int *basics, int m);

void Nt_times_y(SparseMatrix AT, int *basicflag, SparseVector y,
    SparseVector * yN);

int ratio_test(TYPE *dy, int *idy, int ndy, TYPE *y);

int pick_neg(int m, TYPE *x);

int solver(SparseMatrix A, TYPE *b, /* right-hand side */
  TYPE *c, /* objective coefficients */
  TYPE f, /* objective function shift */
  TYPE *x, /* primal solution (output) */
  TYPE *y, /* dual solution (output) */
  TYPE *w, /* primal slacks (output) */
  TYPE *z /* dual slacks (output) */
) {
  TYPE *x_B; /* primal basics */
  TYPE *y_N; /* dual nonbasics */

  SparseVector dyN_dualStepDir;
  SparseVector dxB_primalStepDir;
  SparseVector vector;

  int *basics; /* list of basic variable indices */
  int *nonbasics; /* list of non-basic variable indices */
  int *basicflag; /* neg if nonbasic, pos if basic */

  int col_in; /* entering column; index in 'nonbasics' */
  int col_out; /* leaving column; index in 'basics' */

  int iter = 0; /* number of iterations */
  int i, j, k, m, n, N, v = 0;

  TYPE s, t;
  float primal_obj;
```

```
int from_scratch;

SparseMatrix AT;

pthread_t invertProcessor;
pthread_t basisChangeManager;
pthread_t columnSelectionManager;
IterationProcess iterationProcesses[numIterationProcesses];

unsigned long long startTime = rdtsc();

/*****************************************************************
 * For convenience, we put...
 *****************************************************************/
m = A.rows;
N = A.cols;
n = N - m;

/*****************************************************************
 * Add slack variables.  We assume the calling routine allocated
 * enough memory.
 *****************************************************************/

i = 0;
k = A.colPos[n];
for (j = n; j < N; j++) {
  c[j] = 0.0;
  A.values[k] = 1.0;
  A.rowIndices[k] = i;
  i++;
  k++;
  A.colPos[j + 1] = k;
}
A.numNonzeroes = k;

/*****************************************************************
 * Read in the Data and initialize the common memory sites.
 *****************************************************************/

MALLOC( x_B, m, TYPE );
MALLOC( dxB_primalStepDir.values, m, TYPE );
MALLOC( y_N, n, TYPE );
MALLOC( dyN_dualStepDir.values, n, TYPE );
MALLOC( vector.values, m, TYPE );
MALLOC( nonbasics, n, int );
MALLOC( basics, m, int );
MALLOC( basicflag, N, int );
```

```
MALLOC( dxB_primalStepDir.rowIndices, m, int );
MALLOC( dyN_dualStepDir.rowIndices, n, int );
MALLOC( vector.rowIndices, m, int );
MALLOC( AT.values, A.numNonzeroes, TYPE );
MALLOC( AT.rowIndices, A.numNonzeroes, int );
MALLOC( AT.colPos, m+1, int );
vector.rows = m;
dxB_primalStepDir.rows = m;
dyN_dualStepDir.rows = n;



/*************************************************************
 *  Threading initialization.                                *
 *************************************************************/
if (useAsynplex) {
  initPrintFromProcess();
  initCommunication();
  setupInvertProcessor(A, numIterationProcesses);
  setupColumnSelectionManager(m, n);
  pthread_create(&invertProcessor, NULL, runInvertProcessor, NULL);
  pthread_create(&basisChangeManager, NULL, runBasisChangeManager, NULL);
  pthread_create(&columnSelectionManager, NULL, runColumnSelectionManager,
     NULL);
  printf("Using %d iteration processes\n", numIterationProcesses);
  for (i = 0; i < numIterationProcesses; ++i) {
    setupIterationProcess(&iterationProcesses[i], i, numIterationProcesses,
       A, b, c, f);
    pthread_create(&iterationProcesses[i].thread, NULL, runIterationProcess,
        &iterationProcesses[i]);
  }
  goto waitForAsynplex;
}



/*************************************************************
 *  Initialization.                                          *
 *************************************************************/

transposeSparseMatrix(A, &AT);

for (j = 0; j < n; j++) {
  nonbasics[j] = j;
  basicflag[j] = -j - 1;
  y_N[j] = MAX(c[j],1.0); /* to force dual feasibility */
  y_N[j] += drand48(); /* to ensure nondegeneracy   */
```

```
}

for (i = 0; i < m; i++) {
  basics[i] = n + i;
  basicflag[n + i] = i;
  x_B[i] = b[i];
}

lufac(m, A.colPos, A.rowIndices, A.values, basics, 0);

dbsolve(m, x_B); /* could be done explicitly in terms of bounds/ranges */

/*************************************************************
 *  Begin Phase I (i.e., dual simplex method)
 *************************************************************/

printf("m = %d,n = %d,nz = %d\n", m, N, A.numNonzeroes);
printf(
    "
       ------------------------------------------------------------------------\
       n"
      "          |   Primal      |        |
         arithmetic  \n"
      "  Iter    |  Obj Value    |   mu   |   nonz(L)     nonz(U)
         operations  \n"
      "- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          - - -\n");
fflush(stdout);

/*************************************************************
 *  Main loop                                                *
 *************************************************************/

for (iter = 0; iter < MAX_ITER; iter++) {

  /*************************************************************
   * STEP 6: Print statistics and factor/refactor              *
   *************************************************************/

  primal_obj = recalculateObjectiveValues(c, x_B, basics, m) + f;
  if (iter % 100 == 0) {
  printf("%8d   %14.7e     NA    \n", iter, high(primal_obj));
  fflush(stdout);
  }

  /*************************************************************
   * STEP 1: Pick most negative basic primal                   *
```

```
 ************************************************************/

col_out = pick_neg(m, x_B);
if (col_out == -1)
  break; /* ready for Phase II */

/************************************************************
 *                          -1  T                           *
 * STEP 2: Compute dy  = -(B  N) e                          *
 *                   N           i          *
 *         where i = col_out                                *
 ************************************************************/

vector.values[0] = -1.0;
vector.rowIndices[0] = col_out;
vector.numNonzeroes = 1;

solveTransposeUsingLU(&vector);

Nt_times_y(AT, basicflag, vector, &dyN_dualStepDir);

/************************************************************
 * STEP 3: Ratio test to find entering column               *
 ************************************************************/
col_in = ratio_test(dyN_dualStepDir.values, dyN_dualStepDir.rowIndices,
    dyN_dualStepDir.numNonzeroes, y_N);

if (col_in == -1) {
  return 2; /* INFEASIBLE */
}

/************************************************************
 *                         -1                               *
 * STEP 4: Compute dx  = B  N e                             *
 *                   B         j                            *
 *                                                          *
 ************************************************************/

j = nonbasics[col_in];
for (i = 0, k = A.colPos[j]; k < A.colPos[j + 1]; i++, k++) {
  dxB_primalStepDir.values[i] = A.values[k];
  dxB_primalStepDir.rowIndices[i] = A.rowIndices[k];
}
dxB_primalStepDir.numNonzeroes = i;

solveUsingLU(&dxB_primalStepDir);
```

```
/*************************************************************
 *                                                           *
 * STEP 5: Put       t = x /dx                               *
 *                        i   i                              *
 *                   s = y /dy                               *
 *                        j   j                              *
 *************************************************************/

/* this is inefficient - it should be fixed */
for (k = 0; k < dxB_primalStepDir.numNonzeroes; k++)
  if (dxB_primalStepDir.rowIndices[k] == col_out)
    break;

t = x_B[col_out] / dxB_primalStepDir.values[k];

/* this is inefficient - it should be fixed */
for (k = 0; k < dyN_dualStepDir.numNonzeroes; k++)
  if (dyN_dualStepDir.rowIndices[k] == col_in)
    break;

s = y_N[col_in] / dyN_dualStepDir.values[k];

/*************************************************************
 * STEP 7: Set y  = y  - s dy                                *
 *             N    N       N                                *
 *                                                           *
 *            y  = s                                         *
 *             i                                             *
 *                                                           *
 *            x  = x  - t dx                                 *
 *             B    B       B                                *
 *                                                           *
 *            x  = t                                         *
 *             j                                             *
 *************************************************************/

for (k = 0; k < dyN_dualStepDir.numNonzeroes; k++) {
  j = dyN_dualStepDir.rowIndices[k];
  y_N[j] -= s * dyN_dualStepDir.values[k];
}

y_N[col_in] = s;

for (k = 0; k < dxB_primalStepDir.numNonzeroes; k++) {
  i = dxB_primalStepDir.rowIndices[k];
  x_B[i] -= t * dxB_primalStepDir.values[k];
}
```

```
  x_B[col_out] = t;
  /*************************************************************
   * STEP 8: Update basis                                      *
   *************************************************************/

  i = basics[col_out];
  j = nonbasics[col_in];
  basics[col_out] = j;
  nonbasics[col_in] = i;
  basicflag[i] = -col_in - 1;
  basicflag[j] = col_out;

  from_scratch = refactor(m, A.colPos, A.rowIndices, A.values, basics,
      col_out, v);
} /* End of Phase I */

printf("%8d   %14.7e     NA    \n", iter, high(primal_obj));
printf("End of Phase I \n");

/*************************************************************
 *   Restore objective function by setting                   *
 *                 -1  T                                     *
 *         y  = (B  N) c  - c                                *
 *          N           B    N                               *
 *************************************************************/

vector.numNonzeroes = 0;
for (i = 0; i < m; i++) {
  if (ABS(c[basics[i]]) > EPS) {
    vector.values[vector.numNonzeroes] = c[basics[i]];
    vector.rowIndices[vector.numNonzeroes] = i;
    vector.numNonzeroes++;
  }
}

solveTransposeUsingLU(&vector);

Nt_times_y(AT, basicflag, vector, &dyN_dualStepDir);

for (j = 0; j < n; j++)
  y_N[j] = -c[nonbasics[j]];
for (k = 0; k < dyN_dualStepDir.numNonzeroes; k++) {
  j = dyN_dualStepDir.rowIndices[k];
  y_N[j] += dyN_dualStepDir.values[k];
}
```

```
/*************************************************************
 *   Begin Phase II (I.e., primal simplex method)            *
 *************************************************************/

for (; iter < MAX_ITER; iter++) {

  /*********************************************************
   * STEP 6: Print statistics and factor/refactor          *
   *********************************************************/

  primal_obj = recalculateObjectiveValues(c, x_B, basics, m) + f;
  if (iter % 100 == 0) {
  printf("%8d   %14.7e     NA    \n", iter, high(primal_obj));
  fflush(stdout);
  }

  /*********************************************************
   * STEP 1: Pick most negative nonbasic dual              *
   *********************************************************/

  col_in = pick_neg(n, y_N);
  if (col_in == -1)
    break; /* optimal */

  /*********************************************************
   *                        -1                             *
   * STEP 2: Compute dx  = B  N e                          *
   *                   B         j                         *
   *         where j = col_in                              *
   *********************************************************/

  j = nonbasics[col_in];
  for (i = 0, k = A.colPos[j]; k < A.colPos[j + 1]; i++, k++) {
    dxB_primalStepDir.values[i] = A.values[k];
    dxB_primalStepDir.rowIndices[i] = A.rowIndices[k];
  }
  dxB_primalStepDir.numNonzeroes = i;

  solveUsingLU(&dxB_primalStepDir);


  /*********************************************************
   * STEP 3: Ratio test to find leaving column             *
   *********************************************************/

  col_out = ratio_test(dxB_primalStepDir.values,
      dxB_primalStepDir.rowIndices, dxB_primalStepDir.numNonzeroes,
```

```
    x_B);

if (col_out == -1)
  return 1; /* UNBOUNDED */


/*************************************************************
 *                            -1  T                          *
 * STEP 4: Compute dy  = -(B  N) e                           *
 *                   N           i          *                *
 *         where i = col_out                                 *
 *************************************************************/

vector.values[0] = -1.0;
vector.rowIndices[0] = col_out;
vector.numNonzeroes = 1;

solveTransposeUsingLU(&vector);

Nt_times_y(AT, basicflag, vector, &dyN_dualStepDir);

/*************************************************************
 * STEP 5: Put       t = x /dx                               *
 *                        i   i                              *
 *                   s = y /dy                               *
 *                        j   j                              *
 *************************************************************/

/* this is inefficient - it should be fixed */
for (k = 0; k < dxB_primalStepDir.numNonzeroes; k++)
  if (dxB_primalStepDir.rowIndices[k] == col_out)
    break;

t = x_B[col_out] / dxB_primalStepDir.values[k];

/* this is inefficient - it should be fixed */
for (k = 0; k < dyN_dualStepDir.numNonzeroes; k++)
  if (dyN_dualStepDir.rowIndices[k] == col_in)
    break;

s = y_N[col_in] / dyN_dualStepDir.values[k];
/*************************************************************
 * STEP 7: Set y  = y  - s dy                                *
 *              N    N       N                               *
 *                                                           *
 *             y  = s                                        *
 *              i                                            *
```

```
   *                                                           *
   *             x  = x  - t dx                                *
   *              B    B       B                               *
   *                                                           *
   *             x  = t                                        *
   *              j                                            *
   ************************************************************/

  for (k = 0; k < dyN_dualStepDir.numNonzeroes; k++) {
    j = dyN_dualStepDir.rowIndices[k];
    y_N[j] -= s * dyN_dualStepDir.values[k];
  }

  y_N[col_in] = s;

  for (k = 0; k < dxB_primalStepDir.numNonzeroes; k++) {
    i = dxB_primalStepDir.rowIndices[k];
    x_B[i] -= t * dxB_primalStepDir.values[k];
  }

  x_B[col_out] = t;


  /************************************************************
   * STEP 8: Update basis                                     *
   ************************************************************/

  i = basics[col_out];
  j = nonbasics[col_in];
  basics[col_out] = j;
  nonbasics[col_in] = i;
  basicflag[i] = -col_in - 1;
  basicflag[j] = col_out;

  from_scratch = refactor(m, A.colPos, A.rowIndices, A.values, basics,
      col_out, v);
} /* End of Phase II */

primal_obj = recalculateObjectiveValues(c, x_B, basics, m) + f;
printf("%8d   %14.7e     NA    \n", iter, high(primal_obj));
printf("End of Phase II \n");

/************************************************************
 *  Transcribe solution to x vector and dual solution to y  *
 ************************************************************/

for (j = 0; j < N; j++)
```

```
    x[j] = 0.0;
  for (i = 0; i < m; i++)
    x[basics[i]] = x_B[i];
  //printTypeArray(x, N);

  for (j = 0; j < N; j++)
    y[j] = 0.0;
  for (i = 0; i < n; i++)
    y[nonbasics[i]] = y_N[i];

  /*************************************************************
   *  Split out slack variables and shift dual variables.
   *************************************************************/

  for (j = 0; j < n; j++)
    z[j] = y[j];
  for (i = 0; i < m; i++) {
    y[i] = y[n + i];
    w[i] = x[n + i];
  }

  /*************************************************************
   *  Free work space                                          *
   *************************************************************/

  FREE( vector.values );
  FREE( vector.rowIndices );
  FREE( x_B );
  FREE( y_N );
  FREE( dxB_primalStepDir.values );
  FREE(dxB_primalStepDir.rowIndices );
  FREE( dyN_dualStepDir.values );
  FREE(dyN_dualStepDir.rowIndices );
  FREE( nonbasics );
  FREE( basics );

  if (useAsynplex) {
waitForAsynplex:
    for (i = numIterationProcesses - 1; i >= 0; --i) {
      pthread_join(iterationProcesses[i].thread, NULL);
    }
    pthread_join(invertProcessor, NULL);
    pthread_join(columnSelectionManager, NULL);
    pthread_join(basisChangeManager, NULL);
  }
  printf("Time spent (minus input handling): %llu\n", rdtsc() - startTime);
```

```
  return 0;

} /* End of solver */

void Display_Solution(int m, int *basics, TYPE *X) {
  int i;

  printf("SOLUTION:\n\n");
  for (i = 0; i < m; i++)
    printf("  X[%d] = %lf\n", basics[i], high(X[i]));
}

void Nt_times_y(SparseMatrix AT, int *basicflag, SparseVector y,
    SparseVector * yN) {
  int i, j, jj, k, kk;

  static TYPE *a = NULL;
  static int *tag = NULL;
  static int *link = NULL;
  static int currtag = 1;

  if (a == NULL)
    MALLOC( a, AT.rows, TYPE);
  if (tag == NULL)
    CALLOC( tag, AT.rows, int);
  if (link == NULL) {
    CALLOC(link, AT.rows+2, int);
    link++;
  }

  jj = -1;
  for (k = 0; k < y.numNonzeroes; k++) {
    i = y.rowIndices[k];
    for (kk = AT.colPos[i]; kk < AT.colPos[i + 1]; kk++) {
      j = AT.rowIndices[kk];
      if (basicflag[j] < 0) {
        if (tag[j] != currtag) {
          a[j] = 0.0;
          tag[j] = currtag;
          link[jj] = j;
          jj = j;
        }
        a[j] += y.values[k] * AT.values[kk];
      }
    }
  }
  link[jj] = AT.rows;
```

```
  currtag++;

  k = 0;
  for (jj = link[-1]; jj < AT.rows; jj = link[jj]) {
    if (ABS(a[jj]) > EPS_MATRIX) {
      yN->values[k] = a[jj];
      yN->rowIndices[k] = -basicflag[jj] - 1;
      k++;
    }
  }
  yN->numNonzeroes = k;
}

int ratio_test(TYPE *dy, int *idy, int ndy, TYPE *y) {
  int j, jj = -1, k;
  TYPE min = HUGE_VAL;

  for (k = 0; k < ndy; k++) {
    if (dy[k] > EPS1) {
      j = idy[k];
      if (y[j] / dy[k] < min) {
        min = y[j] / dy[k];
        jj = j;
      }
    }
  }

  return jj;
}

TYPE recalculateObjectiveValues(TYPE *c, TYPE *x_B, int *basics, int m) {
  int i;
  TYPE prod = 0.0;

  for (i = 0; i < m; i++) {
    prod += c[basics[i]] * x_B[i];
  }

  return prod;
}

int pick_neg(int m, TYPE *x) {
  int i;
  int col = -1;
  TYPE min = -EPS2;

  for (i = 0; i < m; i++) {
```

```
    if (x[i] < min) {
      min = x[i];
      col = i;
    }
  }
  return col;
}
```

simpo/basischangemanager.h and simpo/basischangemanager.c

```
#ifndef BASISCHANGEMANAGER_H
#define BASISCHANGEMANAGER_H

void * runBasisChangeManager(void * parameters);

#endif
```

```
#include "communication.h"
#include "invertprocessor.h"
#include "print.h"
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>

#define NAME "R"

static int currentBasis = 0;

static void send(const char * receiver, const char * tag, void * payload) {
  Message msg = {NAME, receiver, tag, payload};
  sendMessage(msg);
}

static Message receiveFromAnyone() {
  return receiveMessageFromAnyone(NAME);
}

extern int enablePrint; // Defined in iterationprocess.c

#define print(formatString, ...) if (enablePrint) { printFromProcess(NAME,
    formatString , ## __VA_ARGS__); }

void * runBasisChangeManager(void * parameters) {
  Message message;
  const char * result;
  int * payload;
```

```
  int incomingBasis;
  print("Basis change manager starting");
  while (!isProgramFinished()) {
    while (!(message = receiveFromAnyone()).sender) {
      if (isProgramFinished()) {
        print("Basis change manager exiting");
        pthread_exit(NULL);
        return NULL; // To prevent a compiler warning
      }
      sched_yield();
    }
    assert(!strcmp(message.tag, "I5->R1"));
    incomingBasis = *(int*)message.payload;
    free(message.payload);
    payload = (int*)malloc(sizeof(int));
    if (incomingBasis == currentBasis) {
      *payload = 1;
      send(message.sender, "R2->I6", payload);
      ++currentBasis;
      result = "accepted";
    }
    else {
      *payload = 0;
      send(message.sender, "R3->I6", payload);
      result = "refused";
    }
    print("%s offers chuzr on basis %d... %s", message.sender, incomingBasis,
        result);
  }
  print("Basis change manager exiting");
  pthread_exit(NULL);
  return NULL; // To prevent a compiler warning
}
```

simpo/columnselectionmanager.h and simpo/columnselectionmanager.c

```
#ifndef COLUMNSELECTIONMANAGER_H
#define COLUMNSELECTIONMANAGER_H

typedef enum {
  OTHER_SIDE,
  UNSELECTED,
  SELECTED,
  REJECTED,
  UNDEFINED
} VariableStatus;
```

```
typedef struct {
  VariableStatus status;
  int basisWhereStatusChanged;
} VariableInfo;

typedef struct {
  int unattractiveVariable;
  int basis;
  const char * processName;
  const char * tagToSend;
} QueuedVariableRequest;

void setupColumnSelectionManager(int m, int n);
void * runColumnSelectionManager(void * parameters);

#endif
```

```
#include "communication.h"
#include "print.h"
#include "../common/genericvectors.h"
#include "payloadtypes.h"
#include "invertprocessor.h"
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <vector>
using namespace std;

#define NAME "C"

static char * VARIABLE_STATUS_NAMES[] = {"OTHER_SIDE", "UNSELECTED", "SELECTED
    ", "REJECTED", "UNDEFINED"};

static VariableInfo * variables;
static vector<int> attractive;
static QueuedVariableRequestVector queuedVariableRequests;
static int m;
static int n;
static int kc;

static void send(const char * receiver, const char * tag, void * payload) {
  Message msg = {NAME, receiver, tag, payload};
  sendMessage(msg);
}
```

```
static Message receiveFromAnyone() {
  return receiveMessageFromAnyone(NAME);
}

extern int enablePrint; // Defined in iterationprocess.c

#define print(formatString, ...) if (enablePrint) { printFromProcess(NAME,
    formatString , ## __VA_ARGS__); }

void setupColumnSelectionManager(int _m, int _n) {
  int i;
  m = _m;
  n = _n;
  kc = 0;
  initQVRV(&queuedVariableRequests);
  variables = (VariableInfo*)malloc(sizeof(VariableInfo) * (m + n));
  //TODO: This must be switched between the phases
  for (i = 0; i < n; ++i) { // nonbasic
    variables[i].status = OTHER_SIDE;
    variables[i].basisWhereStatusChanged = -1;
  }
  for (i = 0; i < m; ++i) { // basic
    variables[n + i].status = UNSELECTED;
    variables[n + i].basisWhereStatusChanged = -1;
  }
}

bool installAttractiveCandidates(Message message) {
  int i;
  AttractiveCandidatesMessage msg = *(AttractiveCandidatesMessage*)message.
      payload;
  delete (AttractiveCandidatesMessage*)message.payload;
  assert(msg.attractiveVariables.size() >= 1);
  print("received %d candidates from %s, which is at basis %d; currently at
      basis %d", msg.attractiveVariables.size(), message.sender, msg.
      basisNumber, kc);
  if (msg.basisNumber > kc) {
    int installCount = 0;
    attractive.clear();
    for (i = 0; i < msg.attractiveVariables.size(); ++i) {
      int cand = msg.attractiveVariables.at(i);
      if (!((variables[cand].status == SELECTED || variables[cand].status ==
          REJECTED) && variables[cand].basisWhereStatusChanged >= msg.
          basisNumber)) {
        ++installCount;
        attractive.push_back(cand);
        variables[cand].status = UNSELECTED;
```

```
        variables[cand].basisWhereStatusChanged = msg.basisNumber;
      }
      else {
        print("rejected x%d because it got %s at basis %d", cand,
            VARIABLE_STATUS_NAMES[variables[cand].status], variables[cand].
            basisWhereStatusChanged);
      }
    }
    print("Installed %d candidates", installCount);
    kc = msg.basisNumber;
    return true;
  }
  else {
    print("Rejected the candidates");
    return false;
  }
}

void * runColumnSelectionManager(void * parameters) {
  Message message;
  int var;
  int * varPayload;
  QueuedVariableRequest req;
  print("Column selection manager starting");

  while (!isProgramFinished()) {
    while (queuedVariableRequests.size > 0 && attractive.size() > 0) {
      if (isProgramFinished()) {
        print("Column selection manager exiting");
        pthread_exit(NULL);
        return NULL; // To prevent a compiler warning
      }
      do {
        var = attractive.at(0);
        attractive.erase(attractive.begin());
      } while (variables[var].status != UNSELECTED && attractive.size() > 0);
      if (variables[var].status != UNSELECTED && attractive.size() == 0)
        break;
      variables[var].status = SELECTED;
      variables[var].basisWhereStatusChanged = kc;
      req = getQVRV(queuedVariableRequests, 0);
      removeAtQVRV(&queuedVariableRequests, 0);
      print("x%d gets SELECTED (C5) at basis %d", var, kc);
      varPayload = (int*)malloc(sizeof(int));
      *varPayload = var;
      send(req.processName, req.tagToSend, varPayload);
    }
```

```
while (!(message = receiveFromAnyone()).sender) {
  if (isProgramFinished()) {
    print("Column selection manager exiting");
    pthread_exit(NULL);
    return NULL; // To prevent a compiler warning
  }
  sched_yield();
}
assert(message.sender[0] == 'I');
if (!strcmp(message.tag, "I8->C1")) {
  ColSelPivotPayload * payload = (ColSelPivotPayload*)message.payload;
  variables[payload->enteringVar].status = OTHER_SIDE;
  variables[payload->enteringVar].basisWhereStatusChanged = payload->
     basisNumber;
  variables[payload->leavingVar].status = UNSELECTED;
  variables[payload->leavingVar].basisWhereStatusChanged = payload->
     basisNumber;
  //print("x%d left basis %d (by %s) and got unselected", payload->
     leavingVar, payload->basisNumber, message.sender);
  free(payload);
}
else if (!strcmp(message.tag, "I9->C2")) {
  if (installAttractiveCandidates(message)) {
    //queuedVariableRequests.Enqueue(new QueuedVariableRequest() {
       processName = message.sender, tagToSend = "C3->I11" });
    var = attractive.at(0);
    attractive.erase(attractive.begin());
    assert(variables[var].status == UNSELECTED);
    variables[var].status = SELECTED;
    variables[var].basisWhereStatusChanged = kc;
    print("x%d gets SELECTED (C3) at basis %d", var, kc);
    varPayload = (int*)malloc(sizeof(int));
    *varPayload = var;
    send(message.sender, "C3->I11", varPayload);
  }
  else {
    req.processName = message.sender;
    req.tagToSend = "C3->I11";
    appendQVRV(&queuedVariableRequests, req);
  }
}
else if (!strcmp(message.tag, "I4->C4")) {
  QueuedVariableRequest req;
  CandidateIsUnattractivePayload * payload = (
     CandidateIsUnattractivePayload*)message.payload;
  if (payload->var != -1) {
    variables[payload->var].status = REJECTED;
```

```
        variables[payload->var].basisWhereStatusChanged = payload->basisNumber
            ;
        print("x%d gets REJECTED (C5) at basis %d", payload->var, payload->
            basisNumber);
      }
      free(payload);
      req.processName = message.sender;
      req.tagToSend = "C5->I11";
      appendQVRV(&queuedVariableRequests, req);
    }
    else {
      print("Unexpected tag '%s'", message.tag);
      assert(0);
    }
  }
  print("Column selection manager exiting");
  pthread_exit(NULL);
  return NULL; // To prevent a compiler warning
}
```

simpo/communication.h and simpo/communication.c

```
#ifndef COMMUNICATION_H
#define COMMUNICATION_H

#include "../common/message.h"

void initCommunication();
void sendMessage(Message message);
void sendAll(Message * messagesToBeSent, int numMessages);
Message receiveMessageFromAnyone(const char * receiver);
Message receiveMessage(const char * receiver, const char * sender);
Message receiveMessageWithTag(const char * receiver, const char * sender,
    const char * tag);
void printMessages();

#endif
```

```
#include "../common/message.h"
#include "../common/genericvectors.h"
#include "communication.h"
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include "print.h"
```

```
Message createMessage(const char * sender, const char * receiver, const char *
    tag, void * payload) {
  Message message;
  message.sender = sender;
  message.receiver = receiver;
  message.tag = tag;
  message.payload = payload;
  return message;
}

static MessageVector messages;
static pthread_mutex_t queueLock;
static int printSends = 1;
static int printReceives = 1;

void initCommunication() {
  pthread_mutex_init(&queueLock, NULL);
  initMV(&messages);
}

void sendMessage(Message message) {
  pthread_mutex_lock(&queueLock);
  appendMV(&messages, message);
  pthread_mutex_unlock(&queueLock);
  if (printSends) {
    printFromProcess("COMM", "%s sends to %s (tag: '%s', payload %p) (queue: %
        d)", message.sender, message.receiver, message.tag, message.payload,
        messages.size);
  }
}

void sendAll(Message * messagesToBeSent, int numMessages) {
  int i;
  pthread_mutex_lock(&queueLock);
  for (i = 0; i < numMessages; ++i) {
    appendMV(&messages, messagesToBeSent[i]);
    if (printSends) {
      printFromProcess("COMM", "%s sends to %s (tag: '%s', payload %p) (queue:
          %d)", messagesToBeSent[i].sender, messagesToBeSent[i].receiver,
          messagesToBeSent[i].tag, messagesToBeSent[i].payload, messages.size)
          ;
    }
  }
  pthread_mutex_unlock(&queueLock);
}

// Returns a message with sender == null if no message is available
```

```
Message receiveMessageFromAnyone(const char * receiver) {
  return receiveMessageWithTag(receiver, NULL, NULL);
}

// Returns a message with sender == null if no message is available
Message receiveMessage(const char * receiver, const char * sender) {
  return receiveMessageWithTag(receiver, sender, NULL);
}

Message receiveMessageWithTag(const char * receiver, const char * sender,
    const char * tag) {
  Message msg;
  int i;
  pthread_mutex_lock(&queueLock);
  for (i = 0; i < messages.size; ++i) {
    msg = getMV(messages, i);
    if (!strcmp(msg.receiver, receiver) && (!sender || !strcmp(msg.sender,
        sender) || (!strcmp(sender, "I*") && msg.sender[0] == 'I')) && (!tag
        || !strcmp(msg.tag, tag))) {
      removeAtMV(&messages, i);
      if (printReceives) {
        printFromProcess("COMM", "%s receives from %s (tag: '%s', payload: %p)
            (queue: %d)", msg.receiver, msg.sender, msg.tag, msg.payload,
            messages.size);
      }
      pthread_mutex_unlock(&queueLock);
      return msg;
    }
  }
  pthread_mutex_unlock(&queueLock);
  msg.sender = NULL;
  return msg;
}

void printMessages() {
  int i;
  pthread_mutex_lock(&queueLock);
  printFromProcess("COMM", "Remaining messages:");
  for (i = 0; i < messages.size; ++i) {
    printFromProcess("COMM", "%s %s %s", getMV(messages, i).sender, getMV(
        messages, i).receiver, getMV(messages, i).tag);
  }
  fflush(stdout);
  pthread_mutex_unlock(&queueLock);
}
```

simpo/invertprocessor.h and simpo/invertprocessor.c

```
#ifndef INVERTPROCESSOR_H
#define INVERTPROCESSOR_H

#include "sparse.h"

void setupInvertProcessor(SparseMatrix _A, int _numIterationProcesses);
void * runInvertProcessor(void * parameters);

int isProgramFinished();
int announceProgramFinished();

#endif
```

```
#include "communication.h"
#include "invertprocessor.h"
#include "iterationprocess.h" // For the FactoredInverse structure
#include "sparse.h"
#include "../common/genericvectors.h"
#include "util.h"
#include "payloadtypes.h"
#include "../common/myalloc.h"
#include "../common/macros.h"
#include "../common/heap.h"
#include "../common/linalg.h"
#include "print.h"
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <vector>
using namespace std;

static const char * NAME = "V";

static SparseMatrix A;
static int * basicVariables;
static int * nonbasicVariables;
static int m, n, N;
static int verbose = 0;
static SparseMatrix L = {0, 0, 0, NULL, NULL, NULL};
static SparseMatrix LT = {0, 0, 0, NULL, NULL, NULL};
static SparseMatrix U = {0, 0, 0, NULL, NULL, NULL};
static SparseMatrix UT = {0, 0, 0, NULL, NULL, NULL};

typedef struct {    /* nonzero entry */
  TYPE value;  /* value */
```

```
  int rowIndex;  /* row index */
} ValueAndRowIndex;

static SafeVector<int> colPerm, invColPerm, rowPerm, invRowPerm;
static SafeVector<TYPE> diagU;

static double cumtime = 0.0; //TODO: This one was used both by the functions
    that were moved to invertprocessor.c and to interationprocess.c

static int numIterationProcesses;
static char ** iterationProcessNames;

#define E_N   200
#define E_NZ  20000
#define MD    1
#define EPSNUM  1.0e-9

static int    rank;

static int receivedBasisNumber = 0;
static int lastReinversionBasis = 0;

static pthread_mutex_t finishLock;
static int _isProgramFinished = 0;

static void invertProcessor_lufac();
static void invertProcessor_broadcastInverse();

static void send(const char * receiver, const char * tag, void * payload) {
  Message msg = {NAME, receiver, tag, payload};
  sendMessage(msg);
}

static Message receiveFromAnyone() {
  return receiveMessageFromAnyone(NAME);
}

extern int enablePrint; // Defined in iterationprocess.c

//http://ou800doc.sco.com/cgi-bin/info2html?(gcc.info)Macro%2520Varargs&lang=
    en
#define print(formatString, ...) if (enablePrint) { printFromProcess(NAME,
    formatString , ## __VA_ARGS__); }

int isProgramFinished() {
  int result;
  pthread_mutex_lock(&finishLock);
```

```
  result = _isProgramFinished;
  pthread_mutex_unlock(&finishLock);
  return result;
}

int announceProgramFinished() {
  int oldValue;
  pthread_mutex_lock(&finishLock);
  oldValue = _isProgramFinished;
  _isProgramFinished = 1;
  pthread_mutex_unlock(&finishLock);
  return oldValue;
}

void setupInvertProcessor(SparseMatrix _A, int _numIterationProcesses) {
  int i;
  numIterationProcesses = _numIterationProcesses;
  iterationProcessNames = (char**)malloc(numIterationProcesses * sizeof(char
     *));
  for (i = 0; i < numIterationProcesses; ++i) {
    iterationProcessNames[i] = (char*)malloc(16);
    sprintf(iterationProcessNames[i], "I%d", i);
  }
  copySparseMatrix(_A, &A);
  m = A.rows;
  N = A.cols;
  n = N - m;
  basicVariables = (int*)malloc(m * sizeof(int));
  for (i = 0; i < m; ++i) {
    basicVariables[i] = n + i;
  }
  nonbasicVariables = (int*)malloc(n * sizeof(int));
  for (i = 0; i < n; ++i) {
    nonbasicVariables[i] = i;
  }
  pthread_mutex_init(&finishLock, NULL);
}

void * runInvertProcessor(void * parameters) {
  Message message;
  InvProcPivotPayload * pivot;
  int temp;
  print("Starting");
  seedRandomGeneratorForThisThread();

  while (!isProgramFinished()) {
    while ((message = receiveFromAnyone()).sender || receivedBasisNumber ==
```

```
       lastReinversionBasis) {
      if (isProgramFinished()) {
        print("Terminating");
        pthread_exit(NULL);
        return NULL;
      }
      if (!message.sender)
        continue;
      assert(!strcmp(message.tag, "I8->V1"));
      pivot = (InvProcPivotPayload*)message.payload;
      //print("Received pivot: l/e: %d/%d", pivot->leavingRow, pivot->
          enteringCol);
      temp = basicVariables[pivot->leavingRow];
      basicVariables[pivot->leavingRow] = nonbasicVariables[pivot->enteringCol
          ];
      nonbasicVariables[pivot->enteringCol] = temp;
      ++receivedBasisNumber;
      free(pivot);
    }
    if (isProgramFinished()) {
      print("Terminating");
      pthread_exit(NULL);
      return NULL;
    }

    print("Starting inverse calculation for basis %d", receivedBasisNumber);
    invertProcessor_lufac();
    lastReinversionBasis = receivedBasisNumber;
    print("Inverse for basis %d done; broadcasting", lastReinversionBasis);
    invertProcessor_broadcastInverse();
    /*
    printRawSparseMatrix(L);
    printRawSparseMatrix(U);
    for (i = 0; i < m; ++i)
      printf("%f ", diagU[i]);
    printf("\n");
    */
  }

  print("Terminating");
  pthread_exit(NULL);
  return NULL; // To prevent a compiler warning
}

static void invertProcessor_broadcastInverse() {
  int i;
  FactoredInverse * factoredInverse;
```

```
  Message * messages = new Message[numIterationProcesses];
  for (i = 0; i < numIterationProcesses; ++i) {
    //TODO: Do we want each process to get its own copy? If not, how do we
       delete the common copy after all processes are done with it?
    //TODO: Each ItProc must free BOTH the matrices/vectors AND the
       FactoredInverse object
    factoredInverse = new FactoredInverse;
    copySparseMatrix(L, &factoredInverse->L);
    copySparseMatrix(LT, &factoredInverse->LT);
    copySparseMatrix(U, &factoredInverse->U);
    copySparseMatrix(UT, &factoredInverse->UT);
    factoredInverse->colPerm = colPerm;
    factoredInverse->invColPerm = invColPerm;
    factoredInverse->rowPerm = rowPerm;
    factoredInverse->invRowPerm = invRowPerm;
    factoredInverse->diagU = diagU;
    factoredInverse->rank = rank;
    factoredInverse->basisNumber = lastReinversionBasis;
    messages[i].sender = NAME;
    messages[i].receiver = iterationProcessNames[i];
    messages[i].tag = "V2->I1";
    messages[i].payload = factoredInverse;
  }
  sendAll(messages, numIterationProcesses);
  delete [] messages;
}

static void invertProcessor_lufac() {
  int kk, kkk, tag, rowdeg, coldeg, row, col, row2, col2;

  int i, j, k, cnt, lnz, unz, lnzbnd, unzbnd, okey, deg, heapnum, cur,
      method = MD;

  int *degB = NULL, *degBt = NULL, *hkey = NULL, *heap = NULL, *iheap = NULL,
      *iwork = NULL, *iwork2 = NULL;

  ValueAndRowIndex tempB, **B = NULL, **Bt = NULL;

  TYPE narth;

  double starttime, endtime;

  starttime = (double) clock();

  /*-----------------------------------------------------+
   | allocate space for perm and iperm.                  */
  colPerm.resize(m);
```

```
invColPerm.resize(m);
rowPerm.resize(m);
invRowPerm.resize(m);
/*-----------------------------------------------------------+
 | allocate space for work arrays.                           */

MALLOC( degB, m, int );
MALLOC( degBt, m, int );
MALLOC( hkey, m, int );
MALLOC( heap, m, int );
MALLOC( iheap, m, int );
MALLOC( iwork, m, int );
MALLOC( iwork2, m, int );

heap--; /* so that indexing starts from 1 */

/*-----------------------------------------------------------+
 | calculate degrees in B and Bt                             */

for (i = 0; i < m; i++) {
  degBt[i] = 0;
}
for (i = 0; i < m; i++) {
  degB[i] = A.colPos[basicVariables[i] + 1] - A.colPos[basicVariables[i]];
  for (k = A.colPos[basicVariables[i]]; k < A.colPos[basicVariables[i] + 1];
       k++) {
    degBt[A.rowIndices[k]]++;
  }
}

/*-----------------------------------------------------------+
 | calculate initial estimate of number of nonzeros in       |
 | L and UT.values                                           */

lnzbnd = 0;
for (i = 0; i < m; i++)
  lnzbnd += degB[i];
lnzbnd = lnzbnd / 2;
unzbnd = lnzbnd;

/*-----------------------------------------------------------+
 | allocate enough space to store L and UT.values            |
 | (without any fillin)                                      */

if (L.colPos == NULL) {
  MALLOC( L.colPos, m+1, int );
} else {
```

```
  REALLOC( L.colPos, m+1, int );
}
if (L.rowIndices == NULL) {
  MALLOC( L.rowIndices, lnzbnd, int );
} else {
  REALLOC( L.rowIndices, lnzbnd, int );
}
if (L.values == NULL) {
  MALLOC( L.values, lnzbnd, TYPE );
} else {
  REALLOC( L.values, lnzbnd, TYPE );
}
if (UT.colPos == NULL) {
  MALLOC( UT.colPos, m+1, int );
} else {
  REALLOC( UT.colPos, m+1, int );
}
if (UT.rowIndices == NULL) {
  MALLOC( UT.rowIndices, unzbnd, int );
} else {
  REALLOC( UT.rowIndices, unzbnd, int );
}
if (UT.values == NULL) {
  MALLOC( UT.values, unzbnd, TYPE );
} else {
  REALLOC( UT.values, unzbnd, TYPE );
}
diagU.resize(m);

MALLOC( B, m, ValueAndRowIndex * );
MALLOC( Bt, m, ValueAndRowIndex * );
for (i = 0; i < m; i++) {
  B[i] = NULL;
  Bt[i] = NULL;
  MALLOC( B[i], degB[i], ValueAndRowIndex );
  MALLOC( Bt[i], degBt[i], ValueAndRowIndex );
}

/*-----------------------------------------------------------+
 | initialize B and Bt                                       */

for (i = 0; i < m; i++) {
  iwork[i] = 0;
}
for (j = 0; j < m; j++) {
  kkk = 0;
  for (k = A.colPos[basicVariables[j]]; k < A.colPos[basicVariables[j] + 1];
```

```
      k++) {
    row = A.rowIndices[k];
    kk = iwork[row];
    B[j][kkk].rowIndex = row;
    B[j][kkk].value = A.values[k];
    Bt[row][kk].rowIndex = j;
    Bt[row][kk].value = A.values[k];
    iwork[row]++;
    kkk++;
  }
}

/*-----------------------------------------------------------+
 | miscellaneous initializations.                            */

for (i = 0; i < m; i++) {
  invColPerm.at(i) = -1;
  invRowPerm.at(i) = -1;
  iwork[i] = 0;
  iwork2[i] = -1;
}

rank = m;
tag = 0;
lnz = 0;
unz = 0;
L.colPos[0] = 0;
UT.colPos[0] = 0;

/*-----------------------------------------------------------+
 | hkey encodes the tie-breaking rule - currently the rule   |
 | is somewhat random.  to make it first occuring minimum,   |
 | change the formula to:                                    |
 |       hkey[node] = degree[node]*m + node;                 |
 | warning: with this definition of hkey, there is the       |
 | possibility of integer overflow on moderately large       |
 | problems.                                                 |
 |                                                           */

for (j = 0; j < m; j++) {
  if (method == MD)
    hkey[j] = degB[j];
  else
    hkey[j] = j;

  if (hkey[j] == 0)
    hkey[j] = m + 1;
```

```
}

/*-----------------------------------------------------------+
 | set up heap structure for quickly accessing minimum.    */

heapnum = m;
for (j = m - 1; j >= 0; j--) {
  cur = j + 1;
  iheap[j] = cur;
  heap[cur] = j;
  hfall(heapnum, hkey, iheap, heap, cur);
}

/*-----------------------------------------------------------+
 | the min degree ordering loop                              */

for (i = 0; i < m; i++) {

  /*-----------------------------------------------+
   |  select column with min column degree        */

  again: col = heap[1];
  coldeg = degB[col];

  if (coldeg == 0) {
    printf("singular matrix. rank deficiency = %d\n", m - i);
    rank = i;
    goto end;
  }

  /*-----------------------------------------------+
   |  select pivot element from this column by     |
   |  choosing nonzero whose row is of minimal     |
   |  degree                                      */

  rowdeg = m + 1;
  for (k = 0; k < coldeg; k++) {
    if (degBt[B[col][k].rowIndex] < rowdeg && ABS( B[col][k].value ) >
        EPSNUM) {
      row = B[col][k].rowIndex;
      rowdeg = degBt[row];
    }
  }
  if (rowdeg == m + 1) {
    hkey[col] = m + 2;
    hfall(heapnum, hkey, iheap, heap, iheap[col]);
    if (hkey[heap[1]] == m + 2) {
```

```
    printf("singular matrix. rank deficiency = %d\n", m - i);
    rank = i;
    goto end;
  } else {
    goto again;
  }
}

/*-----------------------------------------------+
 |  update permutation information               */

colPerm.at(i) = col;
invColPerm.at(col) = i;

rowPerm.at(i) = row;
invRowPerm.at(row) = i;

/*-----------------------------------------------+
 |  reallocate space for L, L.rowIndices, UT.values, and UT.rowIndices as
       |
 |       necessary.                              |
 |                                               |
 |  lnz stores the number of nonzeros in L so far  |
 |  lnzbnd is an estimate of how many will be in L |
 |  unz stores the number of nonzeros in U so far  |
 |  unzbnd is an estimate of how many will be in U*/

cnt = lnz + coldeg - 1 + coldeg * rowdeg / 2;
if (cnt > lnzbnd) {
  lnzbnd = cnt;
  REALLOC( L.values, lnzbnd, TYPE );
  REALLOC( L.rowIndices, lnzbnd, int );
}

cnt = unz + rowdeg - 1 + coldeg * rowdeg / 2;
if (cnt > unzbnd) {
  unzbnd = cnt;
  REALLOC( UT.values, unzbnd, TYPE );
  REALLOC( UT.rowIndices, unzbnd, int );
}

/*-----------------------------------------------+
 |  copy pivot column into L and pivot row into    |
 |  Ut.                                          */

L.colPos[i + 1] = L.colPos[i] + coldeg - 1;
```

```
for (k = 0; k < coldeg; k++) {
  if (B[col][k].rowIndex != row) {
    L.rowIndices[lnz] = B[col][k].rowIndex;
    L.values[lnz] = B[col][k].value;
    lnz++;
  }
}

UT.colPos[i + 1] = UT.colPos[i] + rowdeg - 1;

for (k = 0; k < rowdeg; k++) {
  if (Bt[row][k].rowIndex != col) {
    UT.rowIndices[unz] = Bt[row][k].rowIndex;
    UT.values[unz] = Bt[row][k].value;
    unz++;
  } else {
    diagU[i] = Bt[row][k].value;
  }
}

/*-----------------------------------------------+
 |  remove eliminated elements from B and Bt      */

for (k = 0; k < coldeg; k++) {
  row2 = B[col][k].rowIndex;
  degBt[row2]--;
  for (kk = 0; Bt[row2][kk].rowIndex != col; kk++)
    ;

  tempB = Bt[row2][degBt[row2]];
  Bt[row2][degBt[row2]] = Bt[row2][kk];
  Bt[row2][kk] = tempB;
}

for (k = 0; k < rowdeg; k++) {
  col2 = Bt[row][k].rowIndex;
  degB[col2]--;
  for (kk = 0; B[col2][kk].rowIndex != row; kk++)
    ;

  tempB = B[col2][degB[col2]];
  B[col2][degB[col2]] = B[col2][kk];
  B[col2][kk] = tempB;
}
degB[col] = 0;
degBt[row] = 0;
```

```
/*-----------------------------------------------+
 |  update heap                                  */

okey = hkey[col];
heap[1] = heap[heapnum];
iheap[heap[1]] = 1;
heapnum--;
if (okey < hkey[heap[1]])
  hfall(heapnum, hkey, iheap, heap, 1);

/*-----------------------------------------------+
 |  generate fillin and update elements          */

for (k = L.colPos[i]; k < L.colPos[i + 1]; k++) {
  row2 = L.rowIndices[k];
  tag++;
  for (kk = 0; kk < degBt[row2]; kk++) {
    iwork[Bt[row2][kk].rowIndex] = tag; /* tag these columns */
    iwork2[Bt[row2][kk].rowIndex] = kk; /* say where they are */
  }
  for (kk = UT.colPos[i]; kk < UT.colPos[i + 1]; kk++) {
    col2 = UT.rowIndices[kk];
    if (iwork[col2] == tag) {
      Bt[row2][iwork2[col2]].value -= L.values[k] * UT.values[kk]
          / diagU[i];
    } else {
      deg = degBt[row2];
      REALLOC( Bt[row2], deg+1, ValueAndRowIndex );
      Bt[row2][deg].rowIndex = col2;
      Bt[row2][deg].value = -L.values[k] * UT.values[kk] / diagU[i];
      degBt[row2]++;
    }
  }
}

for (k = UT.colPos[i]; k < UT.colPos[i + 1]; k++) {
  col2 = UT.rowIndices[k];
  tag++;
  for (kk = 0; kk < degB[col2]; kk++) {
    iwork[B[col2][kk].rowIndex] = tag; /* tag these rows */
    iwork2[B[col2][kk].rowIndex] = kk; /* say where they are */
  }
  for (kk = L.colPos[i]; kk < L.colPos[i + 1]; kk++) {
    row2 = L.rowIndices[kk];
    if (iwork[row2] == tag) {
      B[col2][iwork2[row2]].value -= L.values[kk] * UT.values[k]
          / diagU[i];
```

```
      } else {
        deg = degB[col2];
        REALLOC( B[col2], deg+1, ValueAndRowIndex );
        B[col2][deg].rowIndex = row2;
        B[col2][deg].value = -L.values[kk] * UT.values[k] / diagU[i];
        degB[col2]++;
      }
    }
  }

  /*--------------------------------------------+
   |  adjust heap                                |*/

  for (k = UT.colPos[i]; k < UT.colPos[i + 1]; k++) {
    col2 = UT.rowIndices[k];
    if (method == MD) {
      hkey[col2] = degB[col2];
    } else {
      hkey[col2] = col2;
    }
    if (hkey[col2] == 0)
      hkey[col2] = m + 1;
    hrise(hkey, iheap, heap, iheap[col2]);
    hfall(heapnum, hkey, iheap, heap, iheap[col2]);
  }

  /*--------------------------------------------+
   |  free space no longer needed                |*/

  /*
   FREE(B[col]);  FREE(Bt[row]);
   */
}
end:
/*--------------------------------------------+
 |  process dependent rows/cols                |*/

i = rank;
for (col = 0; col < m; col++) {
  if (invColPerm.at(col) == -1) {
    colPerm.at(i) = col;
    invColPerm.at(col) = i;
    i++;
  }
}

i = rank;
```

```
for (row = 0; row < m; row++) {
  if (invRowPerm.at(row) == -1) {
    rowPerm.at(i) = row;
    invRowPerm.at(row) = i;
    i++;
  }
}

for (i = rank; i < m; i++) {
  L.colPos[i + 1] = L.colPos[i];
  UT.colPos[i + 1] = UT.colPos[i];
  diagU[i] = 0.0;
}

/*-------------------------------------------+
 |  free up space                            */

heap++;
for (i = 0; i < m; i++) {
  FREE( B[i] );
  FREE( Bt[i] );
}
FREE(degB);
FREE(degBt);
FREE(hkey);
FREE(heap);
FREE(iheap);
FREE(iwork);
FREE(iwork2);
FREE(B);
FREE(Bt);

/*-------------------------------------------+
 |  update "i" arrays to new indices         */

for (k = 0; k < L.colPos[m]; k++)
  L.rowIndices[k] = invRowPerm.at(L.rowIndices[k]);
for (k = 0; k < UT.colPos[m]; k++)
  UT.rowIndices[k] = invColPerm.at(UT.rowIndices[k]);

/*-------------------------------------------+
 |  divide each column of L by diagonal      */

for (i = 0; i < m; i++) {
  for (k = L.colPos[i]; k < L.colPos[i + 1]; k++) {
    L.values[k] /= diagU[i];
  }
```

```
}

/*-------------------------------------------------------+
 | calculate and print statistics.                       */

narth = 0.0e0;
for (i = 0; i < m; i++) {
  k = L.colPos[i + 1] - L.colPos[i];
  narth += (TYPE) k * k;
  k = UT.colPos[i + 1] - UT.colPos[i];
  narth += (TYPE) k * k;
}
narth = narth + 3* L .colPos[m] + 3* UT .colPos[m] + 2* m ;

lnz = L.colPos[m];
unz = UT.colPos[m];
if (verbose) {
  printf("%9d   %9d %15.0f ", lnz, unz, narth);
  fflush(stdout);
}

if (LT.values == NULL) {
  MALLOC( LT.values, lnz, TYPE);
} else {
  REALLOC( LT.values, lnz, TYPE);
}
if (LT.rowIndices == NULL) {
  MALLOC( LT.rowIndices, lnz, int);
} else {
  REALLOC( LT.rowIndices, lnz, int);
}
if (LT.colPos == NULL) {
  MALLOC( LT.colPos, m+1, int);
} else {
  REALLOC( LT.colPos, m+1, int);
}

if (U.values == NULL) {
  MALLOC( U.values, unz, TYPE);
} else {
  REALLOC( U.values, unz, TYPE);
}
if (U.rowIndices == NULL) {
  MALLOC( U.rowIndices, unz, int);
} else {
  REALLOC( U.rowIndices, unz, int);
}
```

```
  if (U.colPos == NULL) {
    MALLOC( U.colPos, m+1, int);
  } else {
    REALLOC( U.colPos, m+1, int);
  }

  L.rows = LT.rows = U.rows = UT.rows = L.cols = LT.cols = U.cols = UT.cols =
      m;
  L.numNonzeroes = lnz;
  LT.numNonzeroes = lnz;
  U.numNonzeroes = unz;
  UT.numNonzeroes = unz;
  transposeMatrix(m, m, L.colPos, L.rowIndices, L.values, LT.colPos, LT.
      rowIndices, LT.values);
  transposeMatrix(m, m, UT.colPos, UT.rowIndices, UT.values, U.colPos, U.
      rowIndices, U.values);

  endtime = (double) clock();
  cumtime += endtime - starttime;
}
```

simpo/iterationprocess.h and simpo/iterationprocess.c

```
#ifndef ITERATIONPROCESS_H
#define ITERATIONPROCESS_H

#include <pthread.h>
#include "sparse.h"
#include "../common/genericvectors.h"

typedef struct {
  SparseMatrix L;
  SparseMatrix LT;
  SparseMatrix U;
  SparseMatrix UT;

  SafeVector<TYPE> diagU;
  SafeVector<int> colPerm;
  SafeVector<int> invColPerm;
  SafeVector<int> rowPerm;
  SafeVector<int> invRowPerm;
  int rank;

  int basisNumber;
} FactoredInverse;
```

```
/* The eta file seems to be stored as one sparse matrix,
 * where each column represents one eta matrix. */
typedef struct {
  int iter;       /* Number of eta matrices */
  int size;       /* Number of entries in 'values' */
  SafeVector<TYPE> values;      /* Nonzero values */
  SafeVector<int> rowIndices;   /* Row indices - seem to be based on 'colPerm'
      */
  SafeVector<int> colPos;     /* Indices into 'values' and 'rowIndices'
     telling where each eta matrix column begins */
  SafeVector<int> newColLocations;  /* Column location for each eta matrix (
     leaving column for corresponding pivot) */
} EtaFile;

typedef struct {
  TYPE * yInternal;
  int * tagInternal;
  int currtagInternal;
} InternalData_SolveUsingLU;

typedef struct {
  TYPE * yInternal;
  int * tagInternal;
  int currtagInternal;
} InternalData_SolveTransposeUsingLU;

typedef struct {
  TYPE * a;
  int * tag;
  int * link;
  int currtag;
} InternalData_Gauss_Eta;

typedef struct {
  TYPE * a;
  int * tag;
  int currtag;
} InternalData_Gauss_Eta_T;

typedef struct {
  TYPE * a;
  int * tag;
  int * link;
  int currtag;
} InternalData_Nt_times_y;

typedef struct {
```

```
  int index;
  int numIterationProcesses;
  const char * name;
  char ** iterationProcessNames;
  pthread_t thread; // It's safe to copy pthread_t objects: http://newsgroups.
     derkeiler.com/Archive/Comp/comp.programming.threads/2007-07/msg00283.
     html

  SparseMatrix A;
  SparseMatrix AT;
  SafeVector<TYPE> b; /* right-hand side */
  SafeVector<TYPE> c; /* objective coefficients */
  TYPE f;   /* objective function shift */
  SafeVector<TYPE> x_B; /* primal basics */
  SafeVector<TYPE> y_N; /* dual nonbasics */

  int currentBasis;

  FactoredInverse factoredInverse;

  int     e_iter; /* number of iterations since last refactorization */

  EtaFile etaFile;

  SafeSparseVector dyN_dualStepDir;
  SafeSparseVector dxB_primalStepDir;
  SafeSparseVector tempVector;

  SafeVector<int> basics;   /* list of basic variable indices */
  SafeVector<int> nonbasics;  /* list of non-basic variable indices */
  SafeVector<int> basicflag;  /* negative if nonbasic, nonnegative if basic */

  InternalData_SolveUsingLU internalData_SolveUsingLU;
  InternalData_SolveTransposeUsingLU internalData_SolveTransposeUsingLU;
  InternalData_Gauss_Eta internalData_Gauss_Eta;
  InternalData_Gauss_Eta_T internalData_Gauss_Eta_T;
  InternalData_Nt_times_y internalData_Nt_times_y;

  double cumtime; //TODO: This one was used both by the functions that were
     moved to invertprocessor.c and to interationprocess.c
  double ocumtime;
} IterationProcess;

typedef struct {
  int var;
  TYPE value;
} AttractiveCandidate;
```

```
void setupIterationProcess(IterationProcess * process, int index, int
   numIterationProcesses, SparseMatrix A, TYPE * b, TYPE * c, TYPE f);
void * runIterationProcess(void * parameters);

#endif
```

```
#include "payloadtypes.h"
#include "iterationprocess.h"
#include "invertprocessor.h"
#include "sparse.h"
#include "communication.h"
#include "print.h"
#include "util.h"
#include "sparse.h"
#include "../common/macros.h"
#include "../common/myalloc.h"
#include "../common/tree.h"
#include "../common/linalg.h"
#include "../common/message.h"
#include <stdio.h>
#include <math.h>
#include <assert.h>
using namespace std;

#define EPS_MATRIX 1.0e-8
#define EPSSOL  1.0e-5   /* Zero tolerance for consistent eqns w/dep rows */
#define E_N   200
#define E_NZ  20000
#define SOLVE_USING_LU_EPS  1.0e-14 /* EPS from lueta.c */
#define ITERATION_EPS   1.0e-8  /* EPS from 2phase.c */
#define PICK_NEG_EPS    1.0e-10 /* EPS2 from 2phase.c */
#define GAUSS_ETA_EPS   1.0e-14 /* EPS1 from lueta.c */
#define RATIO_TEST_ETA    1.0e-10 /* EPS1 from 2phase.c */
#define MAX_ITER  1000000

//TODO: Can we find a drand48() implementation for windows?
#ifdef WIN32
#define drand48() (((double)rand())/((double)RAND_MAX))
#endif

#ifdef QuadPrec
#include "Quad.h"
#define TYPE Quad
#else
#define high(x) (x)
#endif
```

```
// Ugly to have this here, but putting it in genericvectors caused heavy
   circular dependency problems. Should me moved.
#define VECTOR_NAME AttractiveCandidateVector
#define VECTOR_TYPE AttractiveCandidate
#define VECTOR_INIT initACV
#define VECTOR_RESIZE resizeACV
#define VECTOR_FREE freeACV
#define VECTOR_APPEND appendACV
#define VECTOR_GET getACV
#define VECTOR_SET setACV
#define VECTOR_REMOVE_AT removeAtACV
#define VECTOR_COPY copyACV
#include "../common/genericvector_headers.h"
#include "../common/genericvector_source.h"
#undef VECTOR_COPY
#undef VECTOR_REMOVE_AT
#undef VECTOR_SET
#undef VECTOR_GET
#undef VECTOR_APPEND
#undef VECTOR_FREE
#undef VECTOR_RESIZE
#undef VECTOR_INIT
#undef VECTOR_TYPE
#undef VECTOR_SHORT_NAME
#undef VECTOR_NAME




static int iterationProcess_dbsolve(IterationProcess * proc, SafeVector<TYPE>
   & y);
static int iterationProcess_solveUsingLU(IterationProcess * proc,
   SafeSparseVector * y);
static int iterationProcess_solveTransposeUsingLU(IterationProcess * proc,
   SafeSparseVector * y);
static void iterationProcess_Gauss_Eta(IterationProcess * proc,
   SafeSparseVector & dx_B);
static void iterationProcess_Gauss_Eta_T(IterationProcess * proc,
   SafeSparseVector & vec);
//static void iterationProcess_lu_clo(IterationProcess * proc);
static void iterationProcess_Nt_times_y(IterationProcess * proc, SparseMatrix
   AT, const SafeVector<int> & basicflag, const SafeSparseVector & y,
   SafeSparseVector & yN);
static int CHUZR1_findLeaving_ratioTest(IterationProcess * proc, const
   SafeSparseVector & dy, const SafeVector<TYPE> & y);
```

```
static TYPE iterationProcess_recalculateObjectiveValues(const SafeVector<TYPE>
    & c, const SafeVector<TYPE> & x_B, const SafeVector<int> & basics);
static void CHUZC_findAttractiveCandidates(const SafeVector<int> &
   columnToVariableMapping, const SafeVector<TYPE> & x,
   AttractiveCandidateVector * attractive);

static void send(IterationProcess * proc, const char * receiver, const char *
   tag, void * payload) {
  Message msg = {proc->name, receiver, tag, payload};
  sendMessage(msg);
}

#define receiveFromAnyone() receiveMessageFromAnyone(proc.name)

#define receive(sender) receiveMessage(proc.name, sender)

#define receiveWithTag(sender, tag) receiveMessageWithTag(proc.name, sender,
   tag)


int enablePrint = 1;

//http://ou800doc.sco.com/cgi-bin/info2html?(gcc.info)Macro%2520Varargs&lang=
   en
#define print(formatString, ...) if (enablePrint) { printFromProcess(proc.name
   , formatString , ## __VA_ARGS__); }
#define printp(formatString, ...) if (enablePrint) { printFromProcess(proc->
   name, formatString , ## __VA_ARGS__); }


#define streq(a, b) (!strcmp(a, b))

#define isBasic(var) (proc.basicflag[(var)] >= 0)
#define isNonbasic(var) (proc.basicflag[(var)] < 0)

#define columnForVariable(var) (proc.basicflag[(var)] >= 0 ? proc.basicflag[(
   var)] : -proc.basicflag[(var)] - 1)
#define columnForVariableP(var) (proc->basicflag[(var)] >= 0 ? proc->basicflag
   [(var)] : -proc->basicflag[(var)] - 1)


void makeIdentityMatrix(FactoredInverse * inv, int size) {
  int i;
  makeZeroSparseMatrix(&inv->L, size, size);
  makeZeroSparseMatrix(&inv->LT, size, size);
  makeZeroSparseMatrix(&inv->U, size, size);
  makeZeroSparseMatrix(&inv->UT, size, size);
```

```
  // All permutation vectors are filled with 0, size-1, ..., 1
  inv->colPerm.resize(size);
  inv->colPerm.at(0) = 0;
  for (i = 1; i < size; ++i) {
    inv->colPerm.at(i) = size - i;
  }
  inv->invColPerm = inv->colPerm;
  inv->rowPerm = inv->colPerm;
  inv->invRowPerm = inv->colPerm;
  // Diagonal is only 1s
  inv->diagU.resize(size);
  for (i = 0; i < size; ++i) {
    inv->diagU[i] = 1;
  }
  inv->rank = size;
  inv->basisNumber = 0;
}

void setupIterationProcess(IterationProcess * process, int index, int
    numIterationProcesses, SparseMatrix A, TYPE * b, TYPE * c, TYPE f) {
  int i;
  int m = A.rows;
  int N = A.cols;
//  int n = N - m;
  char * name = (char*)malloc(16); // Long enough for any conceivable index
      number
  sprintf(name, "I%d", index);
  process->name = name;
  process->index = index;
  process->numIterationProcesses = numIterationProcesses;
  copySparseMatrix(A, &process->A);
  for (i = 0; i < m; ++i)
    process->b.push_back(b[i]);
  for (i = 0; i < N; ++i)
    process->c.push_back(c[i]);
  process->f = f;

  process->currentBasis = 0;

  makeIdentityMatrix(&process->factoredInverse, m);

  // Reset eta file
  //TODO: Make eta file expandable
  //TODO: Do partial reset only
  process->etaFile.iter = 0; // Updated in refactor
  process->etaFile.size = 0; // Updated at the end of solveUsingLU
  process->etaFile.colPos.push_back(0);
```

```
  process->tempVector.setRows(m);

  process->internalData_SolveUsingLU.currtagInternal = 1;
  CALLOC( process->internalData_SolveUsingLU.yInternal, m, TYPE);
  CALLOC( process->internalData_SolveUsingLU.tagInternal, m, int);

  process->internalData_SolveTransposeUsingLU.currtagInternal = 1;
  CALLOC( process->internalData_SolveTransposeUsingLU.yInternal, m, TYPE);
  CALLOC( process->internalData_SolveTransposeUsingLU.tagInternal, m, int);

  process->internalData_Gauss_Eta.currtag = 1;
  CALLOC( process->internalData_Gauss_Eta.a, m, TYPE);
  CALLOC( process->internalData_Gauss_Eta.tag, m, int);
  CALLOC( process->internalData_Gauss_Eta.link, m + 2, int);
  process->internalData_Gauss_Eta.link++; //TODO: Call free() on link-1 !

  process->internalData_Gauss_Eta_T.currtag = 1;
  CALLOC( process->internalData_Gauss_Eta_T.a, m, TYPE);
  CALLOC( process->internalData_Gauss_Eta_T.tag, m, int);

  process->internalData_Nt_times_y.currtag = 1;
  MALLOC( process->internalData_Nt_times_y.a, N, TYPE);
  CALLOC( process->internalData_Nt_times_y.tag, N, int);
  CALLOC( process->internalData_Nt_times_y.link, N + 2, int);
  process->internalData_Nt_times_y.link++; //TODO: Call free() on link-1 !

  process->cumtime = 0;
  process->ocumtime = 0;

  process->iterationProcessNames = (char**)malloc(numIterationProcesses *
      sizeof(char *));
  for (i = 0; i < numIterationProcesses; ++i) {
    process->iterationProcessNames[i] = (char*)malloc(16);
    sprintf(process->iterationProcessNames[i], "I%d", i);
  }
}

static void expandEtaFile(IterationProcess * proc, int col_out) {
  proc->etaFile.newColLocations.push_back(col_out);
  proc->etaFile.iter++;
}

//TODO: Make 'peek' functionality so that we can ignore all but the newest
    inverse
// Repeatedly checks for messages from V with the tag "V2->I1".
// Such a message will contain a more up-to-date LU factorisation of the basis
```

```
    matrix than what this iteration process already has.
// As long as a message is found, the new inverse will be installed (proc->
   factoredInverse will be replaced by the payload of the message)
// and a suitable amount of entries from the front of the eta file will be
   removed.
void installNewInverse(IterationProcess * proc) {
  Message message;
  FactoredInverse * inversePayload;
  int i;

  while ((message = receiveMessage(proc->name, "V")).sender) {
    assert(!strcmp(message.tag, "V2->I1"));
    inversePayload = (FactoredInverse*)message.payload;
    //TODO: Restructure loops
    if (inversePayload->basisNumber <= proc->factoredInverse.basisNumber) {
      printp("Got outdated  B^I for basis %d; currently at basis %d",
          inversePayload->basisNumber, proc->factoredInverse.basisNumber);
    }
    else if (inversePayload->basisNumber > proc->currentBasis) {
      printp("Got too new B^I for basis %d; process is at basis %d; inverse is
           at basis %d", inversePayload->basisNumber, proc->currentBasis, proc
          ->factoredInverse.basisNumber);
    }
    else {
      int numEtaEntriesToRemove;
      int newPos;
      printp("Got acceptable B^I for basis %d; process is at basis %d; inverse
           is at basis %d", inversePayload->basisNumber, proc->currentBasis,
          proc->factoredInverse.basisNumber);
      printp("installNewInverse is modifying eta file");
      numEtaEntriesToRemove = inversePayload->basisNumber - proc->
          factoredInverse.basisNumber;
      newPos = proc->etaFile.colPos.at(numEtaEntriesToRemove);
      printp("%d %d", numEtaEntriesToRemove, newPos);
      proc->etaFile.size -= newPos;
      proc->etaFile.iter -= numEtaEntriesToRemove;
      // TODO: Can we use some sort of queue structure to make this more
          efficient?
      for (i = 0; i < proc->etaFile.size; ++i) {
        proc->etaFile.rowIndices.at(i) = proc->etaFile.rowIndices.at(i +
            newPos);
        proc->etaFile.values.at(i) = proc->etaFile.values.at(i + newPos);
      }
      proc->etaFile.rowIndices.resize(proc->etaFile.size);
      proc->etaFile.values.resize(proc->etaFile.size);
      for (i = 0; i < proc->etaFile.iter + 1; ++i) {
        proc->etaFile.colPos.at(i) = proc->etaFile.colPos.at(i +
```

```
            numEtaEntriesToRemove) - newPos;
      }
      proc->etaFile.colPos.resize(proc->etaFile.iter + 1);
      assert(proc->etaFile.colPos.at(0) == 0);
      for (i = 0; i < proc->etaFile.iter; ++i) {
        proc->etaFile.newColLocations.at(i) = proc->etaFile.newColLocations.at
            (i + numEtaEntriesToRemove);
      }
      proc->etaFile.newColLocations.resize(proc->etaFile.iter);
      // Free the space occupied by the previous matrix, copy the message
         payload and free the original payload
      freeSparseMatrix(&proc->factoredInverse.L);
      freeSparseMatrix(&proc->factoredInverse.LT);
      freeSparseMatrix(&proc->factoredInverse.U);
      freeSparseMatrix(&proc->factoredInverse.UT);
      proc->factoredInverse = *inversePayload;
      printp("basis: %d", proc->factoredInverse.basisNumber);
    }
    delete inversePayload;
  }
}

/*************************************************************
 *                            -1  T                          *
 * STEP 2/4: Compute dy  = -(B  N) e                         *
 *                     N           i                  *
 *           where i = col_out                               *
 *************************************************************/
// proc->dyN_dualStepDir will be updated.
// The objective function coefficients will change by dyN_dualStepDir *
   s_dualStepLength (which will be calculated later).
void FTRAN_calculateDualStepDirection(IterationProcess * proc, int col_out) {
  proc->tempVector.resize(1);
  proc->tempVector.value(0) = -1.0;
  proc->tempVector.rowIndex(0) = col_out;
  iterationProcess_solveTransposeUsingLU(proc, &proc->tempVector);
  iterationProcess_Nt_times_y(proc, proc->AT, proc->basicflag, proc->
     tempVector, proc->dyN_dualStepDir);
}

/*************************************************************
 *                          -1                               *
 * STEP 4/2: Compute dx  = B  N e                            *
 *                     B         j                           *
 *           where j = col_in                                *
 *************************************************************/
// proc->dxB_primalStepDir will be updated.
```

```
// The right hand side will change by dxB_primalStepDir * t_primalStepLength (
    which will be calculated later).
// This function calls iterationProcess_solveUsingLU(), which as a side effect
     will partially update the eta file - but etaFile.iter won't be touched,
    so the change won't be visible until expandEtaFile() is called.
void FTRAN_calculatePrimalStepDirection(IterationProcess * proc, int col_in) {
  int i, j, k;
  j = proc->nonbasics[col_in];
  proc->dxB_primalStepDir.clear();
  for (i = 0, k = proc->A.colPos[j]; k < proc->A.colPos[j + 1]; i++, k++) {
    proc->dxB_primalStepDir.append(proc->A.values[k], proc->A.rowIndices[k]);
  }
  iterationProcess_solveUsingLU(proc, &proc->dxB_primalStepDir);
}

/*************************************************************
 *                                                           *
 * STEP 5: Put       t = x /dx                               *
 *                        i   i                              *
 *                   s = y /dy                               *
 *                        j   j                              *
 *************************************************************/
// Given the leaving and entering columns, computes the factors that
    dxB_primalStepDir and dyN_dualStepDir must be multiplied by.
void CHUZR2_calculateStepLengths(IterationProcess * proc, int col_in, int
    col_out, TYPE * t_primalStepLength, TYPE * s_dualStepLength) {
  int k;

  /* this is inefficient - it should be fixed */
  for (k = 0; k < proc->dxB_primalStepDir.numNonzeroes(); k++)
    if (proc->dxB_primalStepDir.rowIndex(k) == col_out)
      break;

  *t_primalStepLength = proc->x_B[col_out] / proc->dxB_primalStepDir.value(k);

  /* this is inefficient - it should be fixed */
  for (k = 0; k < proc->dyN_dualStepDir.numNonzeroes(); k++)
    if (proc->dyN_dualStepDir.rowIndex(k) == col_in)
      break;

  *s_dualStepLength = proc->y_N[col_in] / proc->dyN_dualStepDir.value(k);
}

/*************************************************************
 * STEP 7: Set y  = y  - s dy                                *
 *              N    N       N                               *
 *                                                           *
```

```
 *             y  = s                                                *
 *              i                                                    *
 *                                                                   *
 *             x  = x  - t dx                                        *
 *              B    B       B                                       *
 *                                                                   *
 *             x  = t                                                *
 *              j                                                    *
 ********************************************************/
void UPRHS_updatePrimalDualSolutions(IterationProcess * proc, int col_in, int
    col_out, TYPE primalStepLength, TYPE dualStepLength) {
  int i, j, k;
  for (k = 0; k < proc->dyN_dualStepDir.numNonzeroes(); k++) {
    j = proc->dyN_dualStepDir.rowIndex(k);
    proc->y_N[j] -= dualStepLength * proc->dyN_dualStepDir.value(k);
  }

  proc->y_N[col_in] = dualStepLength;

  for (k = 0; k < proc->dxB_primalStepDir.numNonzeroes(); k++) {
    i = proc->dxB_primalStepDir.rowIndex(k);
    proc->x_B[i] -= primalStepLength * proc->dxB_primalStepDir.value(k);
  }

  proc->x_B[col_out] = primalStepLength;
}

/************************************************************
 * STEP 8: Update basis                                      *
 ************************************************************/
void UPDATE_BASIS(IterationProcess * proc, int col_in, int col_out, int
    callRefactor) {
  int i, j;
  i = proc->basics[col_out];
  j = proc->nonbasics[col_in];
  proc->basics[col_out] = j;
  proc->nonbasics[col_in] = i;
  proc->basicflag[i] = -col_in - 1;
  proc->basicflag[j] = col_out;
  //if (callRefactor)
    //iterationProcess_refactor(proc, proc->basics, col_out);
}

void applyIncomingBasisChange(IterationProcess * proc, Message message) {
  BasisChangePayload * payload = (BasisChangePayload*)message.payload;
  int i;
  printp("Applying incoming basis change: to %d", payload->fromBasis + 1);
```

```
  assert(payload->fromBasis == proc->currentBasis);
  proc->dxB_primalStepDir = payload->primalStepDirection;
  proc->dyN_dualStepDir = payload->dualStepDirection;
  for (i = 0; i < payload->etaFileSizeDiff; ++i) {
    proc->etaFile.values.push_back(payload->newEtaFileValues.at(i));
    proc->etaFile.rowIndices.push_back(payload->newEtaFileRowIndices.at(i));
  }
  proc->etaFile.size += payload->etaFileSizeDiff;
  proc->etaFile.colPos.push_back(proc->etaFile.colPos.at(proc->etaFile.iter) +
      payload->etaFileSizeDiff);
  proc->etaFile.newColLocations.push_back(payload->leavingRow);
  proc->etaFile.iter++;

  UPDATE_BASIS(proc, payload->enteringCol, payload->leavingRow, 0);
  UPRHS_updatePrimalDualSolutions(proc, payload->enteringCol, payload->
      leavingRow, payload->primalStepLength, payload->dualStepLength);
  ++proc->currentBasis;
  delete payload;
}

void applyIncomingBasisChanges(IterationProcess * proc) {
  Message message;
  while ((message = receiveMessageWithTag(proc->name, "I*", "I7->I2I3I10")).
      sender) {
    assert(!strcmp(message.tag, "I7->I2I3I10"));
    applyIncomingBasisChange(proc, message);
  }
}

// The eta file must be fully updated, including etaFile.iter, before this
    function is called.
void broadcastBasisChange(IterationProcess * proc, int fromBasis, int col_in,
    int col_out, int var_in, int var_out, TYPE primalStepLength, TYPE
    dualStepLength) {
  InvProcPivotPayload * invProcPayload = (InvProcPivotPayload*)malloc(sizeof(
      InvProcPivotPayload)); // Will be free'd by InvProc
  ColSelPivotPayload * colSelPayload = (ColSelPivotPayload*)malloc(sizeof(
      ColSelPivotPayload)); // Will be free'd by ColSelProc
  int i, j, k;
  BasisChangePayload * basisChangePayload;
  Message * messages = new Message[proc->numIterationProcesses - 1];
  Message * message = messages;
  for (i = 0; i < proc->numIterationProcesses; ++i) {
    if (i == proc->index) continue;
    basisChangePayload = new BasisChangePayload;
    basisChangePayload->enteringCol = col_in;
    basisChangePayload->leavingRow = col_out;
```

```
    basisChangePayload->fromBasis = fromBasis;
    basisChangePayload->primalStepLength = primalStepLength;
    basisChangePayload->dualStepLength = dualStepLength;
    basisChangePayload->primalStepDirection = proc->dxB_primalStepDir;
    basisChangePayload->dualStepDirection = proc->dyN_dualStepDir;
    basisChangePayload->etaFileSizeDiff = proc->etaFile.colPos.at(proc->
       etaFile.iter) - proc->etaFile.colPos.at(proc->etaFile.iter - 1);
    for (j = 0, k = proc->etaFile.colPos.at(proc->etaFile.iter - 1); j <
       basisChangePayload->etaFileSizeDiff; ++j, ++k) {
      basisChangePayload->newEtaFileValues.push_back(proc->etaFile.values.at(k
         ));
      basisChangePayload->newEtaFileRowIndices.push_back(proc->etaFile.
         rowIndices.at(k));
    }
    printp("Sending basis change message to I%d (fromBasis == %d)", i,
       fromBasis);
    message->sender = proc->name;
    message->receiver = proc->iterationProcessNames[i];
    message->tag = "I7->I2I3I10";
    message->payload = basisChangePayload;
    ++message;
  }
  sendAll(messages, proc->numIterationProcesses - 1);
  delete [] messages;
  invProcPayload->enteringCol = col_in;
  invProcPayload->leavingRow = col_out;
  send(proc, "V", "I8->V1", invProcPayload);
  colSelPayload->enteringVar = var_in;
  colSelPayload->leavingVar = var_out;
  colSelPayload->basisNumber = proc->currentBasis;
  send(proc, "C", "I8->C1", colSelPayload);
}

void sendAttractiveCandidates(IterationProcess * proc, int basisNumber,
   AttractiveCandidateVector * attractive) {
  int i;
  AttractiveCandidatesMessage * payload = new AttractiveCandidatesMessage;
  payload->basisNumber = basisNumber;
  for (i = 0; i < attractive->size; ++i) {
    payload->attractiveVariables.push_back(getACV(*attractive, i).var);
    //printp("%d is attractive", getACV(*attractive, i).col);
  }
  send(proc, "C", "I9->C2", payload);
}

void * runIterationProcess(void * parameters) {
  IterationProcess proc = *(IterationProcess*)parameters;
```

```
SafeSparseVector tempVector;

int col_in; /* entering column; index in 'nonbasics' */
int col_out; /* leaving column; index in 'basics' */
int var_out;
int var_in;

int iter = 0; /* number of iterations */
int i, j, k, m, n, N;

TYPE s_dualStepLength, t_primalStepLength;
float primal_obj;

TYPE *x; /* primal solution (output) */
TYPE *y; /* dual solution (output) */
TYPE *w; /* primal slacks (output) */
TYPE *z; /* dual slacks (output) */

Message message;
InvProcPivotPayload * pivotPayload;
ChuzrOfferPayload * chuzrOfferPayload;
ChuzrAcceptancePayload * chuzrAcceptancePayload;
CandidateIsUnattractivePayload * unattractivePayload;

AttractiveCandidateVector attractive;

int keepGoing;
int finished;
int accepted;

seedRandomGeneratorForThisThread();

/*****************************************************************
 * For convenience, we put...
 *****************************************************************/
m = proc.A.rows;
N = proc.A.cols;
n = N - m;
print("m = %d, n = %d, N = %d, nz = %d", m, n, N, proc.A.numNonzeroes);

/*****************************************************************
 * Read in the Data and initialize the common memory sites.
 *****************************************************************/

proc.x_B.resize(m);
proc.dxB_primalStepDir.setRows(m);
```

```
proc.y_N.resize(n);
proc.dyN_dualStepDir.setRows(n);
tempVector.setRows(m);
proc.nonbasics.resize(n);
proc.basics.resize(m);
proc.basicflag.resize(N);
MALLOC( proc.AT.values, proc.A.numNonzeroes, TYPE );
MALLOC( proc.AT.rowIndices, proc.A.numNonzeroes, int );
MALLOC( proc.AT.colPos, m+1, int );
nullifySparseMatrix(&proc.AT);
initACV(&attractive);

/*************************************************************
 *  Initialization.                                          *
 *************************************************************/

transposeSparseMatrix(proc.A, &proc.AT);

for (j = 0; j < n; j++) {
  proc.nonbasics[j] = j;
  proc.basicflag[j] = -j - 1;
  proc.y_N[j] = MAX(proc.c[j],1.0); /* to force dual feasibility */
  proc.y_N[j] += drand48(); /* to ensure nondegeneracy   */
}

for (i = 0; i < m; i++) {
  proc.basics[i] = n + i;
  proc.basicflag[n + i] = i;
  proc.x_B[i] = proc.b[i];
}
//TODO: lufac was called here
iterationProcess_dbsolve(&proc, proc.x_B); /* could be done explicitly in
   terms of bounds/ranges */

/*************************************************************
 *  Begin Phase I (i.e., dual simplex method)
 *************************************************************/

CHUZC_findAttractiveCandidates(proc.basics, proc.x_B, &attractive);
if (proc.index < attractive.size) {
  var_out = getACV(attractive, proc.index).var;
}
else {
  var_out = -1;
}

/*************************************************************
```

```
   *  Main loop                                                    *
   ***************************************************************/
  for (iter = 0; iter < MAX_ITER && !isProgramFinished(); iter++) {
    print("iteration %d", iter);
//    assert(iter == proc.currentBasis);
    installNewInverse(&proc);
    do {
      keepGoing = 0;
      finished = 0;

      // Not sure we need to do basis change and FTRAN inside the loop; it
         might suffice to do them outside
      applyIncomingBasisChanges(&proc);
      col_out = -1;
      if (var_out != -1 && isBasic(var_out)) {//TODO: isNonbasic in phase II
        col_out = columnForVariable(var_out);
        print("var_out: %d, col_out: %d", var_out, col_out);
        FTRAN_calculateDualStepDirection(&proc, col_out);
      }
      if (var_out == -1 || isNonbasic(var_out) /*TODO: isBasic in phase II*/
          || proc.x_B[col_out] > 0 /*TODO: y_N in phase II*/) { //TODO: > EPS
          or <= -EPS ?
        print("x%d is unattractive: col %d, basicflag %d, stepLength %f",
            var_out, col_out, (var_out == -1 ? -1 : proc.basicflag[var_out]),
            (col_out == -1 ? 0 : proc.x_B[col_out]));
        unattractivePayload = (CandidateIsUnattractivePayload*)malloc(sizeof(
            CandidateIsUnattractivePayload));
        unattractivePayload->var = var_out;
        unattractivePayload->basisNumber = proc.currentBasis;
        send(&proc, "C", "I4->C4", unattractivePayload);
        break; // After the loop, we wait for a new candidate
      }
      else {
        print("x%d is attractive: col %d, basicflag %d, stepLength %f",
            var_out, col_out, proc.basicflag[var_out], proc.x_B[col_out]);
      }
      chuzrOfferPayload = (ChuzrOfferPayload*)malloc(sizeof(ChuzrOfferPayload)
          );
      chuzrOfferPayload->basisNumber = proc.currentBasis;
      send(&proc, "R", "I5->R1", chuzrOfferPayload);
      while (!(message = receive("R")).sender) {
        if (isProgramFinished()) {
          pthread_exit(NULL);
          return NULL;
        }
        sched_yield();
      }
```

```
assert(streq(message.tag, "R2->I6") || streq(message.tag, "R3->I6"));
chuzrAcceptancePayload = (ChuzrAcceptancePayload*)message.payload;
accepted = chuzrAcceptancePayload->accepted;
free(chuzrAcceptancePayload);
if (accepted) {
  col_in = CHUZR1_findLeaving_ratioTest(&proc, proc.dyN_dualStepDir,
     proc.y_N);
  if (col_in == -1) {
    announceProgramFinished();
    print("INFEASIBLE");
    pthread_exit(NULL);
    return NULL;/*TODO: return 2; INFEASIBLE */
  }
  var_in = proc.nonbasics[col_in]; //TODO: basics in PII
  FTRAN_calculatePrimalStepDirection(&proc, col_in);
  CHUZR2_calculateStepLengths(&proc, col_in, col_out, &
     t_primalStepLength, &s_dualStepLength);
  expandEtaFile(&proc, col_out);
  broadcastBasisChange(&proc, proc.currentBasis, col_in, col_out, var_in
     , var_out, t_primalStepLength, s_dualStepLength); //TODO: Be
     careful - this is different in the two phases!
  UPRHS_updatePrimalDualSolutions(&proc, col_in, col_out,
     t_primalStepLength, s_dualStepLength);
  UPDATE_BASIS(&proc, col_in, col_out, 1);
  ++proc.currentBasis;

  primal_obj = iterationProcess_recalculateObjectiveValues(proc.c, proc.
     x_B, proc.basics) + proc.f;
  print("Basis %d:\t%14.7e", iter, high(primal_obj));
  if (proc.currentBasis % 100 == 0) {
    printf("Basis %d:\t%14.7e\n", proc.currentBasis, high(primal_obj));
    fflush(stdout);
  }

  CHUZC_findAttractiveCandidates(proc.basics, proc.x_B, &attractive);
  if (attractive.size == 0) {
    finished = 1;
    break; /* ready for Phase II */ //TODO: How to notify the others?
  }
  //col_out = getACV(attractive, 0).col;
  sendAttractiveCandidates(&proc, proc.currentBasis, &attractive);
}
else { //Not accepted
  while (!(message = receiveWithTag("I*", "I7->I2I3I10")).sender) {
    if (isProgramFinished()) {
      pthread_exit(NULL);
      return NULL;
```

```
        }
        sched_yield();
      }
      assert(streq(message.tag, "I7->I2I3I10"));
      applyIncomingBasisChange(&proc, message);
      keepGoing = 1;
    }
  } while (keepGoing);
  if (finished)
    break;

  while (!(message = receive("C")).sender) {
    if (isProgramFinished()) {
      pthread_exit(NULL);
      return NULL;
    }
    sched_yield();
  }
  assert(streq(message.tag, "C3->I11") || streq(message.tag, "C5->I11"));
  applyIncomingBasisChanges(&proc);
  var_out = *(int*)message.payload;
  free(message.payload);
  print("got suggested entering variable: x%d", var_out);
} /* Phase I loop */

print("End of Phase I");
if (announceProgramFinished()) {
  pthread_exit(NULL);
  return NULL;
}
printf("Phase I solution at basis %d: %14.7e\n", proc.currentBasis, high(
    primal_obj));
// At the current stage of development, we are at the point where Phase I '
    almost' works (except for a few threading glitches).
// Therefore, we have been experimenting with continuing with Phaase II, but
     we are unsure of how all threads should get there properly, not just
    the thread that discovers that Phase I is over.
// The above construct will make sure that only one of the threads reaches
    this point.
// We have added these two lines in order to shut the program down after
    Phase I so that we can perform some timing measurements.
pthread_exit(NULL);
return NULL;

/*************************************************************
 *   Restore objective function by setting                   *
 *                -1  T                                      *
```

```
 *          y  = (B  N) c  - c                                  *
 *           N           B    N                                  *
 ***************************************************************/

tempVector.clear();
for (i = 0; i < m; i++) {
  if (ABS(proc.c[proc.basics[i]]) > ITERATION_EPS) {
    tempVector.append(proc.c[proc.basics[i]], i);
  }
}

iterationProcess_solveTransposeUsingLU(&proc, &tempVector);

iterationProcess_Nt_times_y(&proc, proc.AT, proc.basicflag, tempVector, proc
    .dyN_dualStepDir); // Use dyN_dualStepDir temporarily (this is not the
    way it is normally used)

for (j = 0; j < n; j++)
  proc.y_N[j] = -proc.c[proc.nonbasics[j]];
for (k = 0; k < proc.dyN_dualStepDir.numNonzeroes(); k++) {
  j = proc.dyN_dualStepDir.rowIndex(k);
  proc.y_N[j] += proc.dyN_dualStepDir.value(k);
}

/*************************************************************
 *   Begin Phase II (I.e., primal simplex method)             *
 *************************************************************/

for (; iter < MAX_ITER; iter++) {
  primal_obj = iterationProcess_recalculateObjectiveValues(proc.c, proc.x_B,
      proc.basics) + proc.f;
  print("Iteration %d:\t%14.7e", iter, high(primal_obj));

  CHUZC_findAttractiveCandidates(proc.nonbasics, proc.y_N, &attractive);
  if (attractive.size == 0)
    break; /* optimal */
  var_in = getACV(attractive, 0).var;
  col_in = columnForVariable(var_in);

  installNewInverse(&proc);

  FTRAN_calculatePrimalStepDirection(&proc, col_in);

  col_out = CHUZR1_findLeaving_ratioTest(&proc, proc.dxB_primalStepDir, proc
      .x_B);
  if (col_out == -1) {
    announceProgramFinished();
```

```
    print("UNBOUNDED");
    pthread_exit(NULL);
    return NULL; /* return 1; UNBOUNDED */
  }
  var_out = proc.basics[col_out];

  //TODO: Just for testing messaging w/ InvProc
  pivotPayload = (InvProcPivotPayload*)malloc(sizeof(InvProcPivotPayload));
     // Will be free'd by InvProc
  pivotPayload->enteringCol = col_in;
  pivotPayload->leavingRow = col_out;
  send(&proc, "V", "I8->V1", pivotPayload);

  FTRAN_calculateDualStepDirection(&proc, col_out);

  CHUZR2_calculateStepLengths(&proc, col_in, col_out, &t_primalStepLength, &
     s_dualStepLength);

  expandEtaFile(&proc, col_out);

  UPRHS_updatePrimalDualSolutions(&proc, col_in, col_out, t_primalStepLength
     , s_dualStepLength);

  UPDATE_BASIS(&proc, col_in, col_out, 1);

} /* End of Phase II */

primal_obj = iterationProcess_recalculateObjectiveValues(proc.c, proc.x_B,
   proc.basics) + proc.f;
print("%8d   %14.7e     NA    ", iter, high(primal_obj));
print("End of Phase II \n");
printf("Iterations: %d\nOptimal solution: %14.7e\n", iter, high(primal_obj))
   ;

/*************************************************************
 *  Transcribe solution to x vector and dual solution to y      *
 *************************************************************/

x = (TYPE*)malloc(N * sizeof(TYPE));
for (j = 0; j < N; j++)
  x[j] = 0.0;
for (i = 0; i < m; i++)
  x[proc.basics[i]] = proc.x_B[i];
//printTypeArray(x, N);

y = (TYPE*)malloc(N * sizeof(TYPE));
for (j = 0; j < N; j++)
```

```
    y[j] = 0.0;
  for (i = 0; i < n; i++)
    y[proc.nonbasics[i]] = proc.y_N[i];

  /*************************************************************
   *  Split out slack variables and shift dual variables.
   *************************************************************/

  z = (TYPE*)malloc(n * sizeof(TYPE));
  for (j = 0; j < n; j++)
    z[j] = y[j];
  w = (TYPE*)malloc(m * sizeof(TYPE));
  for (i = 0; i < m; i++) {
    y[i] = y[n + i];
    w[i] = x[n + i];
  }

  announceProgramFinished();
  print("OPTIMAL");
  pthread_exit(NULL);
  return NULL;
}

/*-------------------------------------------------------------------+
| Forward/backward solve using LU factorization                     |
| Input:                                                            |
|    m          dimension of array y                                |
|    y          array containing right-hand side                    |
|                                                                   |
|    static global variables (assumed setup by lufac()):           |
|                                                                   |
|    rank       rank of B                                           |
|    L.colPos, L.rowIndices, L, three array sparse representation of L
               |
|    UT.colPos,UT.rowIndices,UT.values three array sparse representation of U
   transpose   |
|               without its diagonal                                |
|    diagU      diagonal entries of U                               |
|    colPerm, invColPerm, rowPerm, invRowPerm                           |
|               column and row permutations and their inverses      |
| Output:                                                           |
|                                          -1                       |
|    y          array containing solution B  y                      |
|                                                                   |
|    integer flag indicating whether system is consistent           |
+-------------------------------------------------------------------*/
// Will update the eta file based on col_out.
```

```
static int iterationProcess_solveUsingLU(IterationProcess * proc,
   SafeSparseVector * y) {
  int i;
  int k, row, consistent = TRUE;
  TYPE beta;
  TYPE eps;

  double starttime, endtime;

  Tree tree = {NULL, NULL};

  starttime = (double) clock();

  for (k = 0; k < y->numNonzeroes(); k++) {
    i = proc->factoredInverse.invRowPerm.at(y->rowIndex(k));
    proc->internalData_SolveUsingLU.yInternal[i] = y->value(k);
    proc->internalData_SolveUsingLU.tagInternal[i] = proc->
       internalData_SolveUsingLU.currtagInternal;
    addtree(&tree, i);
  }

  if (proc->factoredInverse.rank < y->numRows())
    eps = EPSSOL * y->maxValue();

  /*-----------------------------------------------------+
   |               -1                                    |
   |       y  <-  L  y                                   */

  for (i = getfirst(&tree); i < proc->factoredInverse.rank && i != -1; i =
     getnext(&tree)) {
    beta = proc->internalData_SolveUsingLU.yInternal[i];
    for (k = proc->factoredInverse.L.colPos[i]; k < proc->factoredInverse.L.
       colPos[i + 1]; k++) {
      row = proc->factoredInverse.L.rowIndices[k];
      if (proc->internalData_SolveUsingLU.tagInternal[row] != proc->
         internalData_SolveUsingLU.currtagInternal) {
        proc->internalData_SolveUsingLU.yInternal[row] = 0.0;
        proc->internalData_SolveUsingLU.tagInternal[row] = proc->
           internalData_SolveUsingLU.currtagInternal;
        addtree(&tree, row);
      }
      proc->internalData_SolveUsingLU.yInternal[row] -= proc->factoredInverse.
         L.values[k] * beta;
    }
  }

  /*-----------------------------------------------------+
```

```
 |              -1                                          |
 |       y  <-  U  y                                       */

for (i = getlast(&tree); i >= proc->factoredInverse.rank && i != -1; i =
    getprev(&tree)) {
  if (ABS( proc->internalData_SolveUsingLU.yInternal[i] ) > eps)
    consistent = FALSE;
  proc->internalData_SolveUsingLU.yInternal[i] = 0.0;
}

for (; i >= 0; i = getprev(&tree)) {
  beta = proc->internalData_SolveUsingLU.yInternal[i] / proc->
      factoredInverse.diagU[i];
  for (k = proc->factoredInverse.U.colPos[i]; k < proc->factoredInverse.U.
      colPos[i + 1]; k++) {
    row = proc->factoredInverse.U.rowIndices[k];
    if (proc->internalData_SolveUsingLU.tagInternal[row] != proc->
        internalData_SolveUsingLU.currtagInternal) {
      proc->internalData_SolveUsingLU.yInternal[row] = 0.0;
      proc->internalData_SolveUsingLU.tagInternal[row] = proc->
          internalData_SolveUsingLU.currtagInternal;
      addtree(&tree, row);
    }
    proc->internalData_SolveUsingLU.yInternal[row] -= proc->factoredInverse.
        U.values[k] * beta;
  }
  proc->internalData_SolveUsingLU.yInternal[i] = beta;
}

y->clear();
for (i = getfirst(&tree); i != -1; i = getnext(&tree)) {
  if (ABS(proc->internalData_SolveUsingLU.yInternal[i]) > SOLVE_USING_LU_EPS
      ) {
    y->append(proc->internalData_SolveUsingLU.yInternal[i], proc->
        factoredInverse.colPerm.at(i));
  }
}

proc->internalData_SolveUsingLU.currtagInternal++;
killtree(&tree);

iterationProcess_Gauss_Eta(proc, *y);

/*************************************************************
 * Update etaFile.values and save col_out in etaFile.newColLocations[etaFile
    .iter]                    *
 *************************************************************/
```

```
  printp("solveUsingLU is modifying eta file");
  for (i = 0, k = proc->etaFile.colPos.at(proc->etaFile.iter); i < y->
      numNonzeroes(); i++, k++) {
    proc->etaFile.values.push_back(y->value(i));
    proc->etaFile.rowIndices.push_back(y->rowIndex(i));
  }

  proc->etaFile.size = k;
  proc->etaFile.colPos.push_back(k);

  endtime = (double) clock();
  proc->cumtime += endtime - starttime;

  return consistent;
}

/*-------------------------------------------------------------------+
 | Forward/backward solve using LU factorization                     |
 | Input:                                                            |
 |    y->rows          dimension of array y                              |
 |    y          array containing right-hand side                    |
 |                                                                   |
 |    static global variables (assumed setup by lufac()):           |
 |                                                                   |
 |    rank       rank of B                                           |
 |    L.colPos, L.rowIndices, L, three array sparse representation of L
                |
 |    UT.colPos,UT.rowIndices,UT.values three array sparse representation of U
     transpose   |
 |               without its diagonal                                |
 |    diagU      diagonal entries of U                               |
 |    colPerm, invColPerm, rowPerm, invRowPerm                           |
 |               column and row permutations and their inverses      |
 | Output:                                                           |
 |                                          -T                       |
 |    y          array containing solution B  y                      |
 |                                                                   |
 |    integer flag indicating whether system is consistent          */

static int iterationProcess_solveTransposeUsingLU(IterationProcess * proc,
    SafeSparseVector * y) {
  int i;
  int k, row, consistent = TRUE;
  TYPE beta;
  TYPE eps;
```

```
double starttime, endtime;

Tree tree = {NULL, NULL};

starttime = (double) clock();

iterationProcess_Gauss_Eta_T(proc, *y);

for (k = 0; k < y->numNonzeroes(); k++) {
  i = proc->factoredInverse.invColPerm.at(y->rowIndex(k));
  proc->internalData_SolveTransposeUsingLU.yInternal[i] = y->value(k);
  proc->internalData_SolveTransposeUsingLU.tagInternal[i] = proc->
     internalData_SolveTransposeUsingLU.currtagInternal;
  addtree(&tree, i);
}

if (proc->factoredInverse.rank < y->numRows())
  eps = EPSSOL * y->maxValue();

/*-----------------------------------------------------+
 |               -T                                    |
 |       y  <-  U  y                                   */

for (i = getfirst(&tree); i < proc->factoredInverse.rank && i != -1; i =
    getnext(&tree)) {
  beta = proc->internalData_SolveTransposeUsingLU.yInternal[i] / proc->
     factoredInverse.diagU[i];
  for (k = proc->factoredInverse.UT.colPos[i]; k < proc->factoredInverse.UT.
     colPos[i + 1]; k++) {
    row = proc->factoredInverse.UT.rowIndices[k];
    if (proc->internalData_SolveTransposeUsingLU.tagInternal[row] != proc->
       internalData_SolveTransposeUsingLU.currtagInternal) {
      proc->internalData_SolveTransposeUsingLU.yInternal[row] = 0.0;
      proc->internalData_SolveTransposeUsingLU.tagInternal[row] = proc->
         internalData_SolveTransposeUsingLU.currtagInternal;
      addtree(&tree, row);
    }
    proc->internalData_SolveTransposeUsingLU.yInternal[row] -= proc->
       factoredInverse.UT.values[k] * beta;
  }
  proc->internalData_SolveTransposeUsingLU.yInternal[i] = beta;
}
for (i = getlast(&tree); i >= proc->factoredInverse.rank && i != -1; i =
    getprev(&tree)) {
  if (ABS( proc->internalData_SolveTransposeUsingLU.yInternal[i] ) > eps)
    consistent = FALSE;
  proc->internalData_SolveTransposeUsingLU.yInternal[i] = 0.0;
```

```
  }

  /*-------------------------------------------------+
   |               -T                                |
   |       y  <-  L  y                               */

  for (; i >= 0; i = getprev(&tree)) {
    beta = proc->internalData_SolveTransposeUsingLU.yInternal[i];
    for (k = proc->factoredInverse.LT.colPos[i]; k < proc->factoredInverse.LT.
        colPos[i + 1]; k++) {
      row = proc->factoredInverse.LT.rowIndices[k];
      if (proc->internalData_SolveTransposeUsingLU.tagInternal[row] != proc->
          internalData_SolveTransposeUsingLU.currtagInternal) {
        proc->internalData_SolveTransposeUsingLU.yInternal[row] = 0.0;
        proc->internalData_SolveTransposeUsingLU.tagInternal[row] = proc->
            internalData_SolveTransposeUsingLU.currtagInternal;
        addtree(&tree, row);
      }
      proc->internalData_SolveTransposeUsingLU.yInternal[row] -= proc->
          factoredInverse.LT.values[k] * beta;
    }
  }

  y->clear();
  for (i = getfirst(&tree); i != -1; i = getnext(&tree)) {
    if (ABS(proc->internalData_SolveTransposeUsingLU.yInternal[i]) >
        SOLVE_USING_LU_EPS) {
      y->append(proc->internalData_SolveTransposeUsingLU.yInternal[i], proc->
          factoredInverse.rowPerm.at(i));
    }
  }

  proc->internalData_SolveTransposeUsingLU.currtagInternal++;
  killtree(&tree);

  endtime = (double) clock();
  proc->cumtime += endtime - starttime;

  return consistent;
}

/*
static void iterationProcess_lu_clo(IterationProcess * proc) {
  freeIV( &proc->factoredInverse.rowPerm );
  freeIV( &proc->factoredInverse.invRowPerm );
  freeIV( &proc->factoredInverse.colPerm );
  freeIV( &proc->factoredInverse.invColPerm );
```

```
  FREE( proc->factoredInverse.L.values );
  FREE( proc->factoredInverse.L.rowIndices );
  FREE( proc->factoredInverse.L.colPos );
  FREE( proc->factoredInverse.UT.values );
  FREE( proc->factoredInverse.UT.rowIndices );
  FREE( proc->factoredInverse.UT.colPos );
  FREE( proc->factoredInverse.diagU );
  freeIV( &proc->etaFile.newColLocations );
  freeTV( &proc->etaFile.values );
  freeIV( &proc->etaFile.rowIndices );
  freeIV( &proc->etaFile.colPos );
}
*/

/*-------------------------------------------------------------+
 | Forward/backward solve using LU factorization               |
 | Input:                                                      |
 |    m          dimension of array y                          |
 |    y          array containing right-hand side              |
 |                                                             |
 |    static global variables (assumed setup by lufac()):     |
 |                                                             |
 |    rank       rank of B                                     |
 |    L.colPos, L.rowIndices, L.values, three array sparse representation of L
    .values             |
 |    UT.colPos,UT.rowIndices,UT.values three array sparse representation of U
     transpose   |
 |               without its diagonal                          |
 |    diagU      diagonal entries of U                         |
 |    colPerm, invColPerm, rowPerm, invRowPerm                     |
 |               column and row permutations and their inverses |
 | Output:                                                     |
 |                                          -1                 |
 |    y          array containing solution B  y                |
 |                                                             |
 |    integer flag indicating whether system is consistent    */

static int iterationProcess_dbsolve(IterationProcess * proc, SafeVector<TYPE>
   & y) {
  int i;
  int k, row, consistent = TRUE;
  TYPE beta, *dwork;
  TYPE eps;
  int m = y.size();

  double starttime, endtime;
```

```
starttime = (double) clock();

MALLOC (dwork,m,TYPE);

if (proc->factoredInverse.rank < m)
  eps = EPSSOL * maxv(y);
for (i = 0; i < m; i++)
  dwork[i] = y[i];
for (i = 0; i < m; i++)
  y[proc->factoredInverse.invRowPerm.at(i)] = dwork[i];
/*-------------------------------------------------+
 |               -1                                |
 |       y  <-  L  y                               */

for (i = 0; i < proc->factoredInverse.rank; i++) {
  beta = y[i];
  for (k = proc->factoredInverse.L.colPos[i]; k < proc->factoredInverse.L.
      colPos[i + 1]; k++) {
    row = proc->factoredInverse.L.rowIndices[k];
    y[row] -= proc->factoredInverse.L.values[k] * beta;
  }
}

/*-------------------------------------------------+
 |               -1                                |
 |       y  <-  U  y                               */

for (i = m - 1; i >= proc->factoredInverse.rank; i--) {
  if (ABS( y[i] ) > eps)
    consistent = FALSE;
  y[i] = 0.0;
}

for (i = proc->factoredInverse.rank - 1; i >= 0; i--) {
  beta = y[i];
  for (k = proc->factoredInverse.UT.colPos[i]; k < proc->factoredInverse.UT.
      colPos[i + 1]; k++) {
    beta -= proc->factoredInverse.UT.values[k] * y[proc->factoredInverse.UT.
        rowIndices[k]];
  }
  y[i] = beta / proc->factoredInverse.diagU[i];
}

for (i = 0; i < m; i++)
  dwork[i] = y[i];
for (i = 0; i < m; i++)
  y[proc->factoredInverse.colPerm.at(i)] = dwork[i];
```

```
  FREE(dwork);

  endtime = (double) clock();
  proc->cumtime += endtime - starttime;

  return consistent;
}

static void iterationProcess_Gauss_Eta(IterationProcess * proc,
   SafeSparseVector & dx_B) {
  int i, j, k, col, kcol, ii;
  TYPE temp;

  if (proc->etaFile.iter <= 0) {
    printp("Gauss_Eta is returning due to empty eta file");
    return;
  }

  ii = -1;
  for (k = 0; k < dx_B.numNonzeroes(); k++) {
    i = dx_B.rowIndex(k);
    proc->internalData_Gauss_Eta.a[i] = dx_B.value(k);
    proc->internalData_Gauss_Eta.tag[i] = proc->internalData_Gauss_Eta.currtag
       ;
    proc->internalData_Gauss_Eta.link[ii] = i;
    ii = i;
  }
  printp("Gauss_Eta is reading eta file");
  for (j = 0; j < proc->etaFile.iter; j++) {
    assert(proc->etaFile.colPos.at(j) < proc->etaFile.colPos.at(j + 1));
    col = proc->etaFile.newColLocations.at(j);

    for (k = proc->etaFile.colPos.at(j); k < proc->etaFile.colPos.at(j + 1); k
       ++) {
      i = proc->etaFile.rowIndices.at(k);
      if (proc->internalData_Gauss_Eta.tag[i] != proc->internalData_Gauss_Eta.
         currtag) {
        proc->internalData_Gauss_Eta.a[i] = 0.0;
        proc->internalData_Gauss_Eta.tag[i] = proc->internalData_Gauss_Eta.
           currtag;
        proc->internalData_Gauss_Eta.link[ii] = i;
        ii = i;
      }
      if (i == col)
        kcol = k;
    }
```

```
    temp = proc->internalData_Gauss_Eta.a[col] / proc->etaFile.values.at(kcol)
        ;
    if (temp != 0.0) {
      for (k = proc->etaFile.colPos.at(j); k < kcol; k++) {
        i = proc->etaFile.rowIndices.at(k);
        proc->internalData_Gauss_Eta.a[i] -= proc->etaFile.values.at(k) * temp
            ;
      }
      proc->internalData_Gauss_Eta.a[col] = temp;
      for (k = kcol + 1; k < proc->etaFile.colPos.at(j + 1); k++) {
        i = proc->etaFile.rowIndices.at(k);
        proc->internalData_Gauss_Eta.a[i] -= proc->etaFile.values.at(k) * temp
            ;
      }
    }
  }
  proc->internalData_Gauss_Eta.link[ii] = dx_B.numRows();
  proc->internalData_Gauss_Eta.currtag++;

  dx_B.clear();
  for (i = proc->internalData_Gauss_Eta.link[-1]; i < dx_B.numRows(); i = proc
      ->internalData_Gauss_Eta.link[i]) {
    if (ABS(proc->internalData_Gauss_Eta.a[i]) > GAUSS_ETA_EPS) {
      dx_B.append(proc->internalData_Gauss_Eta.a[i], i);
    }
  }
}

/*************************************************************
 *  Gaussian elimination for Eta transformations which will solve *
 * each system B y = c for y                                  *
 *************************************************************/

static void iterationProcess_Gauss_Eta_T(IterationProcess * proc,
    SafeSparseVector & vec) {
  int i = -1, j, k, kk = -1, kkk = -1, col;
  TYPE temp;

  //printp("Gauss_Eta_T is reading eta file");
  for (j = proc->etaFile.iter - 1; j >= 0; j--) {
    col = proc->etaFile.newColLocations.at(j);

    for (k = 0; k < vec.numNonzeroes(); k++) {
      i = vec.rowIndex(k);
      if (i == col)
        kk = k;
```

```
      proc->internalData_Gauss_Eta_T.a[i] = vec.value(k);
      proc->internalData_Gauss_Eta_T.tag[i] = proc->internalData_Gauss_Eta_T.
          currtag;
    }

    if (proc->internalData_Gauss_Eta_T.tag[col] != proc->
        internalData_Gauss_Eta_T.currtag) {
      kk = vec.numNonzeroes();
      vec.append(0.0, col);
      proc->internalData_Gauss_Eta_T.a[col] = 0.0;
      proc->internalData_Gauss_Eta_T.tag[col] = proc->internalData_Gauss_Eta_T
          .currtag;
    }
    temp = vec.value(kk);
    for (k = proc->etaFile.colPos.at(j); k < proc->etaFile.colPos.at(j + 1); k
        ++) {
      i = proc->etaFile.rowIndices.at(k);
      if (i == col)
        kkk = k;
      if (proc->internalData_Gauss_Eta_T.tag[i] == proc->
          internalData_Gauss_Eta_T.currtag) {
        if (i != col) {
          temp -= proc->etaFile.values.at(k) * proc->internalData_Gauss_Eta_T.
              a[i];
        }
      }
    }
    proc->internalData_Gauss_Eta_T.currtag++;

    vec.value(kk) = temp / proc->etaFile.values.at(kkk);
  }
}

static void iterationProcess_Nt_times_y(IterationProcess * proc, SparseMatrix
    AT, const SafeVector<int> & basicflag, const SafeSparseVector & y,
    SafeSparseVector & yN) {
  int i, j, jj, k, kk;

  jj = -1;
  for (k = 0; k < y.numNonzeroes(); k++) {
    i = y.rowIndex(k);
    for (kk = AT.colPos[i]; kk < AT.colPos[i + 1]; kk++) {
      j = AT.rowIndices[kk];
      if (basicflag.at(j) < 0) {
        if (proc->internalData_Nt_times_y.tag[j] != proc->
            internalData_Nt_times_y.currtag) {
          proc->internalData_Nt_times_y.a[j] = 0.0;
```

```
          proc->internalData_Nt_times_y.tag[j] = proc->internalData_Nt_times_y
              .currtag;
          proc->internalData_Nt_times_y.link[jj] = j;
          jj = j;
        }
        proc->internalData_Nt_times_y.a[j] += y.value(k) * AT.values[kk];
      }
    }
  }
  proc->internalData_Nt_times_y.link[jj] = AT.rows;
  proc->internalData_Nt_times_y.currtag++;

  yN.clear();
  for (jj = proc->internalData_Nt_times_y.link[-1]; jj < AT.rows; jj = proc->
      internalData_Nt_times_y.link[jj]) {
    if (ABS(proc->internalData_Nt_times_y.a[jj]) > EPS_MATRIX) {
      yN.append(proc->internalData_Nt_times_y.a[jj], columnForVariableP(jj));
    }
  }
}

/*************************************************************
 * STEP 3: Ratio test to find leaving column                 *
 *************************************************************/
static int CHUZR1_findLeaving_ratioTest(IterationProcess * proc, const
    SafeSparseVector & dy, const SafeVector<TYPE> & y) {
  int j, jj = -1, k;
  TYPE min = HUGE_VAL;
  //printp("ratio test");
  //printRawSparseVector(dy);
  for (k = 0; k < dy.numNonzeroes(); k++) {
    if (dy.value(k) > RATIO_TEST_ETA) {
      j = dy.rowIndex(k);
      if (y[j] / dy.value(k) < min) {
        min = y[j] / dy.value(k);
        jj = j;
      }
    }
  }
  return jj;
}

static TYPE iterationProcess_recalculateObjectiveValues(const SafeVector<TYPE>
    & c, const SafeVector<TYPE> & x_B, const SafeVector<int> & basics) {
  int i;
  TYPE prod = 0.0;
```

```
  for (i = 0; i < x_B.size(); i++) {
    prod += c[basics[i]] * x_B[i];
  }

  return prod;
}

static int compareAttractiveCandidates(const void * a, const void * b) {
  const AttractiveCandidate * candA = (const AttractiveCandidate *)a;
  const AttractiveCandidate * candB = (const AttractiveCandidate *)b;
  if (candA->value < candB->value)
    return -1;
  else if (candA->value > candB->value)
    return 1;
  else
    return 0;
}

/*
    ***************************************************************************

 * STEP 1: Pick most negative nonbasic (primal or dual, depending on the
     parameters) *
 *         'attractive' is assumed to be initialised.
 ***************************************************************************
     */
static void CHUZC_findAttractiveCandidates(const SafeVector<int> &
    columnToVariableMapping, const SafeVector<TYPE> & x,
    AttractiveCandidateVector * attractive) {
  int i;
  TYPE best = x[0];
  attractive->size = 0;
  for (i = 0; i < x.size(); i++) {
    if (x[i] < best) {
      best = x[i];
    }
    if (x[i] < -PICK_NEG_EPS) {
      AttractiveCandidate cand;
      cand.var = columnToVariableMapping[i];
      cand.value = x[i];
      appendACV(attractive, cand);
    }
  }
  //printf("best candidate: %e\n", best);fflush(stdout);
  qsort(attractive->array, attractive->size, sizeof(AttractiveCandidate),
      compareAttractiveCandidates);
}
```

simpo/payloadtypes.h

```
#ifndef PAYLOADTYPES_H
#define PAYLOADTYPES_H

#include "../common/genericvectors.h"
#include "sparse.h"
#include <vector>

typedef struct {
  int leavingRow;
  int enteringCol;
} InvProcPivotPayload;

typedef struct {
  int leavingVar;
  int enteringVar;
  int basisNumber;
} ColSelPivotPayload;

typedef struct {
  int basisNumber;
  std::vector<int> attractiveVariables;
} AttractiveCandidatesMessage;

typedef struct {
  int var;
  int basisNumber;
} CandidateIsUnattractivePayload;

typedef struct {
  int basisNumber;
} ChuzrOfferPayload;

typedef struct {
  int accepted;
} ChuzrAcceptancePayload;

typedef struct {
  int leavingRow;
  int enteringCol;
  int fromBasis;
  SafeSparseVector primalStepDirection;
  SafeSparseVector dualStepDirection;
  TYPE primalStepLength;
  TYPE dualStepLength;
  std::vector<TYPE> newEtaFileValues;
```

```
  std::vector<int> newEtaFileRowIndices;
  int etaFileSizeDiff;
} BasisChangePayload;

#endif
```

simpo/print.h and simpo/print.c

```
#ifndef PRINT_H
#define PRINT_H

#include <stdio.h>
#include <stdarg.h>

void initPrintFromProcess();
void printFromProcess(const char * processName, const char * formatString,
    ...);

#endif
```

```
#include "print.h"
#include <pthread.h>
#include <stdarg.h>

static pthread_mutex_t printLock;

void initPrintFromProcess() {
  pthread_mutex_init(&printLock, NULL);
}

// If processName is NULL, no prefix will be printed
void printFromProcess(const char * processName, const char * formatString,
    ...) {
  va_list args;
  pthread_mutex_lock(&printLock);
  if (processName)
    printf("%s: ", processName);
  va_start(args, formatString);
  vprintf(formatString, args);
  va_end(args);
  printf("\n");
  fflush(stdout);
  pthread_mutex_unlock(&printLock);
}
```

simpo/sparse.h and simpo/sparse.c

```
#ifndef SPARSE_H
#define SPARSE_H

#include <string.h>
#include <vector>

typedef struct {
  int rows;
  int cols;
  int numNonzeroes;
  int * rowIndices;
  int * colPos;
  TYPE * values;
} SparseMatrix;

typedef struct {
  int rows;
  int numNonzeroes;
  int * rowIndices;
  TYPE * values;
} SparseVector;

class SafeSparseVector {
private:
  int rows;
  std::vector<int> rowIndices;
  std::vector<TYPE> values;
public:
  int numNonzeroes() const {
    return values.size();
  }
  int numRows() const {
    return rows;
  }
  void resize(int size) {
    rowIndices.resize(size);
    values.resize(size);
  }
  void setRows(int rows) {
    this->rows = rows;
  }
  void clear() {
    rowIndices.clear();
    values.clear();
  }
```

```
  void append(TYPE value, int rowIndex) {
    values.push_back(value);
    rowIndices.push_back(rowIndex);
  }
  TYPE & value(int i) {
    return values.at(i);
  }
  TYPE value(int i) const {
      return values.at(i);
  }
  int & rowIndex(int i) {
    return rowIndices.at(i);
  }
  int rowIndex(int i) const {
    return rowIndices.at(i);
  }
  TYPE maxValue() const;
};

void nullifySparseMatrix(SparseMatrix * matrix);
void makeZeroSparseMatrix(SparseMatrix * matrix, int rows, int cols);
void allocateAndCopyArray(const void * source, void ** destination, int
   numElements, size_t elementSize);
void copySparseMatrix(SparseMatrix source, SparseMatrix * destination);
void copySparseVector(SparseVector source, SparseVector * destination);
void nullifySparseVector(SparseVector * vector);
void printRawSparseMatrix(SparseMatrix matrix);
void printRawSparseVector(SparseVector vector);
void printTypeArray(TYPE * array, int size);
void printIntArray(int * array, int size);

//void initSparseMatrix(SparseMatrix * matrix, int rows, int cols);
void initSparseVector(SparseVector * vector, int rows);
void freeSparseMatrix(SparseMatrix * matrix);
void freeSparseVector(SparseVector * vector);

template<typename T>
class SafeVector : public std::vector<T> {
public:
  T & operator [] (int index) {
    return this->at(index);
  }
  T operator [] (int index) const {
    return this->at(index);
  }
};
```

```
#endif
```

```
#include "sparse.h"
#include "../common/linalg.h"
#include <stdlib.h>
#include <stdio.h>

void nullifySparseMatrix(SparseMatrix * matrix) {
  matrix->rows = 0;
  matrix->cols = 0;
  matrix->numNonzeroes = 0;
  matrix->rowIndices = NULL;
  matrix->colPos = NULL;
  matrix->values = NULL;
}

void makeZeroSparseMatrix(SparseMatrix * matrix, int rows, int cols) {
  matrix->rows = rows;
  matrix->cols = cols;
  matrix->numNonzeroes = 0;
  matrix->rowIndices = (int*)malloc(0);
  matrix->values = (TYPE*)malloc(0);
  matrix->colPos = (int*)malloc((cols + 1) * sizeof(int));
  memset(matrix->colPos, 0, (cols + 1) * sizeof(int));
}

void nullifySparseVector(SparseVector * vector) {
  vector->rows = 0;
  vector->numNonzeroes = 0;
  vector->rowIndices = NULL;
  vector->values = NULL;
}

// does not free destination
void allocateAndCopyArray(const void * source, void ** destination, int
    numElements, size_t elementSize) {
  *destination = malloc(numElements * elementSize);
  memcpy(*destination, source, numElements * elementSize);
}

void copySparseMatrix(SparseMatrix source, SparseMatrix * destination) {
  destination->rows = source.rows;
  destination->cols = source.cols;
  destination->numNonzeroes = source.numNonzeroes;
  allocateAndCopyArray(source.rowIndices, (void**)&destination->rowIndices,
      source.numNonzeroes, sizeof(int)); //TODO: Are we guaranteed this size?
  allocateAndCopyArray(source.colPos, (void**)&destination->colPos, (source.
```

```
    cols + 1), sizeof(int));
  allocateAndCopyArray(source.values, (void**)&destination->values, source.
    numNonzeroes, sizeof(TYPE));
}

void copySparseVector(SparseVector source, SparseVector * destination) {
  destination->rows = source.rows;
  destination->numNonzeroes = source.numNonzeroes;
  //TODO: Allocate 'rows', but only copy 'numNonzeroes'
  allocateAndCopyArray(source.rowIndices, (void**)&destination->rowIndices,
    source.rows, sizeof(int)); //TODO: Are we guaranteed this size?
  allocateAndCopyArray(source.values, (void**)&destination->values, source.
    rows, sizeof(TYPE));
}

void printRawSparseMatrix(SparseMatrix matrix) {
  int i;
  fflush(stdout);
  printf("%d x %d (%d):\n", matrix.rows, matrix.cols, matrix.numNonzeroes);
  for (i = 0; i < matrix.cols + 1; ++i)
    printf("%d ", matrix.colPos[i]);
  printf("\n");
  for (i = 0; i < matrix.numNonzeroes; ++i)
    printf("%d:%e ", matrix.rowIndices[i], matrix.values[i]);
  printf("\n");
  fflush(stdout);
}

void printRawSparseVector(SparseVector vector) {
  int i;
  fflush(stdout);
  printf("%d (%d):", vector.rows, vector.numNonzeroes);
  for (i = 0; i < vector.numNonzeroes; ++i)
    printf(" %d:%e", vector.rowIndices[i], vector.values[i]);
  printf("\n");
  fflush(stdout);
}

void printTypeArray(TYPE * array, int size) {
  int i;
  fflush(stdout);
  printf("%d:", size);
  for (i = 0; i < size; ++i) {
    printf(" %e", array[i]);
  }
  printf("\n");
  fflush(stdout);
```

```
}

void printIntArray(int * array, int size) {
  int i;
  fflush(stdout);
  printf("%d:", size);
  for (i = 0; i < size; ++i) {
    printf(" %d", array[i]);
  }
  printf("\n");
  fflush(stdout);
}

/*
void initSparseMatrix(SparseMatrix * matrix, int rows, int cols) {
  matrix->rows = rows;
  matrix->cols = cols;
  matrix->numNonzeroes = 0;
  matrix->rowIndices = NULL;
  matrix->colPos = NULL;
  matrix->values = NULL;
}
*/

void initSparseVector(SparseVector * vector, int rows) {
  vector->numNonzeroes = 0;
  vector->rowIndices = (int*)malloc(sizeof(int) * rows);
  vector->rows = rows;
  vector->values = (TYPE*)malloc(sizeof(TYPE) * rows);
}


void freeSparseMatrix(SparseMatrix * matrix) {
  free(matrix->colPos);
  free(matrix->rowIndices);
  free(matrix->values);
  nullifySparseMatrix(matrix);
}

void freeSparseVector(SparseVector * vector) {
  free(vector->rowIndices);
  free(vector->values);
  nullifySparseVector(vector);
}

TYPE SafeSparseVector::maxValue() const {
  return maxv(values);
```

```
}
```

A.4 ASYNPLEX for Cell/BE, based on Vanderbei

We only list a few files that highlight the inner workings of our CML RPC-based communication systems. Most of the other files are very similar to the x86 version. The complete source code can be found in the attachment.

common/message.h

Message structure for RPC messages (contrasted to the old x86 message structure).

```
#ifndef MESSAGE_H
#define MESSAGE_H

typedef struct {
  const char * sender;
  const char * receiver;
  const char * tag;
  void * payload;
} Message;

typedef struct {
  int sender;     // Rank / SPE index (0-7) of sender
  int receiver;   // Rank / SPE index (0-7) of receiver
  int tag;      // Arbitrary nonnegative integer (not limited like CML's tags)
  int payloadSize;  // Size (in bytes) of 'payload' buffer
  void * payload;   // The entire RPC buffer - expected to contain FIRST
     sender+receiver+tag+payloadSize, and THEN the actual message payload
} RpcMessage;

#endif
```

common/SafeVector.h

For debugging purposes, we override the `[]` operator of the `std::vector` to perform bounds checking. This can be removed when development is finished.

```
#ifndef SAFEVECTOR_H
#define SAFEVECTOR_H

#include <vector>

template<typename T>
class SafeVector : public std::vector<T> {
public:
```

```
  T & operator [] (int index) {
    return this->at(index);
  }
  T operator [] (int index) const {
    return this->at(index);
  }
};

#endif
```

PPU/asynplexcontrol.h and PPU/asynplexcontrol.cpp

Provides RPC functions to the SPEs, and initiates the MPI programs on the SPEs.

```
#ifndef ASYNPLEXCONTROL_H
#define ASYNPLEXCONTROL_H

#include "common/sparse.h"
#include "invertprocessor.h"
#include <vector>
#include <pthread.h>
extern "C" {
  #include <cellmsg.h>
}

class AsynplexControl {
public:
  void runAsynplex(int numIterationProcesses, int m, int n, SparseMatrix & A,
     TYPE * b, TYPE * c, TYPE f);
  void waitForCompletion();
private:
  static void getDimensions(cellmsg_rpc_data * rpc);
  static void getIntArrays(cellmsg_rpc_data * rpc);
  static void getTypeArrays(cellmsg_rpc_data * rpc);
  int m;
  int n;
  int numNonzeroes; // in A
  int intBytes;
  int typeBytes;
  int * intArrays;
  TYPE * typeArrays;
  pthread_t invertProcessor;
  int numIterationProcesses;
};

#endif
```

```
#include "asynplexcontrol.h"
#include "print.h"
#include "communication.h"
#include <malloc_align.h>
#include <free_align.h>
#include "common/types.h"

extern spe_program_handle_t speProgramHandle;

static AsynplexControl * instance; // Hack to give the static RPC functions
    access to the AsynplexControl object (of which there is only one)

// Input: none
// Output: A two element int array, containing m and n (will not be freed, but
     we can tolerate an eight byte leak; this function is only called once)
void AsynplexControl::getDimensions(cellmsg_rpc_data * rpc) {
  rpc->buffer = _malloc_align(4 * sizeof(int), ALIGN_BUS_LOG2);
  ((int*)rpc->buffer)[0] = instance->m;
  ((int*)rpc->buffer)[1] = instance->n;
  ((int*)rpc->buffer)[2] = instance->numNonzeroes;
  rpc->numbytes = 4 * sizeof(int);
}

void AsynplexControl::getIntArrays(cellmsg_rpc_data * rpc) {
  rpc->buffer = instance->intArrays;
  rpc->numbytes = instance->intBytes;
}

void AsynplexControl::getTypeArrays(cellmsg_rpc_data * rpc) {
  rpc->buffer = instance->typeArrays;
  rpc->numbytes = instance->typeBytes;
}

void AsynplexControl::runAsynplex(int numIterationProcesses, int m, int n,
    SparseMatrix & A, TYPE * b, TYPE * c, TYPE f) {
  instance = this;
  this->m = m;
  this->n = n;
  int N = m + n;
  this->numNonzeroes = A.numNonzeroes;
  this->numIterationProcesses = numIterationProcesses;
  int intCount = numNonzeroes + (N + 1);    // A.rowIndices, A.colPos
  int typeCount = 1 + m + N + numNonzeroes; // f, b, c, A.values
  this->intBytes = ROUND_UP_MULTIPLE(intCount * sizeof(int), ALIGN_BUS_WIDTH);
  this->typeBytes = ROUND_UP_MULTIPLE(typeCount * sizeof(TYPE),
     ALIGN_BUS_WIDTH);
```

```
  this->intArrays = (int*)_malloc_align(intBytes, ALIGN_BUS_LOG2);
  this->typeArrays = (TYPE*)_malloc_align(typeBytes, ALIGN_BUS_LOG2);
  memcpy(this->intArrays, A.rowIndices, numNonzeroes * sizeof(int));
  memcpy(this->intArrays + numNonzeroes, A.colPos, (N + 1) * sizeof(int));
  this->typeArrays[0] = f;
  memcpy(this->typeArrays + 1, b, m * sizeof(TYPE));
  memcpy(this->typeArrays + 1 + m, c, N * sizeof(TYPE));
  memcpy(this->typeArrays + 1 + m + N, A.values, numNonzeroes * sizeof(TYPE));
  initPrintFromProcess();
  initCommunication();
  setupInvertProcessor(A, numIterationProcesses);
  pthread_create(&invertProcessor, NULL, runInvertProcessor, NULL);

  printf("m: %d, n: %d, nz: %d\n", m, n, A.numNonzeroes);fflush(stdout);
  cellmsg_provide_rpc(sendRpcMessage);
  cellmsg_provide_rpc(sendRpcMessageBatch);
  cellmsg_provide_rpc(receiveRpcMessage);
  cellmsg_provide_rpc(getDimensions);
  cellmsg_provide_rpc(getIntArrays);
  cellmsg_provide_rpc(getTypeArrays);
  cellmsg_init(0, NULL);
  cellmsg_run(&speProgramHandle, 0, NULL); // Start the SPE program running on
      all SPEs and wait until all SPEs invoke MPI_Finalize()
  cellmsg_finalize();
}

void AsynplexControl::waitForCompletion() {
  pthread_join(invertProcessor, NULL);
}
```

PPU/communication.h and PPU/communication.cpp

Our CML RPC-based messaging system, which we needed to develop because CML does not support asynchronous MPI message primitives. These functions will be invoked by the SPEs through the RPC functionality of CML.

```
#ifndef COMMUNICATION_H
#define COMMUNICATION_H

#include "../common/message.h"
extern "C" {
  #include <cellmsg.h>
}

void initCommunication();
void sendRpcMessage(cellmsg_rpc_data * rpc);
void sendRpcMessageBatch(cellmsg_rpc_data * rpc);
```

```
void receiveRpcMessage(cellmsg_rpc_data * rpc);
void printMessages();

#endif
```

```
#include "../common/message.h"
#include "../common/sparse.h"
#include "communication.h"
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <malloc_align.h>
#include <free_align.h>
#include "print.h"
#include "common/SafeVector.h"
#include "common/tags.h"
#include "common/types.h"

Message createMessage(const char * sender, const char * receiver, const char *
    tag, void * payload) {
  Message message;
  message.sender = sender;
  message.receiver = receiver;
  message.tag = tag;
  message.payload = payload;
  return message;
}

static SafeVector<RpcMessage> messages;
static pthread_mutex_t queueLock;
static int printSends = 1;
static int printReceives = 1;

void initCommunication() {
  pthread_mutex_init(&queueLock, NULL);
}

void sendMessage(RpcMessage message) {
  pthread_mutex_lock(&queueLock);
  messages.push_back(message);
  pthread_mutex_unlock(&queueLock);
  if (printSends) {
    printFromProcess("COMM", "%d sends to %d (tag: %d, payload %p) (queue: %d)
        ", message.sender, message.receiver, message.tag, message.payload,
        messages.size());
  }
```

```
}
/*
void sendAll(Message * messagesToBeSent, int numMessages) {
  int i;
  pthread_mutex_lock(&queueLock);
  for (i = 0; i < numMessages; ++i) {
    messages.push_back(messagesToBeSent[i]);
    if (printSends) {
      printFromProcess("COMM", "%s sends to %s (tag: '%s', payload %p) (queue:
           %d)", messagesToBeSent[i].sender, messagesToBeSent[i].receiver,
          messagesToBeSent[i].tag, messagesToBeSent[i].payload, messages.size
          ());
    }
  }
  pthread_mutex_unlock(&queueLock);
}
*/
// sender < 0 means "receive from any iteration process"
// tag < 0 means "any tag"
RpcMessage receiveMessageWithTag(int receiver, int sender, int tag) {
  RpcMessage msg;
  int i;
  pthread_mutex_lock(&queueLock);
  for (i = 0; i < messages.size(); ++i) {
    msg = messages[i];
    if (msg.receiver == receiver && ((sender < 0 && msg.sender >=
        RANK_OF_FIRST_ITER_PROC) || msg.sender == sender) && (tag < 0 || msg.
        tag == tag)) {
      messages.erase(messages.begin() + i);
      if (printReceives) {
        printFromProcess("COMM", "%d receives from %d (tag: %d, payload: %p) (
            queue: %d)", msg.receiver, msg.sender, msg.tag, msg.payload,
            messages.size());
      }
      pthread_mutex_unlock(&queueLock);
      return msg;
    }
  }
  pthread_mutex_unlock(&queueLock);
  msg.sender = -1;
  return msg;
}

void printMessages() {
  int i;
  pthread_mutex_lock(&queueLock);
  printFromProcess("COMM", "Remaining messages:");
```

```
  for (i = 0; i < messages.size(); ++i) {
    printFromProcess("COMM", "%s %s %s", messages[i].sender, messages[i].
        receiver, messages[i].tag);
  }
  fflush(stdout);
  pthread_mutex_unlock(&queueLock);
}

void sendRpcMessage(cellmsg_rpc_data * rpc) {
  RpcMessage message;
  int * buffer = (int*)rpc->buffer;
  message.sender = buffer[0];
  message.receiver = buffer[1];
  message.tag = buffer[2];
  message.payloadSize = buffer[3];
  assert(message.payloadSize <= rpc->numbytes);
  message.payload = _malloc_align(message.payloadSize, ALIGN_BUS_LOG2);
  memcpy(message.payload, buffer, message.payloadSize);
  sendMessage(message);
  rpc->numbytes = 0; // No RPC output
}

void sendRpcMessageBatch(cellmsg_rpc_data * rpc) { //TODO
  assert(0);
}

//static void * previouslyReceivedBuffer = NULL;

void receiveRpcMessage(cellmsg_rpc_data * rpc) {
  int * buffer = (int*)rpc->buffer;
  int sender = buffer[0];
  int receiver = buffer[1];
  int tag = buffer[2];
  RpcMessage message = receiveMessageWithTag(receiver, sender, tag);
  if (message.sender < 0) { // No suitable message was found
    rpc->numbytes = sizeof(int);
    buffer[0] = -1; // Sender < 0 signifies no message
  }
  else {
    rpc->buffer = message.payload; //TODO: Must be freed somewhere
    rpc->numbytes = message.payloadSize;
  }
}
```

SPU/BasisChangeManager.h and SPU/BasisChangeManager.cpp

In order to conserve space, we only show the code for the simplest ASYNPLEX process here, to demonstrate how the message system is used from the SPEs.

```
#ifndef BASISCHANGEMANAGER_H
#define BASISCHANGEMANAGER_H

class BasisChangeManager {
public:
  void run();
private:
  int currentBasis;
};

#endif
```

```
#include "BasisChangeManager.h"
//#include "invertprocessor.h"
//#include "print.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <libmisc.h>
//#define print(formatString, ...) printFromProcess(NAME, formatString , ##
   __VA_ARGS__)
#include "rpc.h"
#include "common/tags.h"

#define isProgramFinished() false
#define RANK RANK_BC

void BasisChangeManager::run() {
  /*int * buffer = (int*)malloc_align(8 * sizeof(int), ALIGN_BUS_LOG2);
  do {
    printf("trying receive\n");fflush(stdout);
    buffer[0] = ANY_SENDER; // Sender
    buffer[1] = RANK; // Receiver
    buffer[2] = ANY_TAG; // Tag
    cellmsg_rpc(receiveRpcMessage,
          buffer, 4 * sizeof(int), CML_BYTE_SWAP_NOT_NEEDED,
          buffer, 8 * sizeof(int), CML_BYTE_SWAP_NOT_NEEDED);
  } while (buffer[0] < 0);
  printf("Received %d bytes from %d: tag %d, payload %d\n", buffer[3], buffer
     [0], buffer[2], buffer[4]);*/
  Communication comm;
  comm.setOwner(BASIS_MGR);
```

```
  RpcMessage message;
  const char * result;
  int offerReply;
  int incomingBasis;
  message.payload = &incomingBasis;
  currentBasis = 0;
  printf("R: Basis change manager starting\n");fflush(stdout);
  while (!isProgramFinished()) {
    while (!comm.receiveCopy(ANY_ITER_PROC, I5_R1, message, sizeof(int))) {
      if (isProgramFinished()) {
        printf("R: Basis change manager exiting\n");fflush(stdout);
        return;
      }
    }
    if (incomingBasis == currentBasis) {
      offerReply = 1;
      comm.sendCopy(message.sender, R2_I6, &offerReply, sizeof(int));
      ++currentBasis;
      result = "accepted";
    }
    else {
      offerReply = 0;
      comm.sendCopy(message.sender, R3_I6, &offerReply, sizeof(int));
      result = "refused";
    }
    printf("R: I%d offers chuzr on basis %d... %s\n", message.sender,
        incomingBasis, result);fflush(stdout);
  }
  printf("R: Basis change manager exiting\n");fflush(stdout);
}
```

SPU/rpc.h and SPU/rpc.cpp

Convenience functions for RPC message passing on the SPEs.

```
#ifndef RPC_H
#define RPC_H

#include <spu_intrinsics.h> // Need to include this one before mpi.h, because
    otherwise extern "C" will screw up the files that mpi.h includes (might
    need to include even more of those files here if we start using them)
extern "C" {
  #include <mpi.h>
}
#include "common/message.h"

#define ALIGNED16  __attribute__((aligned(16)))
```

```
#define ALIGN_BUS_LOG2 7
#define ALIGN_BUS_WIDTH 128

// Indices into the int buffer used for transmitting messages
#define SENDER_OFFSET 0
#define RECEIVER_OFFSET 1
#define TAG_OFFSET 2
#define BUFFER_SIZE_OFFSET 3
#define PAYLOAD_OFFSET 4

// Give files that include this one access to the function pointers defined in
    main.cpp
extern ppe_funcptr sendRpcMessage ALIGNED16;
extern ppe_funcptr sendRpcMessageBatch ALIGNED16;
extern ppe_funcptr receiveRpcMessage ALIGNED16;

class Communication {
private:
  int owner;
public:
  void setOwner(int owner) { this->owner = owner; }
  void sendCopy(int receiver, int tag, void * data, int dataSize);
  void sendBuffer(int receiver, int tag, void * buffer, int totalSize);
  bool receiveCopy(int sender, int tag, RpcMessage & message, int payloadSize)
     ;
};

#endif
```

```
#include "rpc.h"
#include <libmisc.h>
#include "common/types.h"
#include <assert.h>
#include <string.h>

void Communication::sendCopy(int receiver, int tag, void * payload, int
   payloadSize) {
  int bufferSize = payloadSize + 4 * sizeof(int);
  int roundedBufferSize = ROUND_UP_MULTIPLE(bufferSize, ALIGN_BUS_WIDTH);
  int * buffer = (int*)malloc_align(roundedBufferSize, ALIGN_BUS_LOG2);
  buffer[SENDER_OFFSET] = this->owner;
  buffer[RECEIVER_OFFSET] = receiver;
  buffer[TAG_OFFSET] = tag;
  buffer[BUFFER_SIZE_OFFSET] = bufferSize;
  memcpy(buffer + PAYLOAD_OFFSET, payload, payloadSize);
  printf("sendCopy: %d %d %d, %d %d %d %p\n", buffer[0], buffer[1], buffer[2],
      payloadSize, bufferSize, roundedBufferSize, buffer);
```

```
  cellmsg_rpc(sendRpcMessage,
        buffer, roundedBufferSize, CML_BYTE_SWAP_NOT_NEEDED,
        NULL, 0, CML_BYTE_SWAP_NOT_NEEDED);
  free_align(buffer);
}

bool Communication::receiveCopy(int sender, int tag, RpcMessage & message, int
    payloadSize) {
  int bufferSize = payloadSize + 4 * sizeof(int);
  int roundedBufferSize = ROUND_UP_MULTIPLE(bufferSize, ALIGN_BUS_WIDTH);
  int * buffer = (int*)malloc_align(roundedBufferSize, ALIGN_BUS_LOG2);
  buffer[SENDER_OFFSET] = sender;
  buffer[RECEIVER_OFFSET] = this->owner;
  buffer[TAG_OFFSET] = tag;
  cellmsg_rpc(receiveRpcMessage,
        buffer, 4 * sizeof(int), CML_BYTE_SWAP_NOT_NEEDED,
        buffer, roundedBufferSize, CML_BYTE_SWAP_NOT_NEEDED);
  bool success = buffer[SENDER_OFFSET] >= 0; // receiveRpcMessage will put -1
     at element 0 if there is no matching message
  if (success) {
    message.sender = buffer[SENDER_OFFSET];
    message.receiver = buffer[RECEIVER_OFFSET];
    message.tag = buffer[TAG_OFFSET];
    assert(buffer[BUFFER_SIZE_OFFSET] >= payloadSize);
    memcpy(message.payload, buffer + PAYLOAD_OFFSET, payloadSize);
  }
  free_align(buffer);
  return success;
}

void Communication::sendBuffer(int receiver, int tag, void * buffer, int
    totalSize) {
  assert(0);
}
```

SPU/main.cpp

The `main()` function for the MPI program on the SPEs. Decides which ASYNPLEX process to initiate.

```
#define SPE_CODE

#include "BasisChangeManager.h"
#include "ColumnSelectionManager.h"
#include "IterationProcess.h"
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
```

```
#include <libmisc.h>
#include <stdio.h>
#include "../common/types.h"
#include "rpc.h"
#include "common/sparse.h"

volatile ParameterContext context ALIGNED_QUAD;

ppe_funcptr sendRpcMessage ALIGNED16;
ppe_funcptr sendRpcMessageBatch ALIGNED16;
ppe_funcptr receiveRpcMessage ALIGNED16;
ppe_funcptr getDimensions ALIGNED16;
ppe_funcptr getIntArrays ALIGNED16;
ppe_funcptr getTypeArrays ALIGNED16;

//int main(unsigned long long spuId __attribute__ ((unused)), unsigned long
    long parameter) {
extern "C" int main (int argc, char * argv[]) { // Using extern "C" because of
     CML
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("MPI: %d/%d\n", rank, size);fflush(stdout);

  sendRpcMessage = cellmsg_accept_rpc();
  sendRpcMessageBatch = cellmsg_accept_rpc();
  receiveRpcMessage = cellmsg_accept_rpc();
  getDimensions = cellmsg_accept_rpc();
  getIntArrays = cellmsg_accept_rpc();
  getTypeArrays = cellmsg_accept_rpc();

  int dimensions[4] ALIGNED16;
  cellmsg_rpc(getDimensions,  // Function to call
        NULL, 0, CML_BYTE_SWAP_NOT_NEEDED,    // No input
        dimensions, 4 * sizeof(int), CML_BYTE_SWAP_NOT_NEEDED); // Output
  int m = dimensions[0];
  int n = dimensions[1];
  int numNonzeroes = dimensions[2];
  int N = m + n;

  //TODO: Merge BasisChange and ColSel
  if (rank == 0) {
    BasisChangeManager basisChangeManager;
    basisChangeManager.run();
  }
  else if (rank == 1) {
```

```
    ColumnSelectionManager columnSelectionManager;
    columnSelectionManager.setup(m, n);
    columnSelectionManager.run();
  }
  else {
    if (rank != 2) {
      MPI_Finalize();
      return 0;
    }
    int intCount = numNonzeroes + (N + 1);    // A.rowIndices, A.colPos
    int typeCount = 1 + m + N + numNonzeroes; // f, b, c, A.values
    int intBytes = ROUND_UP_MULTIPLE(intCount * sizeof(int), ALIGN_BUS_WIDTH);
    int typeBytes = ROUND_UP_MULTIPLE(typeCount * sizeof(TYPE),
       ALIGN_BUS_WIDTH);
    int * intArrays = (int*)malloc_align(intBytes, ALIGN_BUS_LOG2);
    TYPE * typeArrays = (TYPE*)malloc_align(typeBytes, ALIGN_BUS_LOG2);
    printf("Getting %d ints and %d types (numNonzeroes: %d)\n", intCount,
       typeCount, numNonzeroes);fflush(stdout);
    cellmsg_rpc(getIntArrays,
          NULL, 0, CML_BYTE_SWAP_NOT_NEEDED,
          intArrays, intBytes, CML_BYTE_SWAP_NOT_NEEDED);
    cellmsg_rpc(getTypeArrays,
          NULL, 0, CML_BYTE_SWAP_NOT_NEEDED,
          typeArrays, typeBytes, CML_BYTE_SWAP_NOT_NEEDED);
    TYPE f = typeArrays[0];
    TYPE * b = typeArrays + 1;
    TYPE * c = typeArrays + 1 + m;
    SparseMatrix A;
    A.rows = m;
    A.cols = N;
    A.numNonzeroes = numNonzeroes;
    A.rowIndices = intArrays;
    A.colPos = intArrays + numNonzeroes;
    A.values = typeArrays + 1 + m + N;

    IterationProcess iterationProcess;
    iterationProcess.setup(rank - 2, size - 2, A, b, c, f);
    free_align(intArrays);
    free_align(typeArrays);
    iterationProcess.run();
  }
  MPI_Finalize();
  return 0;
}
```

A.5 Utilities

We could not find any available parsers for the MPS or CPLEX file formats, so we had to write our own. Other people may find them useful, so we include them here. Common languages of choice for writing small text manipulation programs are Python and Perl; we selected the former since we are more familiar with it.

Important note: These parsers are *not* fully compliant with the MPS and CPLEX file format specifications. They seem to work with the data sets we have used, but have not been thoroughly tested beyond that.

`mps.py` — MPS file format parser

This parser was written in the early stages of the project, when our standard simplex solver would simply expect a full tableau as input. The parser first outputs a line containing m (the number of rows) and n (the number of columns), followed by m lines containing n numbers each. The first row contains the objective function coefficients, and the leftmost column contains the right hand sides from the constraints. The tableau body contains the negatives of the original coefficients, as per our discussion in Section 2.1.2. Equality constraints are split into two less-than constraints.

We later rewrote the parser to C++ (`mps.cpp` as listed above), so that it could be an integrated part of our solver.

Note that both this parser and the C++ port are fairly simplistic, and they do *not* handle the `BOUNDS` or `RANGES` sections. As such, the number of `netlib` sets on which they (and thereby our entire standard simplex solver) can be used is reduced to 54 (from a total of 98); see the `netlib` README file for information on which sets contain which sections. Also, note that since the MPS format does not specify the direction of optimisation, and the `netlib` default seems to be minimisation, the CPLEX parser will negate the objective function for all maximisation data sets.

```
#!/usr/bin/python
from sys import stdin

class Row:
  label = None
  type = None
  values = None
  index = None
  def __init__(self, label, type, index):
    self.label = label
    self.type = type
    self.index = index
    self.values = {}
  def __str__(self):
    return self.label + " (" + self.type + "): " + str(self.values)

lines = []
for line in stdin:
```

```
  lines.append(line)

rows = {}
columnLabels = []
columnIndices = {}
i = 0
while i < len(lines):
  line = lines[i]
  i += 1
  if line[0] == ' ':
    pass
  else:
    header = line.strip()
    if header == "ROWS":
      rowIndex = 0
      while lines[i][0] == ' ':
        items = lines[i].split()
        row = Row(items[1].strip(), items[0].strip(), rowIndex)
        if row.type == "N":
          objectiveIndex = rowIndex
        rows[row.label] = row
        rowIndex += 1
        i += 1
      tableau = [None] * len(rows)
    elif header == "COLUMNS":
      columnIndex = -1
      while lines[i][0] == ' ':
        items = lines[i].split()
        lim = 2 if len(items) == 5 else 1
        columnLabel = items[0].strip()
        if not columnIndices.has_key(columnLabel):
          columnIndex += 1
          columnLabels.append(columnLabel)
          columnIndices[columnLabel] = columnIndex
        for j in xrange(lim):
          rowLabel = items[1 + j * 2].strip()
          value = float(items[2 + j * 2].strip())
          rows[rowLabel].values[columnLabel] = value
        i += 1
      for j in xrange(len(tableau)):
        tableau[j] = [0] * (len(columnLabels) + 1)
      for row in rows.values():
        for colLabel in row.values:
          tableau[row.index][columnIndices[colLabel]] = row.values[colLabel]
    elif header == "RHS":
      while lines[i][0] == ' ':
        items = lines[i].split()
```

```
        lim = 2 if len(items) == 5 else 1
        for j in xrange(lim):
          rowLabel = items[1 + j * 2].strip()
          value = float(items[2 + j * 2].strip())
          rowIndex = rows[rowLabel].index
          tableau[rowIndex][-1] = value
        i += 1

for row in rows.values():
  tab = tableau[row.index]
  if row.type == "G":
    for i in xrange(len(tab)):
      tab[i] = -tab[i]
  elif row.type == "E":
    tableau.append([-x for x in tab])
tmp = tableau[objectiveIndex]
tableau[objectiveIndex] = tableau[0]
tableau[0] = tmp

ti = 0
while ti < len(tableau):
  nonzero = 0
  for x in tableau[ti]:
    if x != 0:
      nonzero = 1
      break
  if not nonzero:
    tableau.pop(ti)
    ti -= 1
  ti += 1

print len(tableau), len(tableau[0])
for tab in tableau:
  for cell in tab:
    print cell,
  print
```

`cplex.py` — ILOG CPLEX file format parser

This parser was written in order to convert some sample data sets we received from Miriam AS to the MPS format. A restriction of the MPS format is that the row and column names are limited in length. Therefore, our parser will convert any name that is longer than eight characters to a string that is formed by appending a sequence number (starting at zero) to the string `v` (a very arbitrary choice). For instance, the fourth name that is found to be too long will be converted to `v3`. Further occurrences of the same name will of course be replaced by the same string. The parser does *not*,

however, check for name collisions with variables who actually have that name.

Note that while the CPLEX format allows constraints to be split over multiple lines, this parser not handle that, so files containing split constraints must be modified by joining such constraints into one line.

```
#!/usr/bin/python
from sys import stdin, stderr

class Equation:
  comparator = ""
  constant = 0
  values = {}
  name = ""

  def __init__(self, comparator, constant, name):
    self.comparator = comparator
    self.constant = constant
    self.values = {}
    self.name = name



class Bound:
  variable = ""
  lower = 0
  upper = None
  free = False
  fixed = False

  def __init__(self, variable):
    self.variable = variable

variableCodeNames = {}

def truncate(name):
  global variableCodeNames
  if len(name) <= 8:
    return name
  else:
    if variableCodeNames.has_key(name):
      return variableCodeNames[name]
    else:
      codeName = "v" + str(len(variableCodeNames))
      variableCodeNames[name] = codeName
      return codeName

def printCodeNames():
```

```
  global variableCodeNames
  if len(variableCodeNames) > 0:
    stderr.write("Some variable names have been changed:\n")
    stderr.write("New\tOriginal\n")
  for name in variableCodeNames:
    stderr.write(variableCodeNames[name] + "\t" + name + "\n")

def expand(string, length):
  if len(string) > length:
    raise ValueError("string too long")
  return string + " " * (length - len(string))

class LP:
  pos = 0
  lines = []
  variables = {}
  equations = []
  variableList = []
  bounds = []
  direction = "max"

  def __init__(self):
    lines = []
    variables = {}
    equations = []
    variableList = []

  def printMPS(self):
    print "NAME          UNKNOWN"
    print "ROWS"
    for eq in self.equations:
      if eq.comparator == "=":
        print " E ",
      elif eq.comparator[0] == "<":
        print " L ",
      elif eq.comparator[0] == ">":
        print " G ",
      elif eq.comparator == "obj":
        print " N ",
      else:
        raise NameError("Illegal comparator: " + eq.comparator)
      print expand(truncate(eq.name), 8)
    print "COLUMNS"
    for var in self.variableList:
      for eq in self.equations:
        if eq.values.has_key(var):
          line = expand("    " + truncate(var), 14) + truncate(eq.name)
```

```
        print expand(line, 24) + str(eq.values[var])
    print "RHS"
    for eq in self.equations:
      if eq.constant != 0:
        print expand("    B         " + truncate(eq.name), 24) + str(eq.
            constant)
    print "BOUNDS"
    for bound in self.bounds:
      if bound.free:
        print " FR BOUND     " + truncate(bound.variable)
      elif bound.fixed:
        print expand(" FX BOUND     " + truncate(bound.variable), 24) + str(
            bound.upper)
      else:
        if bound.lower != 0:
          print expand(" LO BOUND     " + truncate(bound.variable), 24) + str(
              bound.lower)
        if bound.upper != None:
          print expand(" UP BOUND     " + truncate(bound.variable), 24) + str(
              bound.upper)
    print "ENDATA"

  def parseObjective(self):
    tokens = self.lines[self.pos]
    self.pos += 1
    self.parseEquation(tokens, 1)

  def parseEquation(self, tokens, isObjective):
    if tokens[1] != '+' and tokens[1] != '-':
      tokens.insert(1, '+')
    if isObjective:
      eq = Equation("obj", 0, "OBJ")
    else:
      eq = Equation(tokens[-2], float(tokens[-1]), tokens[0][:-1])
    self.equations.append(eq)
    i = 1
    limit = len(tokens) - 1 if isObjective else len(tokens) - 3
    while i < limit:
      if tokens[i] == '-':
        sign = -1
      elif tokens[i] == '+':
        sign = 1
      else:
        print "Illegal sign on line", self.pos, ":", tokens
      if isObjective and self.direction == "max":
        sign *= -1
      try:
```

```
      value = float(tokens[i + 1])
      i += 2
    except ValueError:
      value = 1
      i += 1
    name = tokens[i]
    self.addVariable(name)
    eq.values[name] = sign * value
    i += 1

def parseEquations(self):
  while 1:
    tokens = self.lines[self.pos]
    if tokens[0][-1] != ':': break
    self.pos += 1
    self.parseEquation(tokens, 0)

def addVariable(self, name):
  if not self.variables.has_key(name):
    self.variables[name] = len(self.variables)
    self.variableList.append(name)

def parseBounds(self):
  while 1:
    tokens = self.lines[self.pos]
    if len(tokens) == 1: break
    self.pos += 1
    if len(tokens) == 2 and tokens[1] == "Free":
      bound = Bound(tokens[0])
      bound.free = True
      self.bounds.append(bound)
    elif len(tokens) == 3:
      bound = Bound(tokens[0])
      if tokens[1][0] == "<":
        bound.upper = float(tokens[2])
      elif tokens[1][0] == ">":
        bound.lower = float(tokens[2])
      elif tokens[1][0] == "=":
        bound.fixed = True
        bound.upper = float(tokens[2])
      else:
        raise NameError("Illegal bound type")
      self.bounds.append(bound)
    elif len(tokens) == 5:
      bound = Bound(tokens[2])
      bound.lower = float(tokens[0])
      bound.upper = float(tokens[4])
```

```
        self.bounds.append(bound)
      else:
        print "Unrecognised bounds line:", self.pos, ":", tokens

  def parse(self):
    for line in stdin:
      tokens = line.split()
      if len(tokens) == 0 or tokens[0] == '\\': continue
      self.lines.append(tokens)
    self.pos = 0
    while self.pos < len(self.lines):
      if self.lines[self.pos][0] == "Maximize":
        self.direction = "max"
        self.pos += 1
        self.parseObjective()
      elif self.lines[self.pos][0] == "Minimize":
        self.direction = "min"
        self.pos += 1
        self.parseObjective()
      elif self.lines[self.pos][0] == "Subject":
        self.pos += 1
        self.parseEquations()
      elif self.lines[self.pos][0] == "Bounds":
        self.pos += 1
        self.parseBounds()
      else:
        self.pos += 1

lp = LP()
lp.parse()
lp.printMPS()
printCodeNames()
```