Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Computer and Information Science

Master Thesis

# Linear programming on Cell/BE

by

**Åsmund Eldhuset**

Supervisor: Dr.Ing. Lasse Natvig
Co-supervisor: Dr. Anne C. Elster

Trondheim, June 1, 2009

**Abstract**

(TODO: )

# Acknowledgements

(TODO: )

# Contents

# List of Figures

# List of Tables

# Listings

# List of Symbols and Abbreviations

| Abbreviation | Description | Definition |
|---|---|---|
| Cell BE | Cell Broadband Engine | page 8 |
| ILP | Integer linear programming | page 4 |
| ILP | Instruction-level parallelism | page 8 |
| LP | Linear programming | page 3 |
| PPE | PowerPC Processing Element | page 8 |
| PPU | PowerPC Processing Unit | page 8 |
| SPE | Synergistic Processing Element | page 8 |
| SPU | Synergistic Processing Unit | page 9 |

Chapter 1

# Introduction

(TODO: )

**Task description** (Natvig: This is the task description I entered in DAIM. Should probably be changed a bit...) The aim of the project is to implement a parallel linear solver for large sparse problems on the Cell BE using the Simplex method. Interior point methods may also be investigated.

(TODO: A paragraph about Miriam)

Chapter 2

# Background

(TODO: Chapter introduction)

## 2.1 Linear programming

(ITP: Section introductions)

This section is primarily based on [7], [2](TODO: and [5] if we write about artificial variables).

### 2.1.1 Problem formulation. Standard and slack forms

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 [2]). 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.

An example of a simple linear programming problem would be a factory that makes two kinds of products based on two different raw materials. (Natvig: This example just uses some random numbers; I will construct an example, probably using only integers, that can be solved neatly in a few iterations.) The profit the company makes per unit of product A is $10.00, and the profit of product B is $12.50. Producing one unit of A requires 2 units of raw material R and 3 units of raw material S; one unit of B requires 3 units of R and 1.5 units of S.

[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(TODO: citation).

The company possesses 100 units of raw material R and 50 units of raw material 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 $10.00x_1 + 12.50x_2$ 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:

- $2x_1 + 3x_2 \leq 100$ (the production of A and B cannot consume more units of raw material R than the company possesses)
- $3x_1 + 1.5x_2 \leq 50$ (same for raw material S)
- $x_1, x_2 \geq 0$ (the company cannot produce negative amounts of its products)

We will use this example throughout this section. (Natvig: This has not yet been done. I intend to interleave it with the presentation of the algorithm steps, unless you think it should be presented separately afterwards.)

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 $f = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$, 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 produce this optimal 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 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

[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 [4].

the *standard form*, in which only less-than-or-equal-to inequalities with all variables on the left hand side are allowed. (TODO: Why are not less-than allowed?) 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: $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ 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 $f$, we let our objective function be $-f$. A linear program in standard form can be expressed as follows:

Maximise

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

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

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

with respect to

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

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

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

**Feasible** if it does not violate any of the constraints

**Infeasible** if it violates any constraint

**Basic** if it consists of setting all variables except the slack variables to zero

**Optimal** if it is feasible and no other feasible solutions yield a higher value for the objective function

(TODO: Move this paragraph to next section? Natvig says "Ta med etter behov".) The *linear programming theorem* (TODO: Is it actually called this? Find something to cite) states that the optimal solution of a linear program, if it exists, occurs when at least $m$ variables are set to zero. (TODO: Combinatorics. Mention cycling here?)

### 2.1.2 The simplex method

*The simplex method*[3], developed by George Dantzig[3], was the first systematic approach for solving linear programs. It requires the linear program to be in slack form. The coefficients are written down in a *tableau* that changes as the method progresses. The nonnegativity constraints are not represented anywhere, but are implicitly maintained by the method. Because the equations will undergo extensive rewriting, it will be convenient to not distinguish the slack variables from the other variables, so we will relabel $w_i$ to $x_{n+i-1}$ for $i = 1, \ldots, m$. Thus, the total number of variables is $n + m$.

(TODO: Show the example problem in slack form and in tableau form)

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, the set of basic variables is the set of the original slack variables. The sizes of the basic and nonbasic sets are constant, with $|\mathcal{B}| = m$ and $|\mathcal{N}| = n$.

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). 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

[3]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.

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 $f = c_0$) is optimal — we can be certain of this since linear functions do not have local maxima.

(TODO: relabel $w_i$) It seems reasonable to select the variable with the greatest coefficient, say, $x_e$. 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 $x_e$ until one of the basic variables becomes zero. Let us look at row $i$, which is now reduced to $w_i = b_i - a_{ie}x_e$ since all nonbasic variables except $x_e$ are zero. If $a_{ie}$ is positive, the value of $w_i$ will decrease as $x_e$ increases, so the largest allowable increase is limited by $b_i$. Thus, by setting $x_e = \frac{b_i}{a_{ie}}$, $w_i$ becomes zero. However, other equations may impose stricter conditions. By looking at all rows where $a_{ie}$ is positive, we can determine an $l$ such that $\frac{b_l}{a_{le}}$ is minimal and set $x_e = \frac{b_l}{a_{le}}$. This will cause $x_l$ to become zero. If all $a_{ie}$ are nonnegative, we can increase $x_e$ indefinitely without any $w_i$ 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.

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 no $b_i$ is negative, so that we can repeat the previous steps all over again. The nonbasic variable that was selected to be increased, $x_e$, is called the *entering variable*, since it is about to enter the collection of basic variables. $x_l$, which becomes zero when $x_e$ is increased appropriately, is called the *leaving variable*, since it is to be removed from said collection. Keep in mind that since $x_l$ is a basic variable, it only occurs in one equation. We can eliminate the entering variable from (and introduce the leaving variable into) the set of *nonbasic* variables (the "main" part of the tableau) by rewriting the selected equation and adding appropriate multiples of it to each of the other equations: (TODO: Complete this) This step is called a *pivot*. After pivoting, we again have a tableau in which all $b_i$ are nonnegative, and the entire process may be repeated.

**Degeneracy** (TODO: Briefly discuss degenerate pivots.)

**Initialisation**

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. (TODO: Phase I and Phase II) This may not always be the case. We get around this by introducing an *auxiliary problem* which is based on the initial problem and is guaranteed to have a basic feasible solution, and whose solution will provide us with a starting point for solving the original problem. (TODO: Complete this)

#### Formal algorithm statement

(TODO: Use the `algorithm` package to give a compact description of the simplex method) (TODO: Should ideally be recognisable in the real code; maybe reference the real code here (or the other way around?))

#### Complexity and numerical instability

(TODO: )

(ITP: Other stuff that should perhaps be added: geometric interpretation; duality)

### 2.1.3 Interior point algorithms

### 2.1.4 Use of LP to solve advanced flow problems

(TODO: Consult Miriam on this)

## 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 was 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[6]. Several obstacles to such goals exist; in particular the infamous *brick walls*[1]:

**Memory wall** (TODO: )

**Power wall** (TODO: )

**ILP wall** *Instruction-level parallelism* (ILP) techniques such as pipelines and (TODO: )

### 2.2.1 Architecture

#### Overview

The Cell BE consists of one *PowerPC Processing Element* (PPE) and eight *Synergistic Processing Elements* (SPE)

#### PPE

*PowerPC Processing Unit* (PPU)

#### SPE

*Synergistic Processing Unit* (SPU)

#### Memory bus and DMA controller

Base addresses (both in local storage and in system memory (TODO: correct?)) for all DMA transfers must be aligned on a 16-byte (quadword) border(TODO: term?), and the data to be transferred must be a multiple of 16 bytes. Performance is improved if aligned, whole cache lines (128 bytes(TODO: verify)) are transferred at a time.

Another method that is available for communication between the cores is

### 2.2.2 Programming methods

### 2.2.3 Tools and libraries

(Natvig's comment: Good: which libs are used in the project? Better: Which libs are relevant for the project?)

Chapter 3

# Design

(TODO: Chapter introduction)

## 3.1 Overall approach

(TODO: Gradual, step by step approach)

## 3.2 Dense simplex

In order to become familiar with programming the Cell BE, we initially implemented a few versions of the simplex method for dense problems. (Natvig's comment: This can be justified when we have a task description and "angrepsmåte")

### 3.2.1 PPE version

(TODO: Far from finished) As described in Section 2.2.1, 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 is the low arithmetic intensity, 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.

(TODO: Something on why we chose C++?)

(Natvig's comment: Avoid the "not invented here" syndrome. Look into reusing existing code/libraries) We wrote a class called `Matrix`, which represents a dense matrix and supports standard matrix operations. The initial version was nonvectorised (SISD) and used just standard C++. Rewriting this to utilise the vector operations involved only a few fairly trivial steps:

- Use `malloc_align` instead of `new` to get memory blocks that are aligned on proper boundaries (TODO: Not sure if this was even necessary)
- Pad the rows with zeroes such that the number of elements in each row is a multiple of four, so that the vector operations will not "fall of the end" of each row
- Rewrite the loops in the matrix operation functions (such as `addRows` and `multiplyRow` to use vector operations. (Natvig's comment: Preferrably pseudocode here) `data` is a pointer to the array that contains the entire matrix. `physicalCols` is the number of columns rounded up to the nearest multiple of `VECTOR_WIDTH`, which is set to 4.

```
void Matrix::addRows(int sourceRow, int destinationRow,
    float factor) {
  vector float factor_v = (vector float){factor, factor,
      factor, factor};
  vector float * source_v = (vector float *)(data +
      sourceRow * physicalCols);
  vector float * destination_v = (vector float *)(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]);
  }
}
```

(TODO: Do loop unrolling as well?)

### 3.2.2 SPE version

## 3.3 Sparse simplex

## 3.4 Dense interior point

## 3.5 Sparse interior point

Chapter 4

# Implementation and testing

(TODO: Chapter introduction)

## 4.1 Simplex algorithm

## 4.2 Test plan

### 4.2.1 Unit testing

(TODO: )

### 4.2.2 Large data sets

(TODO: Something on the `netlib` LP problem set)

### 4.2.3 (TODO: Other implementations)

Chapter 5

# Evaluation

(TODO: Chapter introduction)

## 5.1 Performance measurements

(TODO: Describe system specifications and how timing was performed)

### 5.1.1 (TODO: What to measure)

### 5.1.2 (TODO: How to measure)

## 5.2 Results

### 5.2.1 Dense simplex

### 5.2.2 Sparse simplex

### 5.2.3 Dense interior point

### 5.2.4 Sparse interior point

## 5.3 Discussion

Chapter 6

# Conclusion

(TODO: )

## 6.1 Experiences

## 6.2 Future work

# Bibliography


[1] 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. 8]

[2] 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]

[3] G. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963. [cited at p. 6]

[4] 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]

[5] F. S. HILLIER AND G. J. LIEBERMAN, *Introduction to Operations Research*, McGraw-Hill Science/Engineering/Math, July 2004. [cited at p. 3]

[6] 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. 8]

[7] R. J. VANDERBEI, *Linear Programming: Foundations and Extensions*, Springer, 2nd ed., 2001. [cited at p. 3]

# Appendices

Appendix A

# Schedule

This appendix will obviously be deleted before submission.

**Week 8** Finished the implementation of a dense Simplex for a regular CPU and test with `netlib` datasets. Implement a vectorised (SIMD) dense Simplex on the PPE

**Week 9** Struggle with numerical instability

**Week 10** Implement a vectorised dense Simplex running in parallel on the SPEs

**Week 11** — " — (delayed)

**Week 12** — " — (delayed)

**Week 13** Study interior point algorithms

**Week 14** First draft of report

**Week 15** Easter vacation - read on interior point algorithms

**Week 16** Implement a dense, non-parallelised interior point algorithm

**Week 17** Decide on whether to pursue simplex or interior point. Making a test plan. Experiment with different approaches to sparse storage; look into numerical stability with single-precision values

**Week 18** Look into autotuning?

**Week 19**

**Week 20** Performance measurements and graphing

**Week 21** Frenetic report writing

**Week 22** — " —

**Week 23** Ordinary submission deadline. Will try to submit as close to this date as possible

**Week 24**

**Week 25**

**Week 26**

**Week 27** Natvig goes on vacation

**Week 28**

**Week 29** Final deadline: July 19