Mujahed Eleyat

Accelerating the Regina Network Flow Simulator on Multi-core Systems

Thesis for the degree of Philosophiae Doctor
Trondheim, September 2014

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

Emerging multi-core processors, which came as a result of manufacturers inability to keep increasing the frequency of single core processors, represent a great but also challenging opportunity to accelerate compute-intensive applications. Challenges are related to the fact that not all time-consuming applications are easy to parallelize efficiently. In addition to that, emerging multi-core systems have different architectures and efficient utilization still requires the programmer to have deep knowledge of the architecture of the target multi-core system, which also strongly raises the issue of portability.

The main goal of this thesis is to investigate the use of the inherent parallelism of multi-core systems to accelerate a gas flow simulator called MIRIAM Regina and developed by a company called Miriam AS. MIRIAM Regina spends most time solving a series of sparse linear programming (LP) problems. Therefore, most of the research was focused on building an IPM-based parallel LP solver and testing it using different architectures with different characteristics. The solver was first tested using the heterogeneous Cell BE processor found in PlayStation 3, which was of high interest to Miriam AS because of its high “FLOP/dollar” ratio. We have also tested the solver on the 2 x 6-core AMD Opteron processor (Istanbul), a good representative of the large class of modern homogeneous multicore processors.

We started with a serial IPM-based implementation that is part of the GLPK solver. In addition, we justified using interior point methods for solving these problems which consequently lead our research to investigating ways to enhance the efficiency of sparse Cholesky factorization. The main focus was how to parallelize Cholesky factorization and utilize cache efficiently. Moreover, a new method has been suggested to enhance the amalgamation of small blocks and to create bigger blocks to reduce the overhead of processing small tasks. The data sets provided by the company were too small to give any significant speedup and large data sets were not available. Therefore, we tested our solver using some large NETLIB problems and were able to accelerate the solution of them.

For some customers, MIRIAM Regina also spends relatively long time finding the maximum flow in a network under upper and lower flow constraints on each pipe. We suggest a heuristic to find a near optimal solution and investigate a way to parallelize the method. Both the heuristic and its parallelization were implemented and tested on the 2 x 6-core AMD Opteron processor (Istanbul).
**Preface**

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfillment of the requirements for the degree of philosophiae doctor.

This doctoral project is one of the first projects that started as a result of the *industrial PhD* program launched in 2008 by the Norwegian research council. The main idea of the program is that the research council stimulates research in Norwegian companies by allowing them to get a grant that covers 50% of all the expenses needed to fund a PhD student for three years. In this project, Mujahed Eleyat was employed as a PhD scholar by MIRIAM AS, a Norwegian company that develops software products for oil and gas flow simulations.

As part of the project, the PhD candidate spent one year at the Department of Computer and Information Science, NTNU, Trondheim, and two years at MIRIAM AS, Halden. From the NTNU side, the project was led by Lasse Natvig as a main supervisor during the whole project life cycle and Jørn Amundsen as a co-supervisor for about the second half of the project. From the company side, the project was technically supervised by Christophe Spaggiari, the Chief Technology Officer of MIRIAM AS, who was also the direct manager of the PhD candidate.
Acknowledgements

First, I would like to thank my main supervisor Prof. Lasse Natvig for his continuous support, motivation and patience throughout the research. His guidance helped me in all the time of research and writing of this thesis. I could not wish having a better or friendlier supervisor.

I would also like to express my gratitude to my co-supervisor, Assoc. Prof. Jørn Amundsen, for his advice and feedback and to my direct manager in the company Mr. Christophe Spaggiari for his advice and help. I would also like to thank Prof. Dag Haugland for his ideas and advice and Assoc. Prof. Magnus Lie Hetland for very helpful discussion and feedback.

I would like to thank the Norwegian Research Council and MIRIAM AS for funding the project. In addition, I would like to thank the Department of Computer and Information Science for providing me an office while staying at the Norwegian University of Science and Technology, allowing me to use their resources, and the fund that they provided for research purposes.

I would like to thank my wife for patience and support and my parents for support and best wishes.

Mujahed Eleyat
September 8, 2014
2.3.2 The PPE
2.3.3 The SPEs
2.3.4 Communication Architecture
2.3.5 PowerXCell 8i and usage in supercomputers
2.3.6 Programming the Cell – challenges and programming models
2.4 AMD Opteron 2400-series (Istanbul)
2.5 Using mixed-precision iterative refinement
2.6 Max flow with minimum lot sizes

CHAPTER 3 METHODOLOGY
3.1 The LP solver
3.2 Simplex or IPM
3.3 The GLPK LP solver
3.4 NETLIB and Regina data sets
3.5 Boost C++ libraries – the Boost Graph Library

CHAPTER 4 RESEARCH PROCESS
4.1 Regina and its time critical parts
4.1.1 Regina and Linear Programming
4.2 Building an efficient LP solver
4.2.1 IPM-based LP solver on the Cell BE processor – papers I, II, and III
4.2.2 Cache efficient IPM-based LP solver on 6-core AMD Opteron processors – paper IV
4.3 Maximum flow with minimum lot sizes – papers V and VI

CHAPTER 5 RESEARCH RESULTS
5.1 Paper I Implementation of a Linear Programming Solver on the Cell BE Processor
5.1.1 Abstract
5.1.2 Retrospective view
5.2 Paper II Mixed-Precision Parallel Linear Programming Solver
5.2.1 Abstract
5.2.2 Retrospective view

5.3 Paper III  IPM based sparse LP solver on a heterogeneous processor
5.3.1 Abstract
5.3.2 Retrospective view

5.4 Paper IV  Cache-Aware Matrix Multiplication on Multicore Systems for IPM-based LP Solvers
5.4.1 Abstract
5.4.2 Retrospective view
5.4.3 Errata for Paper IV

5.5 Paper V  The maximum flow problem with minimum lot sizes
5.5.1 Abstract
5.5.2 Retrospective view

5.6 Paper VI  Parallel algorithms for the maximum flow problem with minimum lot sizes
5.6.1 Abstract
5.6.2 Retrospective view

CHAPTER 6  CONCLUDING REMARKS

6.1 Conclusion
6.2 Contributions
6.3 Future Work
6.4 Bibliography

PAPER I  IMPLEMENTATION OF A LINEAR PROGRAMMING SOLVER ON THE CELL BE PROCESSOR

Abstract

1.1 Introduction

1.2 The Cell BE Architecture

1.3 Mehrotra’s Predictor-Corrector Method:

1.4 Parallel Sparse Cholesky Factorization
1.4.1 Sparse Cholesky factorization
1.4.2 Sparse Block Cholesky factorization
I.4.3 Parallel Sparse Block Cholesky Factorization 48
I.4.4 Adapting Parallel Sparse Block Cholesky Factorization to the Cell BE Processor 49
I.5 Parallel GLPK Stability 51
I.6 Related Work 52
I.7 Results and Future Work 52

PAPER II MIXED-PRECISION PARALLEL LINEAR PROGRAMMING SOLVER 57

Abstract 59

II.1 Introduction 61
II.2 Parallel sparse LP solver 61
II.3 Mixed precision 64
II.4 Related Work 65
II.5 Experimental Results 66
II.6 Discussion and future work 69

PAPER III IPM BASED SPARSE LP SOLVER ON A HETEROGENEOUS PROCESSOR 73

Abstract 75

III.1 Introduction 77
III.2 The Cell BE processor - a heterogeneous processors 78
III.3 Parallel Cholesky factorization on the Cell BE processor 79
III.3.1 Two-phase implementation 79
III.3.2 Parallel block Cholesky factorization 80
III.4 Small blocks challenge 81
III.4.1 Parent-child supernode amalgamation 81
III.4.2 New blocking method 82
III.4.3 Blind amalgamation 84
III.5 Performance results 85
III.6 Conclusion 87
Abbreviations

BGL the Boost Graph Library
ccNUMA cache coherent Non-Uniform Memory Access
Cell BE Cell Broadband Engine
CellSs Cell Superscalar
CLP (COIN-OR) An open-source Linear Programming solver
CMP Chip MultiProcessor
DDR3 Double Data Rate Type 3
DMA Direct Memory Access
EIB Element Interconnection Bus
FLOPS Floating Point Operations Per Second
FPGA Field Programmable Gate Arrays
GLPK GNU Linear Programming Kit
GPU Graphics Processing Unit
HPC High Performance Computing
HT HyperTransport
IPM Interior Point Method
KKT Karush-Kuhn-Tucker
LP Linear Programming
MFC Memory Flow Controller
OpenMP Open Multi-Processing
PCI Peripheral Component Interconnect
PPE Power Processing Element
PS3™ PlayStation® 3
QPACE Quantum Chromodynamics Parallel Computing on the Cell
SIMD Single Instruction Multiple Data
SPE Synergistic Processing Element
SPU Synergistic Processing Unit
Chapter 1

Introduction

MIRIAM AS is a company with a main software product called MIRIAM Regina used for oil/gas flow simulation. Simulation of the flow in one network within one country requires running several hundred replications where each replication needs solving tens of thousands of linear programming (LP) problems. Therefore, it usually takes several hours to simulate such network on a traditional uniprocessor system.

The ambitions of MIRIAM Regina are to aggregate networks across Europe into one big network and simulate the resulting network. This means solving much bigger LP problems that is expected to take very long time to simulate using traditional single core processors. One necessary step to achieve its ambition is to use a more powerful computing hardware that could solve the LP problems much faster. This goal led the company towards investigating the possibility of exploiting the computational power of emergent multi-core systems.

The focus of this research is accelerating the solution of LP problems on multi-core systems including both heterogeneous and homogeneous systems. We tested our code on the Cell BE processor found in PlayStation 3 (PS3) especially since it was of high interest to Miriam AS. We have also tested the solver on the 2 x 6-core AMD Opteron processor (Istanbul).

1.1 Multi-core processors

1.1.1 Parallelism and multi-core processors

Multi-core processors are not the first and only hardware that use parallelism to increase the performance of computing systems. In fact, the concept of parallelism has been considered since the early days of computing. *Word parallel* operations, *prefetching*, *function parallelism*, and *pipelining* can all be considered as techniques built around making operations in parallel. On the other hand, an FPGA (Field Programmable Gate Arrays) device is also a parallel hardware made of a matrix of reconfigurable gate array logic circuitry that can be configured to create a hardware implementation of the software application. Because they represent a hardware implementation and do not require an operating system, they are more efficient and deterministic than general purpose processors. Moreover, integration between FPGAs and multi-core processors has also been a topic of interest for the past few years [1].

Increasing the clock speed was the main method for increasing the performance of uniprocessor systems. However, processor designers couldn’t continue to use this method for several reasons, mainly the difficulty of dissipating the resulting heat [2]. To address this challenge, their focus has shifted toward multi-core systems, also called chip multiprocessors (CMP), where a uniprocessor is replaced by two or more cores placed on
the same chip. Given that the power is proportional to the square of the voltage, the new cores consume much less power than traditional uniprocessors because they run at a lower frequency and need lower voltage. On the other hand, multi-core systems offer a higher collective processing power, but its utilization requires efficient parallelization of the application and sometimes deep knowledge of the architectural details.

There are two main categories of multi-core systems. A homogeneous multi-core system has identical cores that execute the same instruction set and are similarly connected to cache and other components on the chip. However, a heterogeneous multi-core system has cores of different types. We will continue this section by introducing a few of the major multi-core processors manufactured by main processor vendors. Then we introduce the architectures of the Cell BE processor and the 2 × 6-core AMD Opteron (Istanbul) processor and the in the following two sections (1.1.2 and 1.1.3 respectively).

In the beginning of 2011, Intel® started manufacturing some products using the Sandy Bridge [3] processor (second generation Intel Core™) and then it released the Ivy Bridge processors (third generation Intel Core™) one year later [4]. Sandy Bridge is based on the 32nm process and represents the latest micro architecture, called a tock in the Intel tick-tock model [5]. This architecture gathered the memory controller, the PCI Express (PCI) controller, and video functions in the processor die allowing them to easily share data and power. Ivy Bridge uses the Sandy Bridge architecture but it is based on the 22nm die process (shrink of production process is called tick in Intel tick-tock model). In addition, Ivy Bridge is more power efficient and it has some additional advancements as it supports PCI Express (PCIe) 3.0 and DDR3L (low-voltage) memory.

Graphics Processing Units (GPUs) have a large number of fine-grained parallel processors that were originally designed to manipulate computer graphics. Modern GPUs are not only efficient for graphics rendering, but their high programmability and capability made them very attractive to be used in high performance computing [6]. To be used as accelerators, the CPU usually manages access to them in order to execute a highly parallel compute critical part of the application. Indeed, a lot of research has been made to redesign and implement a broad range of computationally intensive applications so that they make use of the huge processing capability offered by GPUs. More information about GPU architectures and its use in high performance computing can be found in overview paper GPU Computing by Owen [7].

Major vendors of GPUs include NVIDIA, AMD, Intel, and ARM. NVIDIA [8] reports that its Tesla GPUs are designed taking the acceleration of scientific computing and programmability into much consideration. It offers more than one teraflops of double precision floating point and is built around the Kepler architecture [9] claimed by NVIDIA to be the world’s fastest and most efficient high performance computing (HPC) architecture.

1.1.2 The Cell BE processor

The Cell Broadband Engine (Cell BE) is a heterogeneous processor that was designed by Sony, Toshiba, and IBM, an alliance known as STI [10]. It has one main core (Power Processing Element, PPE) and 8 accelerator cores (called Synergistic processing elements (SPEs)). The cores and an on-chip memory controller are linked together by an element
interconnection bus (EIB). The main goal of the Cell BE was to allow efficient execution of
game/multimedia applications. It was also designed to provide a real-time responsiveness to
both the user and the network [10]. Further details about the Cell BE can be found in section
2.3.

1.1.3 AMD Opteron 2400-series (Istanbul)

Istanbul is the codename for the 6-core AMD Opteron 2400 and 8400 series processors that
was introduced in June 2009. It is manufactured in a 45nm process and based on the AMD
64-bit K10 architecture. The processor has three levels of cache with 6 MiB L3 cache that is
shared for the 6 cores. AMD said the Istanbul processor provides the user with 6-core
performance while it uses the same socket and thermal envelope as its Quad-core Opteron

1.2 Research Questions

The main research question of this thesis is:

**How to accelerate the Regina gas flow simulator on multi-core systems (all papers)**

The following sub-questions stems from the main question above:

RQ1: **How to accelerate an IPM-based LP solver (Papers 1, 2, 3, and 4)**

RQ2: **How to accelerate sparse matrix multiplication in interior point methods (IPMs) (Paper 4)**

RQ3: **How to accelerate finding the maximum flow in a network with minimum lot
sizes (Papers 5 and 6)**

1.3 Thesis outline

The remainder of this thesis is organized in five chapters. Chapter 2 presents the
background information mainly about MIRIAM Regina, LP, and some multi-core systems.
Chapter 3 explains the methodology adopted to make our research including the open
source code and the data sets used to conduct our experiments. Chapters 4 and 5 explain the
way the PhD work was accomplished including an overview and retrospective views of the
published papers. Finally, chapter 6 concludes the thesis with a summary of the
contributions, a match between the published papers and research questions, and a few
suggestions for future work.
Chapter 2

Background

This chapter introduces the background material necessary to understand the thesis and the included papers. MIRIAM Regina, the gas flow simulator, is introduced in section 2.1. Regina spends most time solving many Linear Programming problems; therefore, Linear Programming and the two main methods to solve the problems are presented in section 2.2. Moreover, the two main multicore architectures used to conduct experiments are explained in sections 2.3 and 2.4. Mixed-Precision iterative refinement, a technique used in some applications to exploit the relatively high single-precision arithmetic performance while achieving enhanced accuracy, is discussed in section 2.5. Max flow with minimum lot sizes is finally illustrated in section 2.6.

2.1 Gas flow simulation – MIRIAM Regina

MIRIAM® Regina [12] is a simulation tool that is developed by MIRIAM AS [13] to evaluate the operational performance of continuous process plants in terms of equipment availability, production capability and maintenance requirements. The tool is suitable for industries and businesses that are based on the generation of a continuous production flow. Hence it is very suitable for the petroleum industry to study the production and distribution of oil and gas.

The system to be modeled is defined as a simplified network of elements. Input data include capacities, failure and repair data for each process stage component, information on supplies of utilities and resources, maintenance schedules and system operating rules. Output results include system production statistics, production availability and deliverability, planned maintenance, resource and spare parts usage etc. The results are available both as numerical results and playback of the model. The playback shows how the elements’ status changes throughout the simulation.

MIRIAM Regina models the stochastic behavior of the system over time using the Monte Carlo simulation method [14]. To achieve a realistic behavior of the system, a simulation run generates a sequence of events, corresponding to system state changes, and performance statistics are gathered as the run proceeds. Naturally, the result will differ from one simulation to another due to the random event sequences. This necessitates several simulation replications in order to obtain an estimate with good accuracy. In terms of computation, a sequence of LP instances is generated in correspondence with each sequence of events and solving these sequences is the most compute-intensive part of MIRIAM Regina.
2.1.1 Network elements

The MIRIAM Regina system networks are composed of network elements and links between them. The element types and links are defined in the Model Drawing Interface. There are four main element types available in MIRIAM Regina: Boundary points, Process stages, Items and Storage units. An example of a gas network is shown in Figure 1. Boundary points define the edges of the system to be modeled, and are the points at which flow enters and leaves the system. Process stages are the main functional units of the MIRIAM Regina model. Each process stage must have at least one upstream and one downstream link in the model, and contains at least one Item. Items normally represent a piece of equipment or an equipment failure mode. Items are where failure and repair data are entered in the model. Storage units are defined by their capacity and the rate of throughput. Storage units can act as buffers within the system, as they can be filled during downstream restriction of throughput or drained during upstream restriction.

![Figure 1: An example of Regina model network.](image)

2.1.2 Flow Algorithm and flow allocation

Some of the most complex processing within the system involves the calculation of flow through the network, given the flow rate limits on elements. The flow algorithm is responsible for the flow allocation in the flow network. It primarily attempts to maximize the flow through the network, given the flow limitations at the nodes (boundary points, process stages, items and storage tanks) in the network. These flow limitations are the given flow limitations at each node, failed items in the network, empty storage tanks etc.

2.2 Linear Programming

Linear Programming (LP) is well rooted as a principle for approaching a complex problem, which involves the selection of values for a number of interrelated variables. The selection of these variables is aimed at maximizing or minimizing a certain objective, be it profit or
loss in a business setting, speed or distance in a physical problem, or expected return in the environment of risky investments [15]. Moreover, the solution process of most practical scientific and engineering LP problems involves the manipulation of very large sparse matrices of linear equations [16] i.e. matrices which has a large percentage of zero entries. Such problems require usage of efficient methods to be implemented on powerful computer systems.

From a mathematical point of view LP is an optimization problem that is composed of a linear objective function and a set of constraints that are linear functions of a set of variables [15]. The variables, called decision variables are denoted by $x_1, x_2, ..., x_n$. The objective function can be expressed as:

$$Z = c_1 x_1 + c_2 x_2 + ... + c_n x_n$$

However, a constraint in general form can be either an equation or an inequality and can be expressed as:

$$a_1 x_1 + a_2 x_2 + ... + a_n x_n \leq b$$

**2.2.1 LP problem in standard and canonical forms**

No matter how the constraints differ from one problem to another, the LP problem can be expressed using the following standard form:

minimize $c_1 x_1 + c_2 x_2 + ... + c_n x_n$

subject to $a_{i1} x_1 + a_{i2} x_2 + ... + a_{in} x_n = b_i$

and $x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0$.

(1)

Where the $b_i$'s, $c_i$'s and $a_{ij}$'s are fixed real constants and the $x_i$'s are real numbers to be determined [15].

In vector notation, the standard problem is expressed as:

minimize $c^T x$

subject to $Ax = b$ and $x \geq 0$.

(2)

where is an $m \times n$ matrix, $m$ is the number of constraints and $n$ is the number of variables, $x$ and $c$ are $n$-dimensional row vectors, and $b$ is an $m$-dimensional column vector.

An LP problem in general form can easily be transformed to a standard form as follows:

a) If the objective function $Z$ is of maximization type, it is replaced by a minimization objective function $Z'$ obtained by negating the original objective function $Z$ i.e. $Z' = -Z$

b) Inequality constraints of the form $a_{i1} x_1 + a_{i2} x_2 + ... + a_{in} x_n \leq b_i$ are converted to equality constraints by adding a non-negative variable, $w_i$, called a slack variable. The equivalent constraint is $a_{i1} x_1 + a_{i2} x_2 + ... + a_{in} x_n + w_i = b_i$ and $w_i \geq 0$.
c) Inequality constraints of the form \( a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \geq b_i \) are converted to equality constraints by subtracting a non-negative variable, \( w_i \). The equivalent constraint is \( a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - w_i = b_i \) and \( w_i \geq 0 \)

d) Each unrestricted (free) variable is replaced by the difference of two new non-negative variables.

An LP problem is said to be in canonical form if it is expressed as shown below:

\[
\begin{align*}
\text{minimize} & \quad c_1x_1 + c_2x_2 + \cdots + c_nx_n \\
\text{subject to} & \quad 1x_1 + 0x_2 + \cdots + 0x_m + a_{1,m+1}x_{m+1} + \cdots + a_{1n}x_n = b_1 \\
& \quad 0x_1 + 1x_2 + \cdots + 0x_m + a_{2,m+1}x_{m+1} + \cdots + a_{2n}x_n = b_2 \\
& \quad \vdots \\
& \quad 0x_1 + 0x_2 + \cdots + 1x_m + a_{m,m+1}x_{m+1} + \cdots + a_{mn}x_n = b_m \\
\text{and} & \quad x_1 \geq 0, x_2 \geq 0, \ldots, x_n \geq 0,
\end{align*}
\]

In canonical form, \( x_1 \ldots x_m \) variables are called dependent variables or basic variables while the variables \( x_{m+1} \ldots x_n \) are called independent variables or nonbasic variables. A basic solution can be obtained by setting all nonbasic variables to 0. If a basic solution satisfies all the constraints and all components of \( x \) are non-negative, it is called a basic feasible solution. The set of all feasible solutions to an LP problem is a convex polyhedron and if this set is bounded, one of the extreme points is the optimal solution [17].

In general, an LP problem can be expressed in canonical form by performing a set of pivotal operations through elementary operations. Elementary operations include multiplying a row with a non-zero constant and adding a row to another one multiplied by a number. It can be noted that if all constraints are inequality constraints (\( \leq \)) then the basic variables are the slack variables that are introduced to convert them to equality constraints and the LP problem can be easily expressed in canonical form without the need to elementary operations.

Two main methods are used to solve LP problems namely simplex and IPM methods. The following sections discuss both of them.

2.2.2 Simplex

The simplex method has dominated the area of solving LP problems since its invention by Dantzig in 1949 and until the emergence of practical IPMs in the 80s [18]. There are two main variants of Simplex, the full tableau simplex, also called standard simplex method and the revised simplex method. The latter is commonly used and it is more efficient for sparse problems. However, it is shown by Hall that, mainly due to high ratio of communication to computation and sparsity, none of the attempts to parallelize the revised simplex methods has offered significantly improved performance when solving general large sparse LP problems [19].

The simplex algorithm operates on a tableau which is the array of coefficients of the constraints in canonical form and a row made of the coefficients of the objective function,
called cost coefficients. It starts with a known extreme point (a basic feasible solution) and moves to an adjacent extreme point such that a better solution is found. As explained in the previous section, an initial basic feasible solution can be found by setting all nonbasic variables to zero. In order to move to an adjacent extreme point, a basic variable has to be replaced by a nonbasic variable. The process continues until the optimal solution is found. Each iteration involves the following steps:

a) Determining whether an optimal solution is found
If all cost coefficients are positive then the current solution is the optimal solution. However, if any of the cost coefficients is negative, the current solution can be improved and the following three steps are performed.

b) Selecting a nonbasic variable to be entered into the basis (entering variable)
One of the nonbasic variables is selected to go to the set of basic variables. It is chosen to improve the solution i.e. to decrease the objective function. Usually, \( x_k \) is chosen to be the entering variable if \( c_k \) is the minimum cost coefficient.

c) Selecting a basic variable to be moved out of the basis (leaving variable)
One of the basic variables is selected to go to the set of nonbasic variables. If \( x_k \) is the entering variable then \( x_r \) is decided to be the leaving variable if \( \frac{b_r}{a_{rk}} \) is minimum for all possible \( r \), provided \( a_{rk} \) is positive. \( a_{rk} \) is called the pivotal element.

d) Transforming the equations through pivotal operations to another canonical according to the selections made in steps b and c.

2.2.3 Interior point methods

In 1984, the first practical Interior Point Method (IPM) was developed by Narendra Karmarkar. Karmarkar’s algorithm, which uses spheres and projective geometry to construct a sequence of points converging to a solution of an LP problem, represented a real competitor to simplex [20]. After that, an intensive research has been done in this direction and several variants have appeared such as the dual logarithmic barrier method, the primal-dual logarithmic barrier method, and the primal-dual Newton method [21]. IPM variants that are based on the primal-dual framework makes the most productive and practical algorithms [22], especially the primal dual Mehrotra’s predictor-corrector method which is used in most IPM-based software packages [23, 24].
As mentioned in the previous section, the standard problem is expressed as:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax = b \quad \text{and} \quad x \geq 0.
\end{align*}
\] (4)

where \( A \) is an \( m \times n \) matrix, \( m \) is the number of constraints and \( n \) is the number of variables, \( x \) and \( c \) are \( n \)-dimensional row vectors, and \( b \) is an \( m \)-dimensional column vector. The dual problem for (4) is:

\[
\begin{align*}
\text{maximize} & \quad b^T \lambda \\
\text{subject to} & \quad A^T \lambda + s = c \quad \text{and} \quad s \geq 0.
\end{align*}
\] (5)

where \( \lambda \) is an \( m \)-dimensional column vector and \( s \) is an \( n \)-dimensional column vector.

Finding a solution to (4) and (5) is equivalent to finding a solution to the KKT (Karush-Kuhn-Tucker) conditions:

\[
\begin{align*}
A^T \lambda + s &= c, \quad (6.1) \\
Ax &= b, \quad (6.2) \\
XSe &= 0, \quad (6.3) \\
x \geq 0, s \geq 0. \quad (6.4)
\end{align*}
\]

where \( X = \text{diag}(x_1, x_2, \ldots, x_n) \), \( S = \text{diag}(s_1, s_2, \ldots, s_n) \), and \( e \) is an \( n \)-dimensional column vector of all ones.

Primal-dual methods replace the equation (6.3) by the parameterized equation \( XSe = \mu e \) leading to the following system:

\[
\begin{align*}
A^T \lambda + s &= c, \quad (7.1) \\
Ax &= b, \quad (7.2) \\
XSe &= \mu e, \quad (7.3) \\
x \geq 0, s \geq 0. \quad (7.4)
\end{align*}
\]

The solution for \( \mu > 0 \), denoted by \( (x(\mu), \lambda(\mu), s(\mu)) \) is called the \( \mu \)-centre of the primal-dual pair (4) and (5) and the set of \( \mu \)-centres gives the central path of (4) and (5). As \( \mu \) gets closer to zero, the equality (7.3) gets closer to the equality (6.3) and \( x(\mu) \) gets closer to the optimal solution. \( \mu \) is called the duality measure and it is defined as follows:

\[
\mu = \frac{1}{n} \sum_{i=1}^{n} x_i s_i = \frac{x^T s}{n}
\]
The steps are generated by applying a perturbed Newton methods to the three equalities in (7). All iterates \((x^k, \lambda^k, s^k)\) must satisfy (6.4) strictly and that’s the origin of the term interior point. The primal-dual step \((\Delta x, \Delta \lambda, \Delta s)\) is obtained from the following system [22]:

\[
\begin{bmatrix}
0 & A & 0 \\
A^T & 0 & I \\
0 & S & X
\end{bmatrix}
\begin{bmatrix}
\Delta \lambda \\
\Delta x \\
\Delta s
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
XSe - \sigma \mu e + r
\end{bmatrix}
\] (8)

where \(\sigma \in [0,1]\) and \(r\) is a perturbation term possibly chosen to improve proximity to the central path.

The quantity \(c^T x - b^T \lambda\) is called the duality gap. It is non-negative and becomes zero when the optimal solution is found; therefore, it makes the basis for IPMs termination tests.

The basic framework for primal-dual methods can be stated as follows:

\[
\begin{align*}
& \text{Given } (x^0, \lambda^0, s^0) \text{ with } (x^0, s^0) > 0; \\
& \text{Set } k \leftarrow 0 \text{ and } \mu_0 = \frac{(x^0)^T s^0}{n}; \\
& \text{repeat} \\
& \quad \text{Choose } \sigma_k \text{ and } r^k; \\
& \quad \text{Solve (8) with } (x, \lambda, s) = (x^k, \lambda^k, s^k) \text{ and } (\mu, \sigma, r) = (\mu_k, \sigma_k, r^k) \text{ to obtain } (\Delta x^k, \Delta \lambda^k, \Delta s^k); \\
& \quad \text{Choose step length } \alpha_k \in (0,1] \text{ and set} \\
& \quad \quad (x^{k+1}, \lambda^{k+1}, s^{k+1}) \leftarrow (x^k, \lambda^k, s^k) + \alpha_k (\Delta x^k, \Delta \lambda^k, \Delta s^k); \\
& \quad \quad \mu_{k+1} = \frac{(x^{k+1})^T s^{k+1}}{n}; \\
& \quad \quad k \leftarrow k + 1; \\
& \text{until some termination test is satisfied}
\end{align*}
\]

Algorithm 1. The basic framework for primal-dual methods (adapted from [22])

Primal-dual Mehrotra's predictor-corrector algorithm updates \(\mu\) dynamically and solves the system in (8) twice. In the predictor step, it computes the affine-scaling direction by solving (8) for the pure Newton step that is, setting \(r = 0\) and \(\sigma = 0\). In the corrector step, the algorithm uses the information from the predictor step to compute the centering direction by solving (8) again but with different values of \(r\) and \(\sigma\). If affine-scaling step makes good progress in reducing \(\mu\), \(\sigma_k\) is chosen small so that the step actually taken is quite close to this pure Newton step. Otherwise, more centering is enforced and a more conservative direction is followed by setting \(\sigma_k\) closer to 1.

Karypis, Gupta, and Kumar [25] used a number of data sets on a single core to analyze the relative execution time of each of the computational kernels in each iteration of modern IPMs. The analysis showed that the LP solver spends, on average, half of its time executing a sparse linear solver (Cholesky factorization). It also showed that matrix-matrix multiplication and forward and backward solvers usually take much less time for most LP instances.
2.3 The Cell BE processor

2.3.1 History

The Cell BE is a heterogeneous processor that has one main core (PPE) and 8 accelerator cores (SPEs) as shown in figure 2. It was designed by Sony, Toshiba, and IBM, an alliance known as STI [10]. The design and first implementation was carried out over a 4-year period starting from 2001. Its first commercial use was in the PlayStation3 (PS3) game console in 2005. Since then, Mercury Systems™ [26] has used the Cell processor in blades, conventional rack servers, and PCI Express accelerators. In 2006, IBM released the QS20 blade module made of two Cell processors and later it released an enhanced version called QS22 blade. QS22 uses a new version of the Cell processor called the PowerXCell 8i, which was introduced by IBM in 2008 and has an enhanced double precision performance.

2.3.2 The PPE

The main core (PPE) is a 64-bit Power processor with vector processing extensions and two levels of hardware managed caches, a 32 KiB L1 data cache and a 512 KiB L2 cache. In addition, it is a dual-issue, dual-threaded processor that has a single precision peak of 25.6 GFLOPS and a double precision peak of 6.4 GFLOPS. The PPE is usually responsible for running the operating system and controlling the other cores (SPEs); it can start, stop, interrupt, and schedule processes running on them. In fact, SPEs achieve their work only by following PPE commands. The PPE can read and write the main memory and the local memories of SPEs through the standard load/store instructions. However, data movement to and from an SPE (its local memory) is achieved explicitly using DMA commands which poses a major challenge to software development on the Cell BE processor.

2.3.3 The SPEs

The 8 SPEs are SIMD in-order cores that each of them possesses a 256 KiB local memory called local store (LS) for storing data and instructions, a 128 x 128 bit register file and a Memory Flow Controller (MFC). All SPE instructions operate on 128-bit SIMD vectors and are capable of performing operation on 16 8-bit integers, eight 16-bit integers, four 32-bit integers or single precision floating-point numbers, or two 64-bit double precision floating-point numbers [27]. Each SPU can dispatch and complete two instructions in each clock cycle using two pipelines referred to as the even and odd pipelines. Most of the floating and fixed-point instructions execute on the even pipeline, while most of the memory instructions execute on the odd pipeline. Therefore, and in order to utilize the SPU efficiently, the SPU code should balance pipeline concurrency, as much as possible, whether explicitly by the programmer or implicitly by the compiler. In addition, all single-precision operations are fully pipelined while the double precision operations are partially pipelined. Consequently, the single-precision peak (204.8 GFLOPS) of all SPEs is very high relative to the double precision peak (14.64 GFLOPS) [27].
2.3.4 Communication Architecture

As shown in figure 2, the EIB enables communication between the PPE, the SPEs and memory. It is implemented using a circular ring made of four unidirectional channels: two are running clockwise and two are running counterclockwise. The EIB operates at half the clock speed of the processor and allows each unit to simultaneously send and receive 16 bytes every bus cycle. The EIB theoretical peak data bandwidth is 204.8 GB/s.

Each SPE can directly access data and programs stored only in its local store. Therefore each SPE has its own DMA controller that allows moving data between the local store and the main memory and between the local store and other local stores. Moreover, the SPEs and the PPE can communicate using two simple low latency techniques which are mailboxes and signal notification. Mailboxes are special purpose registers available on the SPEs that are usually used to exchange small data with the PPE. Signal notification registers cannot be written by the SPEs and are therefore used by the SPEs to receive data.

2.3.5 PowerXCell 8i and usage in supercomputers

A revised variant of the Cell BE processor called the PowerXCell 8i was announced by IBM in 2008 and made available in IBM QS22 Blade servers. The SPEs in the new variant have a much better double precision floating-point peak performance (102.4 GFLOPS) compared to the previous one (14.64 GFLOPS). In addition, PowerXCell 8i has been used in several supercomputers such as Roadrunner [28] and QPACE [29]. Moreover, PowerXCell 8i based supercomputers occupied the first 6 slots in the he Green500's energy-efficient supercomputers in November 2009 [30].
2.3.6 Programming the Cell – challenges and programming models

The more the cores are busy, the more the computational power of the Cell processor is being exploited. However, this depends on the characteristics of the application being developed and especially the communication/computation ratio. The main memory peak transfer rate is 25.6 GB/s while the SPE has a peak performance of 25.6 GFLOPS in single precision. Therefore, to keep 8 SPEs work concurrently on vectorized data, \(8 \times 4 = 32\) operations have to be performed on each single precision value in order to hide the communication [31]. Another related challenge is the small size (256 KiB) of the SPE local storage (local store, LS), which needs to have both the code and the data. Consequently, the programmer is required to find out the best way to partition the data and/or the code, transfer it to local store and, in general, develop algorithms that are similar to cache management policies for cache based systems but tailored to the algorithms being implemented. The programmer also needs to utilize the vector processing capabilities of the SPEs which requires proper data manipulation and alignment.

Some high level programming models have been introduced with the goal of easy programming of the Cell BE and exploitation of its processing power. For example, Cell superscalar (CellSs) allows the programmer to write a sequential application and write annotations to specify which functions can be executed on an SPE [32]. A source to source compiler separates the annotated code and generates the necessary code so that the annotated functions get executed by the SPEs. To exploit parallelism, a task dependency graph is built by the CellSs runtime where each instance of the annotated function is represented by a node, and data dependencies are represented by edges between nodes.

Octopiler [33] is another programming model that provides the programmer with the abstraction of single shared-memory address space and allows him to use OpenMP directives to indicate parallel regions. In addition, the Octopiler compiler implements low level optimization techniques such as the execution of scalar code in SIMD units and the overlap of data transfers with computation.

2.4 AMD Opteron 2400-series (Istanbul)

AMD Opteron 2400-series processors, code named Istanbul, are the first 6-core AMD™ Opteron processors and are available for 2-, 4-, and 8-socket systems, with clock speeds ranging from 2.0 to 2.8 GHz. The Istanbul processor was introduced in June 2009 and is manufactured in a 45nm process and based on the AMD 64-bit K10 architecture. The K10 architecture supports the full AMD64 instruction set and SIMD instructions for both integer and floating point operations [34].

Figure 3-a shows a simplified block diagram. The processor has six cores, three levels of cache, a crossbar connecting the cores, the System Request Interface, the Memory controller, and three HyperTransport (HT) 3.0 links. The memory controller supports DDR2 memory with a bandwidth of up to 12.8 GB/s. In addition, the HyperTransport 3.0 links provide an aggregate bandwidth of 57.6 GB/s and are used to allow communication between different Istanbul processors. Each core has two levels of cache, a 512 KiB L2 cache, and 128 KiB L1 cache (64 KiB data cache and 64 KiB instruction cache). However, all cores share a 6 MiB L3 cache.
AMD Opteron multiprocessor systems are based on the cache coherent Non-Uniform Memory Access (ccNUMA) architecture. Each processor is connected directly to its own dedicated memory banks and it uses HT links to communicate with I/O buses and the other processor(s). Figure 3-b shows a block diagram of a 2-socket system.

In a multi-socket system that has 4 or more processors, a hardware technology called HT assist, or Probe Filter, is used to significantly reduce the latency of keeping cache coherency. HT assist uses 1 MiB of the L3 cache to store a map of all cache lines in the cores of a given processor. The main advantage is that checking a cache line takes the processor to do a Probe Filter Lookup instead of generating numerous cache probes. This causes a significant decrease in the local memory latency as probing caches on other processors is not required. In addition, it also increases system bandwidth by reducing probe traffic [11].

Figure 3-a: Simplified block diagram of an AMD Opteron Istanbul processor.
2.5 Using mixed-precision iterative refinement

Single floating point operations on modern architectures are usually at least twice as fast as double precision operations [35]. AMD Opteron 2 x 6, IBM PowerPC 970, and Intel Xeon 5100 are all examples where single precision peak is twice the double precision one. The single/double gap on some other architectures is much greater, for example, double precision peak is 14 times slower than single precision peak on the Cell BE processor.

For some applications, like most practical LP problems, single precision arithmetic lead to inaccurate results and numerical stability problems. As a result, such applications can not benefit from the high single precision processing power. Fortunately, Baboulin, and others [35] presented a methodology that can be used to obtain a 64-bit accurate solution for a system of linear equations while performing most of the computation in single precision. The main idea is to first obtain a 32-bit accurate solution and then refine it to become as accurate as if it was obtained using double precision arithmetic.

Algorithm 2 shows the algorithm presented by Baboulin for using mixed-precision for solving a sparse positive definite symmetric system of the form $Ax = b$ using Cholesky factorization. Step 1 represents Cholesky factorization of $A$ into $LL^T$ after applying permutation $P$ for reducing the fill-in produced in the factor $L$. Steps 2 and 3 (and steps 5 and 6) represent forward and backward solvers. Vector $x$ is the solution to the system and it gets refined through steps 4-7. Note that only steps 4 and 7 are performed in double precision while other steps are performed in single precision. It is important to mention that the overall performance depends on number of refinement iterations and the ability of single precision operations to utilize the available single processing power.
1. \( LL^T \leftarrow PA \) (S)
2. solve \( Ly = Pb \) (S)
3. solve \( L^T x_0 = Pb \) (S)
   \[ \text{do } k = 1, 2, \ldots \]
4. \( r_k \leftarrow b - Ax_{k-1} \) (D)
5. solve \( Ly = Pr_k \) (S)
6. solve \( L^T z_k = y \) (S)
7. \( x_k \leftarrow x_k + z_k \) (D)
   check convergence
   done

Algorithm 2: Mixed precision for Cholesky factorization.

2.6 Max flow with minimum lot sizes

Theory of flows makes a very important part of combinatorial optimization because its many applications. Maximum flow problems involves finding the greatest flow in a transportation network, or flow network, that can be transported from a single source to a single sink without violating the connections capacity constraints [36]. A transportation network is a directed graph where each edge has a maximum capacity and receives a flow. It can be used to model liquids flowing through pipes, parts through assembly lines, current through electrical networks, etc. [37].

In transportation systems, as well as in production and manufacturing, it is required that the flow lies above a given threshold. This could be due to different reasons like cost efficiency and the nature of some mechanical and chemical processes. When such a threshold is enforced on a flow network, the new problem is referred to as max flow with minimum lot sizes. The problem is very relevant to this PhD project as some MIRIAM AS customers require that gas/oil level should not be less than a certain quantity in each pipe. This is different from max flow problem where there are efficient algorithms for finding the max flow. Finding max flow with minimum lot sizes is proved to be NP-hard [38].
Chapter 3

Methodology

This chapter explains the methodology we adopt to make our experiments. The PhD project required developing a source code of an LP solver with various optimizations, and implementations of several algorithms for finding the max flow in a network. To speed up the development of the necessary programs, we used freely available source code when it was found appropriate. In the following sections we justify using IPMs and not simplex. We also explain the use of the GNU Linear Programming Kit (GLPK) solver, the experimental data sets, and the Boost graph library (BGL).

3.1 The LP solver

As mentioned before, an efficient LP solver is required for an efficient implementation of Regina. Regina uses a commercial LP solver called ILOG CPLEX [39]. Building an efficient stable LP solver from scratch is time consuming; therefore, we decided to look for an open source solver that we can use as a base for our experiments. Section 3.2 justifies looking for a solver that uses IPM and not simplex while section 3.3 justifies working with the GLPK package.

3.2 Simplex or IPM

The two main methods used to solve LP problems are simplex and IPMs [15]. Simplex has dominated the area of solving LP problems since its invention by Dantzig in 1949 and until the emergence of practical IPM in the 80s. Theoretically, IPM has the advantage of having a polynomial worst-case complexity while simplex is a non-polynomial algorithm [40]. However, simplex works well in practice and it is faster than IPM when solving some problems.

In the real world, both methods compete with each other and they are still in use [41]. The size and the sparsity of the problem are usually the main factors that decide which of the two methods is more efficient to use. In general and as indicated by Gondzio and Yarmish [41, 42], IPM is more efficient for solving large-scale problems. On the other hand, the sparsity structure of a problem may favour linear algebra operations of one method over the other [41]. In this context, Hall [43] developed techniques that enhance the performance of the revised simplex algorithm when solving hyper-sparse problems. Hyper-sparse problems are those where the result of solving linear systems or performing matrix vector product, the two linear algebra operations used in revised simplex, is sparse.

Some applications, like Regina, require solving a sequence of problems where two consecutive problems are slightly different from each other. For such problems, simplex performs better than IPM because it can solve a problem faster when it starts from the
solution of another slightly different problem, a strategy referred to as warm starting. Unlike simplex, IPM is known to have inefficient use of warm starts [44].

LP is, in general, difficult to parallelize because it is proven to be P-complete [45]. However, there has been a lot of research on how to parallelize both Simplex and IPM methods. Hall shows that it is hard to parallelize simplex when solving large sparse problems [19]. This has also been assured by Yarmish justifying why he chose to parallelize standard simplex for solving dense problems and avoided parallelizing revised simplex used for sparse ones, although most real-world problems are sparse [42]. On the other hand, IPM is much easier to parallelize. It requires a lower number of more expensive iterations than simplex and the major work in each iteration is to solve a structured linear equation system [46]. The success of parallelizing IPM comes mainly from the success of parallelizing the linear system [47, 48].

Simplex is still better than IPM utilizing warm starts when solving current (small) Regina problems; however, it is very difficult to parallelize when solving sparse problems as mentioned previously in this section. On the other hand, IPM is much more parallelizable and the main goal of this PhD project was to exploit the processing power of multi-core systems through parallelization. In fact, 16 PlayStation 3 machines were bought before the start of the PhD project and large data sets were expected to be provided by the company. For the above reasons, we chose to parallelize an IPM-based solver.

3.3 The GLPK LP solver

As we mentioned before, building a stable efficient LP solver requires a lot of work. Therefore, we decided to look for an open source well documented LP solver that we can modify for improved implementations in our context. In addition, the solver should be based on IPMs as we justified in the preceding section. Moreover, it should be able to store and solve sparse problems efficiently because Regina data sets are very sparse.

Benson [23] presents an overview of available IPM codes. He concluded that many of them share common features such as the use of the predictor corrector framework [49]. Among them we chose to use the GLPK (GNU Linear Programming Kit) solver, mainly because it is an open source that can be used commercially. The GLPK solver is also well documented and it gets continuous support and development. Beside this, a lot of the effort in this PhD project is focused on how to enhance the efficiency of Cholesky factorization, the most compute intensive task in IPM, which means that the research outcome can be easily applied to other IPM-based LP solvers.

3.4 NETLIB and Regina data sets

The company provided us with a few data sets that resemble current Regina customer problems shown in Table 1. Unfortunately, all the problems are too small to accelerate by parallelization and the company did not have problems that represent future Regina
problems. For this reason, we also conduct our experiments using mainly some Netlib data sets [50] and few miscellaneous data sets obtained from the BPMPD website [51].

NETLIB data sets are a collection of real-life LP examples from a variety of sources and they are widely used in LP research. The selected NETLIB datasets were usually big (relative to other NETLIB data sets). In addition, most relatively big data sets were profiled and the selection was made based on the results of profiling. Most data sets used in this PhD project were chosen because they spend most time doing Cholesky factorization. However, one of the papers required looking for data sets that spend most time doing matrix multiplication. Table 2 shows the NETLIB data sets and table 3 shows the BPMPD data sets that were used in this work.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size ( rows x columns)</th>
<th># of non-zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>gas.lp</td>
<td>68x75</td>
<td>141</td>
</tr>
<tr>
<td>dp_0.lp</td>
<td>35x38</td>
<td>74</td>
</tr>
<tr>
<td>dp_150.lp</td>
<td>35x35</td>
<td>71</td>
</tr>
<tr>
<td>dp_170.lp</td>
<td>35x36</td>
<td>72</td>
</tr>
<tr>
<td>dp_final.lp</td>
<td>330x321</td>
<td>724</td>
</tr>
<tr>
<td>SNEX_10_Years</td>
<td>312x296</td>
<td>674</td>
</tr>
</tbody>
</table>

Table1: Data sets provided by MIRIAM AS.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size ( rows x columns)</th>
<th># of non-zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>QAP12</td>
<td>3193 x 8856</td>
<td>44244</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>3137 x 9408</td>
<td>144848</td>
</tr>
<tr>
<td>DFL001</td>
<td>6072 x 12230</td>
<td>41873</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>2172 x 5167</td>
<td>35674</td>
</tr>
<tr>
<td>PILOT</td>
<td>1442 x 3652</td>
<td>43220</td>
</tr>
<tr>
<td>BNL2</td>
<td>2325 x 3489</td>
<td>16124</td>
</tr>
<tr>
<td>CYCLE</td>
<td>1904 x 2857</td>
<td>21322</td>
</tr>
<tr>
<td>TRUSS</td>
<td>1001 x 8806</td>
<td>36642</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1504 x 1818</td>
<td>26230</td>
</tr>
<tr>
<td>25FV47</td>
<td>822 x 1571</td>
<td>11127</td>
</tr>
<tr>
<td>PILOT-JA</td>
<td>941 x 1988</td>
<td>14706</td>
</tr>
<tr>
<td>D6CUBE</td>
<td>416 x 6184</td>
<td>43888</td>
</tr>
<tr>
<td>WOODW</td>
<td>1099 x 8405</td>
<td>37478</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>245 x 2594</td>
<td>70216</td>
</tr>
<tr>
<td>GREENBEB</td>
<td>2393 x 5405</td>
<td>31499</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>1152 x 5427</td>
<td>21597</td>
</tr>
<tr>
<td>CZPROB</td>
<td>930 x 3523</td>
<td>14173</td>
</tr>
<tr>
<td>STOCFOR3</td>
<td>16676 x 15695</td>
<td>74004</td>
</tr>
<tr>
<td>80BAU3B</td>
<td>2263 x 9799</td>
<td>29063</td>
</tr>
<tr>
<td>PILOT87</td>
<td>3608 x 8038</td>
<td>21322</td>
</tr>
<tr>
<td>FIT2D</td>
<td>21024 x 10525</td>
<td>150042</td>
</tr>
</tbody>
</table>

Table 2: NETLIB data sets used in published papers of this PhD project.
Table 3: BPMPD data sets used in published papers of this PhD project.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size ( rows x columns)</th>
<th># of non-zeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEMSEMM1</td>
<td>74151 x 5668</td>
<td>1036227</td>
</tr>
<tr>
<td>WORLD</td>
<td>79053 x 47259</td>
<td>220891</td>
</tr>
<tr>
<td>NSCT2</td>
<td>37563 x 23003</td>
<td>697738</td>
</tr>
<tr>
<td>BPMPD</td>
<td>1144020 x 33841</td>
<td>3450992</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>22977 x 11144</td>
<td>108562</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>14286 x 9872</td>
<td>596697</td>
</tr>
<tr>
<td>QAP15</td>
<td>22275 x 6330</td>
<td>94950</td>
</tr>
</tbody>
</table>

3.5 Boost C++ libraries – the Boost Graph Library

The Boost Graph Library (BGL) was used to save time developing data structures for representing graphs and implementing graph related algorithms. BGL provides a standardized generic interface for traversing graphs and encourage reuse of graph algorithms and data structures [52]. A part of BGL is a generic interface that allows access to a graph’s structure while hiding the details of the implementation. Any graph library that implements this interface will be interoperable with the BGL generic algorithms and with other algorithms that also use this interface.

The BGL algorithms consist of a core set of algorithm patterns, implemented as generic algorithms, and a larger set of graph algorithms. The algorithm patterns are merely building blocks for constructing graph algorithms. The core algorithm patterns are Breadth First Search, Depth First Search, and Uniform Cost Search. For our research, the shortest paths algorithms and the max flow algorithms were of great importance.
Chapter 4

Research Process

This chapter explains the way the PhD work was accomplished and the publication of six papers. The PhD work can be described as being adaptive taking into consideration rapid changes in multi-core market and company interests. In November 2009, around one year after the start of the PhD project, IBM announced that it cancelled its plans to update the Cell BE processor [53]. In addition, Kongull, a compute cluster at NTNU with 96 compute nodes (each with two 6-cores Opteron 2431 (Istanbul) processors) became available to be used for free for our project [54]. On the other hand, and almost around the same time, MIRIAM AS got new owners with more priority given to enhance the user interface of MIRIAM Regina and the way it can be accessed by its customers.

The whole PhD is about accelerating Regina, a gas flow simulator, so that it runs faster on multi-core systems. The main methodology used can be summarized by the following steps:
1. Understanding Regina and finding the most compute-expensive parts
2. Building an efficient LP solver
   a. Investigating the possibilities of efficient parallelization on multi-core systems
   b. Investigating the possibilities of better cache utilization on multi-core systems
3. Developing approximation algorithms/heuristics for NP hard computation parts.

4.1 Regina and its time critical parts

Regina spends most time solving a group of sequences of LP problems. Different sequences (called replications in MIRIAM AS terminology) are independent and are, therefore, executed in parallel. The parallel execution of these replications is trivial and done in the current version of the Regina application and therefore it is not discussed in this thesis. However, LP problems within one sequence (replication) are difficult to be executed in parallel because the results from one LP problem are used as input to solve the next LP problem. As a result, the major part of the PhD is focused on accelerating the solution of one LP problem on multi-core systems.

A feature of Regina that is required by some customers requires finding the maximum flow in the network subject to the constraint that the flow in each pipe is bounded by an upper and lower value. We found that this problem is NP hard which means that the time it takes to find a solution to it increases very quickly as the size of the problem grows [38]. The second major part of the PhD research is focused on developing algorithms/heuristics that approximate the solution of this problem and then parallelizing the heuristics on multi-core systems.
4.1.1 Regina and Linear Programming

Regina spends most time solving long sequences of LP problems. This means that the performance of Regina can be enhanced by accelerating the solution of one LP problem. The main topics to investigate were mainly the following:

1. different methods to solve an LP problem
2. parallelizing the solution of an LP problem

The ideal method to use for solving LP problems should match the following criteria:

1. Be parallelizable. This is the most important feature since the main goal of our project is to utilize the processing power of multi-core systems.
2. Can utilize the characteristics of the Regina LP problems.

Choosing the best method to use in Regina was not easy for two reasons. First, Regina was not well documented. Second, the characteristics, i.e. size and sparsity of the Regina real and future LP problems were not known.

4.2 Building an efficient LP solver

We didn’t build an LP solver from scratch; however, we chose to use an open source solver called GLPK to conduct our research. To be more precise, we used only the IPM-based solver of GLPK. We profiled the solver and developed more efficient versions by parallelizing its compute-intensive parts and by enhancing cache utilization. Parallel versions are implemented to run on both the Cell BE processor and the 2 x 6-core AMD Opteron processor while cache utilization was tested only on the Opteron processor.

Moving from the Cell BE processor to the AMD Opteron processor
The PhD project was adaptive to the changes in company interests, multi-core market, and available resources as described in the beginning of the chapter. Therefore, we used the Cell BE processor for the first three papers and then decided to move to another multi-core system to continue our research. For the last three papers, we used the AMD 2431 (Istanbul) processor to test our implementations.

4.2.1 IPM-based LP solver on the Cell BE processor – papers I, II, and III

Paper I: Implementation of a Linear Programming Solver on the Cell BE Processor
The first paper [55] describes parallelizing an IPM-based LP solver (part of the GLPK) on the Cell BE processor. The main idea is to parallelize Cholesky factorization, the most compute-critical part of the LP solver, based on the idea suggested by Rothburg and Gupta [56]. They suggested dividing the matrix to be factorized into a group of two-dimensional blocks and performing three matrix operations on them.

The blocks are kept in the main memory of the Power Processing Element (PPE) and are transferred to/from SPEs so that the SPEs perform the operations in parallel. One of the
operations is performed using three blocks, therefore, the size of one block is not allowed be larger than one third the size the local store of the SPE of the Cell BE processor.

To increase the portability and allow easy debugging of the solver, it has been implemented in two phases. The first phase is developing a multithreaded implementation that can execute on a traditional general purpose uni-processor that supports Pthreads [57]. The second phase starts from the first-phase to implement another version such that the computation of the matrix operations required for Cholesky factorization is performed by other cores. The same technique is used in the second and third paper.

**Paper II: Mixed-Precision Parallel Linear Programming Solver**

In paper II [58], the LP solver developed in the first paper is accelerated using the mixed precision technique to compute Cholesky factorization. This technique allows utilizing the relatively high single-precision computing power of the Cell BE processor to obtain results that are as accurate as if they were obtained using double precision arithmetic.

**Paper III: IPM based sparse LP solver on a heterogeneous processor**

Paper III [59] is also another enhancement to the LP solver on the Cell BE processor. The main contribution is a method to overcome the overhead of transferring small blocks between the main memory and the SPEs (local stores) and scheduling the small tasks associated with them. The paper suggests new methods to amalgamate thin supernodes and form bigger ones and eventually prevents the formation of many small blocks.

**4.2.2 Cache efficient IPM-based LP solver on 6-core AMD Opteron processors – paper IV**

**Paper IV: Cache-Aware Matrix Multiplication on Multi-core Systems for IPM-based LP Solvers**

Utilizing cache effectively can have a great impact on the performance of an application. The Rothberg and Gupta method [56] for parallelizing Cholesky factorization makes it also cache friendly since the sizes of the blocks can be chosen so that they fit into the cache. In paper IV [60], we show that the matrix multiplication used in IPM can take a considerable amount of time when solving some problems. We used the fact that the structure of the matrix is constant while solving an LP problem and suggested methods to enhance cache utilization. The methods are based on 1-dimensional and 2-dimensional partitioning of the matrices. We have also parallelized the multiplication and introduced a method to balance the load among the participating cores.

**4.3 Maximum flow with minimum lot sizes – papers V and VI**

MIRIAM Regina incorporates a flow model where processes can be operated in a semi-continuous range (either they are turned off or they are assigned a capacity above a non-zero bound). This model can be approached by computing the maximum flow under the constraint that the flow in each pipe is either zero or above a minimum value that is specified for each pipe.
**Paper V: The maximum flow problem with minimum lot sizes**
In paper V [38], we prove that computing the maximum flow with minimum lot sizes is an NP hard problem. Then we suggest and implement a heuristic technique for fast computation of a near optimal value.

**Paper VI: Parallel algorithms for the maximum flow problem with minimum lot sizes**
In paper VI [61], we prove that the maximum flow with minimum lot sizes is a strong NP-hard problem. In addition, we increased the efficiency of the heuristic by parallelizing it and testing it on the Opteron processor.
Chapter 5

Research results

This chapter presents an overview of the papers published during the PhD project. The chapter is composed of the abstracts and a retrospective view of each paper. All papers are included in the appendix section.

5.1 Paper I:

| Implementation of a Linear Programming Solver on the Cell BE Processor |
|-----------------------------------------------|---------------------|
| M. Eleyat and L. Natvig                      |
| Procedia Computer Science                     |
| 2010                                          |

5.1.1 Abstract

We describe an implementation of a parallel linear programming solver on the Cell BE processor. This implementation is based on GLPK C routines which solve LP problems using a serial implementation of one of the IPMs. We have identified the computational kernels of the serial version and decided to implement a parallel version of Cholesky factorization and integrate it into GLPK. Our decision stemmed from the fact that Cholesky factorization is the most computationally expensive kernel that has the most potential to parallelize efficiently on the Cell BE processor. Compared to the execution time of serial GLPK on the Cell Power Processing Element (PPE), we were able to obtain a speedup of up to 7 when solving some of the large size NETLIB problems on Sony’s PlayStation 3.

5.1.2 Retrospective view

The focus of the paper was to accelerate an IPM-based solver by parallelizing the factorization of a sparse linear system on the Cell BE. The main algorithm divides the matrix to be factorized into two-dimensional blocks, transfer blocks to the SPEs of the Cell processor and let them perform matrix operations on them. We explained two main reasons why only some of the data sets were accelerated. The first reason was that some problems don’t spend most time doing factorization. The second reason was the overhead of scheduling and transferring small-size blocks that were formed in some problems. We made measurements showing and justifying the first reason. However, we didn’t report any measurements to justify the second reason and it was only investigated in detail later in the third paper. In addition to that, PPE cache utilization was not investigated.
5.2 Paper II:

<table>
<thead>
<tr>
<th>Mixed-Precision Parallel Linear Programming Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>M. Eleyat and L. Natvig</td>
</tr>
<tr>
<td>22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)</td>
</tr>
<tr>
<td>2010</td>
</tr>
</tbody>
</table>

5.2.1 Abstract

We use mixed-precision technique, which is used to exploit the high single precision performance of modern processors, to build the first sparse mixed-precision linear programming solver on the Cell BE processor. The technique is used to enhance the performance of an LP IPM-based solver by implementing mixed-precision sparse Cholesky factorization, the most time consuming part of LP solvers. Moreover, we implemented sparse matrix multiplication of the form required by the solver as it is also very time consuming for some LP problems. Implemented on the Cell BE processor (Playstation 3) and tested using NETLIB data sets, our LP solver achieved a maximum speedup of 2.9 just by using the mixed-precision technique. Moreover, we found that some problems, especially in final iterations, result in ill-conditioned matrices where mixed-precision cannot be used. As a result, the solver needs to switch to double precision if a more accurate solution of an LP problem is required.

5.2.2 Retrospective view

The main goal of this paper was to use the mixed-precision technique to utilize the very high single-precision performance of the Cell BE processor relative to its double precision performance. In addition, storing numbers using single precision format and transferring it between the main memory and the local store of the Cell BE processor makes better use of the available bandwidth and the relatively small-size local store. Because IPM-based LP solvers usually spend most time performing Cholesky factorization, we modified the LP solver developed in the first paper so that it uses this technique to perform Cholesky factorization and consequently obtain the first mixed-precision LP solver. We succeeded to accelerate the solution of some problems. However, some problems could not be accelerated and some could not be solved to the same accuracy of double precision. The problem of insufficient accuracy could have been solved by switching to double precision to perform the last few iterations but we didn’t implement this. In addition, we achieved a better understanding of the overhead of small-size blocks which was the main idea of the third paper.
5.3 Paper III:

<table>
<thead>
<tr>
<th>IPM based sparse LP solver on a heterogeneous processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>M. Eleyat and L. Natvig</td>
</tr>
<tr>
<td>Computational Management Science</td>
</tr>
<tr>
<td>2012</td>
</tr>
</tbody>
</table>

5.3.1 Abstract

We present the parallelization of a linear programming solver using a primal-dual IPM on one of the heterogeneous processors, namely the Cell BE processor. Focus is given to Cholesky factorization as it is the most computationally expensive kernel in IPMs. To make it easier to develop and port to other heterogeneous systems, we propose a two-phase implementation procedure where we first develop a shared-memory multithreaded application that executes only on the main processor, and then offload the compute-intensive tasks to execute on the synergistic processors (Cell accelerator cores). We used parent–child supernode amalgamation to increase sizes of the blocks, but we noticed that the existence of many small blocks cause significant performance degradation. To reduce the overhead of small blocks, we extend the block fan-out algorithm such that small blocks are aggregated into large blocks without adding extra zeros. We also use another type of amalgamation that can merge any two consecutive supernodes and use it to avoid having very small blocks in a composed block. The suggested block aggregation method is able to speedup the whole LP solver of up to 2.5 when compared to using parent–child supernode amalgamation alone.

5.3.2 Retrospective view

The main goal of the work in this paper was to reduce the overhead of scheduling and transferring small sized blocks that we noticed in papers I and II. We suggest two methods for building bigger blocks out of small blocks. One of the methods, that we called blind amalgamation, stores more extra zeros than traditional amalgamation to amalgamate small blocks. The other method, which we called block aggregation, does not store extra zeros, however, it stores extra metadata necessary to retrieve small blocks and perform operations on them. We showed that the best is to use a hybrid method where the first method eliminates the existence of very small blocks and the second method aggregate the resulted blocks without the need to store much metadata. Our methods were successful when tried on the Cell BE processor. Future work should include testing the scalability of our methods using higher number of cores because our methods do not take block dependency into consideration and, therefore, are expected reduce the degree of parallelization.
5.4 Paper IV:

<table>
<thead>
<tr>
<th>Cache-Aware Matrix Multiplication on Multicore Systems for IPM-based LP Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>M. Eleyat, L. Natvig and J. Amundsen</td>
</tr>
<tr>
<td>Federated Conference on Computer Science and Information Systems (FedCSIS)</td>
</tr>
<tr>
<td>2011</td>
</tr>
</tbody>
</table>

5.4.1 Abstract

We profile GLPK, an open source linear programming solver, and show empirically that the form of matrix multiplication used in IPMs takes a significant portion of the total execution time when solving some of the NETLIB and other LP data sets. Then, we discuss the drawbacks of the matrix multiplication algorithm used in GLPK in terms of cache utilization and use blocking to develop two cache aware implementations. We apply OpenMP to develop parallel implementations with load balancing. The best implementation achieved a median speedup of 21.9 when executed on a 12-core AMD Opteron.

5.4.2 Retrospective view

IPM-based LP solvers usually spend most time performing Cholesky factorization when solving most data sets, however, we noticed that they spend more than 50% of the execution time performing matrix multiplication when solving some data sets. The idea of this paper was to enhance the performance of IPM-based LP solvers by developing cache-aware algorithms for the form of the matrix multiplication used in IPM-based LP solvers. We have also used OpenMP to parallelize our algorithms and developed a simple but efficient load balancing algorithm. Our code was tested on AMD Opteron 2 x 6 processor but was compiled to run as a 32-bit application as it was the default compilation parameter in our development IDE. We recompiled the code to run as a 64-bit application and tested it using two data sets and did not notice big difference; therefore, we decided to keep the 32-bit results. Using 32-bits code should have been explicitly stated in paper. Later on after we published our paper, Kristoffer Stensen used our code and noticed that compiling using 64-bit make a considerable difference in the parallel results when solving some data sets [62].

5.4.3 Errata for Paper IV

It should have been noted in the paper that the execution times in Figures 5 to 9 are unintentionally for two iterations. However, those values were correctly divided by 2 before calculating the speedup results, that consequently are correct in the paper.
5.5 Paper V:

### The maximum flow problem with minimum lot sizes

D. Haugland, M. Eleyat and M. Hetland

Second International Conference on Computational Logistics, ICCL

2011

#### 5.5.1 Abstract

In many transportation systems, the shipment quantities are subject to minimum lot sizes in addition to regular capacity constraints. That is, either the quantity must be zero, or it must be between the two bounds. In this work, we consider a directed graph, where a minimum lot size and a flow capacity are defined for each arc, and study the problem of maximizing the flow from a given source to a given terminal. We prove that the problem is NP-hard. Based on a straightforward mixed integer programming formulation, we develop a Lagrangian relaxation technique, and demonstrate how this can provide strong bounds on the maximum flow. For fast computation of near-optimal solutions, we develop a heuristic that departs from the zero solution and gradually augments the set of flow-carrying (open) arcs. The set of open arcs does not necessarily constitute a feasible solution. We point out how feasibility can be checked quickly by solving regular maximum flow problems in an extended network, and how the solutions to these subproblems can be productive in augmenting the set of open arcs. Finally, we present results from preliminary computational experiments with the construction heuristic.

#### 5.5.2 Retrospective view

This paper was different from first four papers because it did not try to enhance the performance of an LP solver. However, its goal was to develop heuristics to find the maximum flow in a network under the constraint that the flow in each pipe is either above a certain value or zero but, of course, not exceeding the capacity of the pipe. The problem was proved to be NP-hard and so finding exact solutions takes relatively long time. Developing a fast heuristic algorithm is an improvement to MIRIAM Regina because it can be a fast more accurate replacement to its current algorithm.
5.6 Paper VI:

**Parallel algorithms for the maximum flow problem with minimum lot sizes**

M. Eleyat, D. Haugland, M. Hetland and L. Natvig

Operations Research Proceedings

2011

5.6.1 Abstract

In many transportation systems, the shipment quantities are subject to minimum lot sizes in addition to regular capacity constraints. This means that either the quantity must be zero, or it must be between the two bounds. In this work, we prove that the maximum flow problem with minimum lot-size constraints on the arcs is strongly NP-hard, and we enhance the performance of a previously suggested heuristic. Profiling the serial implementation shows that most of the execution time is spent on solving a series of regular max flow problems. Therefore, we develop a parallel augmenting path algorithm that accelerates the heuristic by an average factor of 1.25.

5.6.2 Retrospective view

In this paper, we try to parallelize the heuristic we developed in the previous paper. The main challenge was that most work is spent solving a series of small regular max flow problems. The implementation was tested on 2 x 6-core AMD Opteron processor and little speedup is achieved.
Chapter 6

Concluding remarks

6.1 Conclusion

This thesis has investigated the use of multi-core systems to enhance the performance of Miriam Regina, a time-consuming gas flow simulator. The main research activities were focused on accelerating an IPM-based LP solver and accelerating finding the max flow in a network with minimum lot sizes.

Because Miriam Regina spends most time solving LP data sets, we have studied the main characteristics of the main LP methods and decided to use IPMs mainly because of being parallelizable, although Simplex is more efficient than IPMs solving a series of similar related problems. Accelerating an IPM-based solver is achieved by mainly accelerating Cholesky factorization of sparse matrices and matrix multiplication.

Cholesky factorization of sparse matrices is harder to accelerate than that for dense matrices because it has to consider sparsity issues. Sparse matrices are stored using special storage formats that store only non-zero elements in order to save storage space, reduce communication, and avoid performing operations on zero values. In case of factorizing dense matrices, it is relatively easy to divide the matrix into equal sized blocks, distribute them to multiple cores and consider load balancing among the cores. However, it is much harder to divide a sparse matrix into blocks while taking dependency of associated tasks and load balancing into consideration. In addition, blocks could be very small resulting in high overhead of scheduling and distributing them among the cores. To avoid the creation of too many small blocks, enhanced amalgamation methods are suggested and implemented.

Acceleration of Cholesky factorization on modern computer systems can also be achieved by using the mixed-precision technique. This technique works well on modern systems because their high performance of performing single-precision floating point operations relative to that of double precision operations. However, last iterations of solving an LP problem may result in factorizing ill-conditioned matrices which leads to less accurate factorization and requires switching to double precision arithmetic.

Accelerating an LP solver may also be achieved by accelerating the form of sparse matrix multiplication used in IPMs especially for some LP data sets that spend relatively long time performing the multiplication. It is hard to accelerate general sparse matrix-matrix multiplication; however, multiplication in IPMs is performed each iteration and the structure of the result matrix is fixed through all iterations. These facts allow using blocking to develop successful cache-aware serial and load balanced parallel variants of the original serial multiplication algorithm used in GLPK.

For some customer requirements, Miriam Regina needs to find the max flow in a network under the constraint that the flow in each pipe must be above a specific value or zero. This
task is proven to be NP-hard and; therefore, the time it takes to find an exact solution increases very quickly with problem size. Serial and parallel versions of a heuristic algorithm are proposed and tested.

There were several challenges influencing important design choices during this project. Realistic/representative large data sets were expected to be provided by the company, but only small data sets were provided. Moreover, we did not get access to detailed documentation of MIRIAM Regina and we could not have access to the complete application for profiling although that was supposed to be done as the first task in the project description. The Regina source code was obtained in the last year of the project but without an ILOG CPLEX license, so it could not be used for execution profiling or running experiments. A major obstacle for parallelization was the fact that the LP problems need to be solved one after the other in a series. This was not known before midway in the project.

6.2 Contributions

The main research question of this thesis is:

**How to accelerate the Regina gas flow simulator on multi-core systems.**

Because Miriam Regina spends most time solving a series of LP data sets and we chose IPMs to solve LP problems, most of the research was about accelerating an IPM-based LP solver leading to the sub research questions 1 and 2. Moreover, some customers require finding the maximum flow under certain constraints which is also a time consuming task leading to the sub research question 3.

1. **How to accelerate an IPM-based LP solver**

   We approach this question as follows:
   
   a) Building in paper I a parallel IPM-based solver to execute on the Cell BE processor. The main work was to parallelize Cholesky factorization processor based on the idea suggested by Rothburg and Gupta. The idea is to divide the matrix to be factorized into two-dimensional blocks and perform operations on them using the Cell SPEs based on their dependency. The parallelization took into account porting the solver easily to other multi-core systems.
   
   b) In paper II, the performance of the LP solver was enhanced by performing Cholesky factorization using the mixed precision technique used to utilize the high performance of single-precision arithmetic operations relative to that of double precision arithmetic operations.
   
   c) In the third paper, the performance of the LP solver was improved by reducing the overhead of scheduling and executing small tasks associated with small-sized blocks. For that purpose, two supernodes amalgamation methods were proposed and tested.

2. **How to accelerate sparse matrix multiplication in IPM.**

   We approach this question in paper IV by analyzing empirically the performance of matrix multiplication involved in an IPM-based LP solver. Blocking is applied to develop two
cache aware implementations of the multiplication. Moreover, a load-balanced parallel implementation has also been provided utilizing the fact that the structure of the output matrix is fixed throughout all IPM iterations.

3. **How to accelerate finding the maximum flow in a network with minimum lot sizes.**

We approach this question in Paper V by developing a heuristic algorithm for finding the maximum flow in a network under the constraint that the flow in each pipe must be zero or above a specific value. The heuristic requires solving a series of regular (without constraints) max flow problems. In paper VI, we enhanced the performance by providing parallel implementations of the heuristic algorithm built in paper V.

### 6.3 Future Work

The current version of Regina solves a series of small related data sets making a serial simplex-based solver that exploits warm starts be the most appropriate solver to be used. However, the future version of Miriam Regina is expected to simulate multiple gas networks across several countries, and therefore, future data sets are expected to be much larger and may have different characteristics. If future data sets are dense, a parallel standard simplex solver could be implemented. However and more likely, they will be large and still sparse. In this case, a comparison could be made between the gain obtained using a serial revised simplex-based solver that exploits warm starts and a parallel IPM-based LP solver.

Miriam AS may also consider replacing the IBM CPLEX solver with an open source Simplex-based solver like the CLP solver. In fact, we provided CLP with the API required to store the model, utilize warm starts, and solve the instances in a way similar to CPLEX so that switching to CLP can be performed without doing major changes to the CPLEX-based implementation.

### 6.4 Bibliography


47. Gondzio, J. and A. Grothey, *Direct Solution of Linear Systems of Size 10^9 Arising in Optimization with Interior Point Methods*, in *Parallel Processing and Applied


The NETLIB LP Test Problem Set. Available from: http://www.numerical.rl.ac.uk/cute/netlib.html


Paper I

Implementation of a linear programming solver on the Cell BE processor

M. Eleyat and L. Natvig
Procedia Computer Science, 2010
Abstract

We describe an implementation of a parallel linear programming solver on the Cell BE processor. This implementation is based on GLPK C routines which solve LP problems using a serial implementation of one of the interior point methods. We have identified the computational kernels of the serial version and decided to implement a parallel version of Cholesky factorization and integrate it into GLPK. Our decision stemmed from the fact that Cholesky factorization is the most computationally expensive kernel that has the most potential to parallelize efficiently on the Cell BE processor. Compared to the execution time of serial GLPK on the Cell Power Processing Element (PPE), we were able to obtain a speedup of up to 7 when solving some of the large size Netlib problems on Sony's PlayStation 3.
I.1 Introduction

Linear programming (LP) is a mathematical technique used to solve an abundance of problems within science and engineering, as well as commercial and transportation applications. It is used to find values for variables that maximize or minimize a certain objective function while satisfying a set of equality and/or inequality constraints [1]. The GLPK (GNU Linear Programming Kit) package is a set of ANSI C routines contained into a callable library and intended for solving large-scale linear programming, mixed integer programming, and other related problems [2]. GLPK has routines for solving LP problems using either simplex or one of the primal-dual interior point methods (IPMs), namely the Mehrotra’s predictor-corrector method [3].

Mehrotra’s predictor-corrector method, as well as other primal-dual interior point methods, keeps repeating the same set of matrix operations, until it converges to the optimal solution. Main computation includes sparse Cholesky factorization, sparse matrix-matrix multiplication, and backward/forward solving. However, we have decided to parallelize Cholesky factorization only because it is the most computationally expensive kernel [15]. Moreover, its efficient parallelization has been the focus of much research and led to techniques that’s allows utilizing cache and vector processors. On the other hand, the other kernels represent sparse matrix operations with low computation-to-communication ratios, and therefore they have less potential to parallelize efficiently on the Cell processor [9].

In this paper, we implemented a parallel version of GLPK on the Cell BE processor by parallelizing Cholesky factorization based on the algorithm suggested by Rothberg and Gupta [13]. We adapted their algorithm, which divides the matrix to be factorized into a set of blocks that can utilize cache and SIMD processors, to meet the specific features of the Cell BE processor, and implemented all the code required to integrate with GLPK and maintain its level of stability.

This paper is organized as follows: Section 2 gives an overview of the Cell BE architecture. Then, primal dual Interior Point Methods (IPM) and its main computational tasks are introduced in section 3. Section 4 describes Cholesky factorization and its parallelization while section 5 discusses the stability of parallel GLPK. We present related work in section 6 and conclude with experimental results and future work.

I.2 The Cell BE Architecture

The Cell processor architecture is shown in Figure 1 It is mainly composed of one power processing element (PPE), 8 synergistic processing elements (SPEs), an on-chip memory controller, and a controller for a configurable I/O interface — all linked together by an element interconnection bus (EIB) [5].

The PPE is a 64-bit Power processor that has two levels of caches, a 32 KB data Level 1 cache and a 512 KB Level 2 cache. In addition, it is a dual-issue, dual-thread processor that has a single precision peak of 6.4 GFLOPS and a double precision peak of 25.6 GFLOPS.
The PPE is usually responsible for running the operating system and providing application control.

SPEs are SIMD cores which each possesses a 256 KB local store for storing both data and code, a 128 x 128-bit register file and Memory flow controller (MFC). MFC has the capability to move code and data between main memory and local stores using the direct memory access (DMA) technique. Each SPE has a single precision peak of 25.6 GFLOPS and double precision peak of only 1.83 GFLOPS.

The EIB is composed of 4 unidirectional rings that are used as a communication bus between the different elements that are connected to it. The bus can deliver a 25.6 GB/s to each connected element.

The Cell processor has to have a high speed memory and I/O system which is necessary to feed the other units. As shown in the figure, a memory interface controller (MIC) is used to connect a dual-channel Rambus Extreme Data Rate (XDR) memory which can deliver a bandwidth of 25.6 GB/s. In addition, the Cell has a high-bandwidth configurable I/O interface, called FlexIO interface (labeled as I/O in the figure), which can be dedicated to up to two separate logical interfaces [10]. These interfaces provide all chip-to-chip connections and can be used to design an efficient dual-processor system.

![Cell Processor Architecture](image)

**Figure 1: The Cell Processor Architecture (from [9])**

**I.3 Mehrotra’s Predictor-Corrector Method:**

A linear programming problem can be expressed in the following standard form [11]:

\[ \text{Minimize } c^T x \text{ subject to } Ax = b, \ x \geq 0. \]
Where \(c, x \in \mathbb{R}^n\), \(b \in \mathbb{R}^m\), \(A\) is an \(m \times n\) real matrix and \(c^\top\) is the transpose of the vector \(c\).

The dual problem is

\[
\text{Maximize } b^\top y \text{ subject to } A^\top y + z = c, \ z \geq 0,
\]

Where \(y \in \mathbb{R}^m\), \(z \in \mathbb{R}^n\) and \(b^\top, A^\top\) denote transpose of the vector \(b\) and the matrix \(A\), respectively.

The main algorithm of Mehrotra’s predictor-corrector method that is used in GLPK is shown in figure 2. The method starts by generating a starting point and keeps iterating until the residuals (step 2) are less than an input parameter \((10^{-8}\) in GLPK). In each iteration, a sparse symmetric positive definite (SSPD) system of linear equations is solved (steps 3 and 6) to compute some increments that are added to the current solution (Step 8). Moreover, generation of the SSPD system involves sparse matrix-matrix multiplication.

**Step 1.** choose initial point \((x^0, y^0, z^0)\) using Mehrotra’s heuristic.

**Step 2.** calculate relative primal infeasibility \((rpi)\), relative dual infeasibility \((rdi)\), and primal-dual gap

\[
\begin{align*}
\text{rpi} & = \frac{||Ax - b||}{||b|| + 1}, \\
\text{rdi} & = \frac{||A^\top y + z - c||}{||c|| + 1}, \\
\text{gap} & = \frac{|c^\top x - b^\top y|}{1 + |c^\top x|},
\end{align*}
\]

if \(rpi < 10^{-8}, rdi < 10^{-8}\), and \(gap < 10^{-8}\), \(x\) is the optimal solution. Stop.

**Step 3.** Compute the affine scaling (predictor) direction by solving the following system with respect to \((\Delta x^{aff}, \Delta y^{aff}, \Delta z^{aff})\).

\[
\begin{align*}
A \Delta x^{aff} & = b - Ax \\
A^\top \Delta y^{aff} + \Delta z^{aff} & = c - A^\top y - z \\
Z \Delta x^{aff} + X \Delta z^{aff} & = -XZe
\end{align*}
\]

where \(Z = \text{diag}(z_1, ..., z_n), X = \text{diag}(x_1, ..., x_n), e = (1, ..., 1)^T\)

**Step 4.** calculate the measure of duality \(\mu = \frac{x^\top z}{n}\)

**Step 5.** compute the centering parameter \(\sigma\)

\[
\sigma = \left(\frac{\mu_{aff}}{\mu}\right)^3 \quad \text{Where}
\]

\[
\mu_{aff} = \frac{1}{n} (x + \alpha_{aff}^{pri} \Delta x^{aff})(z + \alpha_{aff}^{dual} \Delta z^{aff})
\]

\[
\alpha_{aff}^{pri} = \max \{\alpha \in [0, 1]: x + \alpha \Delta x^{aff} \geq 0\} = \min \left[1, \left\{\frac{x_i}{\Delta x_i^{aff}}, \Delta x_i^{aff} < 0\right\}\right]
\]

\[
\alpha_{aff}^{dual} = \max \{\alpha \in [0, 1]: z + \alpha \Delta z^{aff} \geq 0\} = \min \left[1, \left\{\frac{z_i}{\Delta z_i^{aff}}, \Delta z_i^{aff} < 0\right\}\right]
\]
**Step 6.** compute the centering (corrector) direction by solving the following system

\[ A\Delta x^\text{cor} = 0 \]
\[ A^T \Delta y^\text{cor} + \Delta z^\text{cor} = 0 \]
\[ Z\Delta x^\text{cor} + X\Delta z^\text{cor} = \sigma \mu e - XZe \]

**Step 7. Compute**

\[ (\Delta x, \Delta y, \Delta z) = (\Delta x^\text{aff}, \Delta y^\text{aff}, \Delta z^\text{aff}) + (\Delta x^\text{cor}, \Delta y^\text{cor}, \Delta z^\text{cor}) \]

\[ \alpha_\text{max} = \max \{ \alpha \geq 0 : x + \alpha \Delta x \leq 0 \} = \min \left[ 1, \left\{ \frac{x_i}{\Delta x_i} \right\} \leq 0 \right] \]

\[ \alpha^\text{dual} = \max \{ \alpha \geq 0 : z + \alpha \Delta z \leq 0 \} = \min \left[ 1, \left\{ \frac{z_i}{\Delta z_i} \right\} \leq 0 \right] \]

**Step 8. Compute next point** \( x_{\text{new}}, y_{\text{new}}, z_{\text{new}} \) (x,y,z for next iteration)

\[ x_{\text{new}} = x + 0.9 \alpha^\text{max} \Delta x \]
\[ y_{\text{new}} = y + 0.9 \alpha^\text{max} \Delta y \]
\[ z_{\text{new}} = z + 0.9 \alpha^\text{max} \Delta z \]

Figure 2: Mehrotra’s predictor-corrector method

### I.4 Parallel Sparse Cholesky Factorization

#### I.4.1 Sparse Cholesky factorization

Solving the SSPD system is the most computationally expensive kernel and that’s why most attention has been given to have a parallel efficient version of this kernel. A SSPD system can be solved by either a direct method such as Cholesky Factorization, or an iterative method such as Conjugate Gradient, however, we chose to use Cholesky since it is the method being used in GLPK and because there has been a lot of research on parallel sparse Cholesky factorization.

Cholesky factorization of a symmetric positive definite matrix \( M \) is the process of factoring it into the product of triangular matrices \( M = LL^T \). The four basic steps of this method are [12]:

a. **Ordering:** reordering of rows and columns so that the Cholesky factor \( L \) has less fill (fill are the non-zeros in \( L \) that were zeros in \( M \)).

b. **Symbolic factorization:** setting up, in advance, a data structure to accommodate the non-zero elements including the fill in.

c. **Numeric Factorization,** computing the numeric entries of the Cholesky factor. This step is by far the most expensive one.

d. **Triangular solution:** Computing the solution by forward and back substitution.

The pseudo-code of the serial implementation of Cholesky factorization (Step c above) is shown in figure 3.
1. for $k = 1$ to $n$ do
2.     for $i = k$ to $n$ do
3.         $L_{ik} := L_{ik} / \sqrt{L_{kk}}$
4.     for $j = k+1$ to $n$ do
5.         for $i = j$ to $n$ do
6.             $L_{ij} := L_{ij} - L_{ik}L_{jk}$

**Figure 3: Serial Cholesky Factorization**

Lines 2 and 3 in the above algorithm are usually expressed as cdiv($k$), dividing column $k$ by the square root of the $k$’th diagonal element, while lines 5 and 6 are expressed as cmod($j,k$), modification of column $j$ by column $k$. These expressions are the base for several formulations of parallel sparse column oriented Cholesky factorization which allocates different columns to different processors.

### I.4.2 Sparse Block Cholesky factorization

The authors in [13] presented an alternative parallel formulation which divides the matrix into a group of two-dimensional blocks in a way that leads to a more reduced communication volume and allows more parallelism. The key to decompose the matrix into blocks is based on the concept of a supernode, which is a set of contiguous columns of $L$ whose non-zero structure form a dense triangular block on the diagonal and share the same non-zero structure below the diagonal. The matrix is divided vertically into a set of partitions such that columns belong to a partition belongs to one supernode. Rows are partitioned the same way as columns, i.e. if columns $c_n$, $c_{n+1}$…,$c_m$ belong to a partition, then rows $r_n$, $r_{n+1}$…,$r_m$ belong to the corresponding row partition. Blocks are the intersections between horizontal and vertical partitions. Rothberg and Gupta have referred to this method as global partitioning guided by supernodes [13]. Figure 5 (a) taken from [15] shows an example.

After forming the blocks, Serial block factorization can be accomplished by applying several operations to the blocks as shown in figure 4 [13]. The algorithm is analogous to the one shown in figure 3 but it manipulates columns of blocks instead of columns of individual elements.

Implementation of sparse block factorization requires implementing several matrix operations including block (matrix) factorization (BFAC), block division by the inverse of a diagonal block (BDIV), and block modification (BMOD) which are, respectively, the lines 2, 4, and 7 in figure 4. The resulted diagonal blocks are dense blocks, while the other blocks are composed of dense rows which facilitate their storage and allow better utilization of vector processors.
1. for \( k = 1 \) to \( n \) do
2. \( L_{kk} := \text{Factor}(L_{kk}) \)
3. for \( i = k + 1 \) to \( N \) with \( L_{ik} \neq 0 \) do
4. \( L_{ik} := L_{ik}(L_{kk})^{-1} \)
5. for \( j = k + 1 \) to \( N \) with \( L_{jk} \neq 0 \) do
6. for \( i = j \) to \( n \) with \( L_{ik} \neq 0 \) do
7. \( L_{ij} := L_{ij} - L_{ik}(L_{jk})^{T} \)

Figure 4: Serial Block factorization

I.4.3 Parallel Sparse Block Cholesky Factorization

Parallelism can be revealed by creating a data structure called “the supernodal elimination tree” where each node represents a supernode. A parent supernode is the one which has the diagonal element that has the same row index of the first non diagonal element of the most right column of a child supernode. Figure 5 (b) shows an example of a supernodal elimination tree. Despite that other structures may show more fine-grained level of parallelism, we focus on the supernodal tree as it reveals course-grain parallelism that is used in our implementation. For each parent supernode (column of blocks), the children represent the supernodes that are used to modify that parent supernode (step 7 of figure 4). On the other hand, more concurrency it obtained by the fact that some blocks have no zeros elements at all. In fact, the number of operations that should be applied to each block is calculated taking into consideration the existence of zero blocks.

We implemented block sparse Cholesky factorization on the Cell processor using the block fan-out algorithm that is suggested in [13]. Before factorization, each block is assumed to have a specific processor owner. A processor will be responsible for performing all block modifications to the blocks it owns. Assuming that processors are arranged in a 2D \((p_r \times p_c)\) grid where the bottom left processor labeled \(p_{0,0}\) and the top right processor labeled \(p_{r-1,c-1}\), block \(B_{ij}\) is mapped to (or owned by) \(p_{i \% r, j \% c}\), where \% is the modulus operation. In other words, a row of blocks is mapped to a row of processors and a column of blocks is assigned to a column of processors. More details about mapping blocks to processors in a way that reduces communication can be found in [13].
The main idea of the block fan-out algorithm is that when a block has received all block modifications and multiplied by the inverse of the diagonal block, it is sent to all processors that own blocks that could be modified by it. The receiver processor performs all related modifications to the blocks it owns. When all block modifications to certain block are done, the block is either factorized if it is a diagonal block or multiplied by the inverse of the diagonal block and then sent to other processors as specified before.

Several data structures are needed to implement this algorithm [5]: First, a queue called task queue, is maintained by each processor so that it can receive blocks and react upon. Second, two data structures, BMODQ and BDIVQ are created for each column of the matrix. BDIVQ(k) indicates the blocks in column k that need to be divided by the inverse of the diagonal block, while BMODQ(k) has the blocks in column k that have received all block modifications and can participate in modifying other blocks.

Each processor starts by searching its share of blocks for diagonal blocks that are ready to be factorized (have zero block modification operations), factorize them, and send them to the task queues of other processors that may use them. Then each processor starts a loop fetching blocks from its task queue, performs all possible operations that can be done to the blocks it has, and send blocks to other processors when they are ready as specified earlier. Figure 6 summarizes the algorithm for one processor, readers can see [13] for a more detailed one.

I.4.4 Adapting Parallel Sparse Block Cholesky Factorization to the Cell BE Processor

The Cell BE processor has its own specific features that should be taken into account to better utilize its processing power. Being a heterogeneous multi-core processor, it leads the programmer to think of tasks that best fit the PPE and tasks that are more efficient to be executed by the SPEs. In addition, SPEs don’t have caches but small size local stores (256 KB) that are completely managed by the programmer. Moreover, it is difficult to utilize the
high processing power of the SPEs unless there are enough operations to be executed on each transferred value, for example 24 operations need to be performed on each single precision value in order to hide the communication [9].

To adapt the algorithm in Figure 6 to the Cell BE processor, POSIX threads are launched on the PPE as representatives of a processors (SPEs) participating in the factorization. The algorithm in Figure 6 is not executed by SPEs, however, it is executed by PPE threads which in turn offload the block operations to the SPEs. Once a task is offloaded to an SPE, the PPE thread is blocked waiting a mailbox notification message from the SPE that it has finished the task. Data blocks' sizes are adjusted to best fit in and utilize the SPE local store by splitting large supernodes and amalgamating [19] small (thin) supernodes. This method frees the SPEs from exchanging data with each other and accessing task queues and other data structures that are stored in main memory. In addition, it opens the door for investigating other methods of task scheduling taking into consideration load balancing among SPEs.

for each diagonal block $b_{j,j}$ in my share of blocks
  
  if $b$ requires zero modifications
    
    factorize $b_{j,j}$ and send it to the task queues of processors that own blocks in row $J$ or column $J$
  
while factorization is not done

  Receive a block $b_{i,k}$ from my task queue
  
  if $b_{i,k}$ is diagonal ($b_{k,k}$)
    
    for each block $b_{i,k}$ in BMODQ(k)
      
      $b_{i,k} = b_{i,k} *$ inverse ($b_{k,k}$)
    
    send $b_{i,k}$ to the task queues of all processors that own blocks in row $i$ or column $k$
  
else

    add $b_{i,k}$ to BMODQ(K)

    for each pair of blocks in BMODQ(K) $b_{i,k}$ and $b_{j,k}$ where I am the owner of $b_{i,j}$
      
      $b_{i,j} = b_{i,j} - b_{i,k} *$ transpose ($b_{j,k}$)
    
update number of remaining BMOD operations of $b_{i,j}$

if no more BMOD operations on block $b_{i,j}$ is left

  if $b_{i,j}$ is diagonal
    
    factorize it and send it to the task queues of processors in column $j$
  
else

    if $b_{j,j}$ is factorized
      
      multiply $b_{i,j}$ by inverse ($b_{j,j}$) and send it to the task queues of all processors in row $i$
    
    or column $i$
else
add $b_{ij}$ to $BDIVQ(j)$

Figure 6: Parallel Sparse Block Cholesky Factorization

To overlap communication with computation, two PPU threads are launched for each SPE making it possible for an SPE to receive a new task request from one thread while it is still executing another task from the other thread. An SPE will then overlap the transfer of data of the new task with the execution of the current task. PPU-SPE communication is implemented using mailboxes where PPU threads send mailbox messages to the SPU indicating what kind of task is requested and wait for a notification from the SPE that the task is done. When the SPE receives the message it brings the required blocks from the main memory, performs the task, writes the result back to main memory and sends a notification to the PPU thread that the task is done. Figure 7 shows a pseudo code executed by the SPU.

Read message (m1) from the incoming mailbox
Start reading m1 blocks
While (more tasks to execute)
    Finish reading m1 blocks
    Read message (m2) from the incoming mailbox
    Start reading m2 blocks
    Process m1, write results back to memory and notify the PPU thread
    Finish reading m2 blocks
    Read message (m1) from the incoming mailbox
    Start reading m1 blocks
    Process m2, write results back to memory and notify the PPU thread

Figure 7: SPU task overlapping pseudo code

I.5 Parallel GLPK Stability

To maintain the same level of stability as in GLPK, we implemented all kernels using double precision arithmetic. Unfortunately, the Cell BE processor double precision operations are more than one order of magnitude slower than single precision. We are planning to study the possibility of overcoming this problem using the mixed precision technique, which is mainly based on performing the factorization in single precision and use iterative refinement to increase the precision of the result. We also notice that using LAPACK to compute the Cholesky factorization of a block decreases the stability of GLPK especially for degenerate data sets. To overcome this problem, we implemented the block factorization using the algorithm suggested by Meszaros [17].
I.6 Related Work

Multi-core processors and their programmability have recently gotten a lot of interest as increasing the efficiency of single-core processors has become very difficult. The Cell BE processor is one of these systems whose programmability and potential for scientific computing have been investigated by several authors [4, 5, 6]. Up to our knowledge, this is the first LP solver that is implemented on the Cell BE, however, LP solvers using both Simplex and interior point methods have recently been implemented on GPU [7, 8].

Vishwas and others have implemented a single precision sparse Cholesky factorization running on a two-node 3.2 GHz Cell BladeCenter (exercising a total of sixteen SPEs) [5]. Their implementation for the largest data set (28924 x 28924 with 1036208 non-zero entries) that they use has delivered an average of 81.5 GFLOPS which is about 20% of the single precision peak. Our implementation of Cholesky factorization delivered a max of 2.9 GFLOPS when running on a PS3 (6 SPEs) which is about 25% of double precision peak. Both implementations are based on the block algorithm proposed in [13], however, ours uses double precision to achieve a practical numerical stability and the task queues are only accessed by associated PPE threads and not by the SPEs whose main work is to transfer blocks and perform operations on them based on the PPE instructions. Kurzak, Buttari, and Dongarra [18] have also developed Cholesky factorization on the Cell BE processor, but their implementation is intended for dense symmetric positive definite matrices and not for sparse ones. Moreover, they used well-conditioned input matrices while we use Netlib data sets where many of them results in degenerate matrices to be factorized.

I.7 Results and Future Work

Figures 8 and 9 show the speedup obtained by executing the new parallel implementation on the Sony’s PlayStation 3 (PPE + 6 SPEs) relative to executing the original serial GLPK on the PPE using different Netlib datasets [16]. We used the PPU GNU C compiler and the SPU GNU C compilers that come with IBM SDK 3.1. The optimization level was set to 3 and both SIMDMATH and BLAS libraries where used in the SPE implementation. The speedup varies with the dataset being solved, and reaches a maximum of 7.28 for the QAP12 data set. Moreover, the sparse Cholesky factorization delivers 2.9 GFLOPS which is only 25% of the double precision peak.

Several factors affect the obtained speedup. Since only Cholesky factorization is parallelized, the time spent by serial GLPK executing Cholesky factorization relative to the total time it spends solving the data set determines the maximum possible speedup according to (Amdahl's law). This ratio is shown in column 4 of figure 9 for all used Netlib data sets. Another factor affecting the speedup is the size of the data set. The larger the data sets the higher the possibility to have wide supernodes and therefore better utilizing the local store of the SPE. A third related factor is that having many thin supernodes caused the formation of many small blocks which results in inefficient use of local store and large communication overhead.
Figure 8: Speedup of executing parallelized GLPK on PlayStation 3 (PPE + 6 SPEs) relative to executing the original serial GLPK on the PPE using different Netlib data sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size ( rows x columns)</th>
<th># of non-zeros</th>
<th>Cholesky time/total time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>QAP12</td>
<td>3193 x 8856</td>
<td>44244</td>
<td>0.99</td>
<td>7.28</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>3137 x 9408</td>
<td>144848</td>
<td>0.93</td>
<td>7.02</td>
</tr>
<tr>
<td>DFL001</td>
<td>6072 x 12230</td>
<td>41873</td>
<td>0.98</td>
<td>4.73</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>2172 x 5167</td>
<td>35674</td>
<td>0.86</td>
<td>2.23</td>
</tr>
<tr>
<td>PILOT</td>
<td>1442 x 3652</td>
<td>43220</td>
<td>0.79</td>
<td>2.18</td>
</tr>
<tr>
<td>BNL2</td>
<td>2325 x 3489</td>
<td>16124</td>
<td>0.86</td>
<td>1.62</td>
</tr>
<tr>
<td>CYCLE</td>
<td>1904 x 2857</td>
<td>21322</td>
<td>0.80</td>
<td>1.61</td>
</tr>
<tr>
<td>TRUSS</td>
<td>1001 x 8806</td>
<td>36642</td>
<td>0.62</td>
<td>1.50</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1504 x 1818</td>
<td>26230</td>
<td>0.61</td>
<td>1.35</td>
</tr>
<tr>
<td>25FV47</td>
<td>822 x 1571</td>
<td>11127</td>
<td>0.66</td>
<td>1.29</td>
</tr>
<tr>
<td>PILOT-JA</td>
<td>941 x 1988</td>
<td>14706</td>
<td>0.71</td>
<td>1.28</td>
</tr>
<tr>
<td>D6CUBE</td>
<td>416 x 6184</td>
<td>43888</td>
<td>0.29</td>
<td>1.24</td>
</tr>
<tr>
<td>WOODW</td>
<td>1099 x 8405</td>
<td>37478</td>
<td>0.23</td>
<td>1.10</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>245 x 2594</td>
<td>70216</td>
<td>0.04</td>
<td>0.97</td>
</tr>
<tr>
<td>GREENBEB</td>
<td>2393 x 5405</td>
<td>31499</td>
<td>0.57</td>
<td>0.65</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>1152 x 5427</td>
<td>21597</td>
<td>0.13</td>
<td>0.22</td>
</tr>
<tr>
<td>CZPROB</td>
<td>930 x 3523</td>
<td>14173</td>
<td>0.05</td>
<td>0.22</td>
</tr>
<tr>
<td>STOCFOR3</td>
<td>16676 x 15695</td>
<td>74004</td>
<td>0.38</td>
<td>0.11</td>
</tr>
<tr>
<td>80BAU3B</td>
<td>2263 x 9799</td>
<td>29063</td>
<td>0.48</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Figure 9: Information about used Netlib data sets. The 4th column is the ratio of Cholesky factorization time to the total execution time when using serial GLPK on the PPE.
We believe that the performance of the current implementation can be improved in different ways. Despite the use of SIMD, loop unrolling, and a few BLAS operations, a better optimization of the SPE code can result in obtaining higher speedup, while it is still a challenge to implement the MOD operation (line 7 in Fig. 4) in a way that better utilizes the SPE processing power due to different sparsity structure of the blocks. Our parallel SSPD is computationally bounded, which actually limits the benefit of using double buffering and demands more SPE code optimization. Moreover, double precision arithmetic, which is required to have a numerically stable LP solver, is much slower than single precision arithmetic; therefore, more speed up and almost the same stability can be obtained by implementing the mixed-precision technique which is based on using single precision arithmetic and iterative refinement [18]. We are also planning to analyze load balancing among SPEs and study the possibility of using dynamic scheduling of tasks to be executed by the SPEs. On the other hand, our parallel solver can be enhanced to efficiently solve a wider range of datasets by parallelizing other computational kernels of the Mehrotra’s predictor-corrector method and by processing very thin independent supernodes (like SN1 in figure 5) serially so that we avoid manipulating many very small blocks.
References


Paper II

Mixed-Precision Parallel Linear Programming Solver

M. Eleyat and L. Natvig
Computer Architecture and High Performance Computing (SBAC-PAD), 2010
Abstract

We use mixed-precision technique, which is used to exploit the high single precision performance of modern processors, to build the first sparse mixed-precision linear programming solver on the Cell BE processor. The technique is used to enhance the performance of an LP IPM-based solver by implementing mixed-precision sparse Cholesky factorization, the most time consuming part of LP solvers. Moreover, we implemented sparse matrix multiplication of the form required by the solver as it is also very time consuming for some LP problems. Implemented on the Cell BE processor (Playstation 3) and tested using Netlib data sets, our LP solver achieved a maximum speedup of 2.9 just by using the mixed-precision technique. Moreover, we found that some problems, especially in final iterations, result in ill-conditioned matrices where mixed-precision cannot be used. As a result, the solver needs to switch to double precision if a more accurate solution of an LP problem is required.
II.1 Introduction

Linear programming (LP), part of an area of mathematics called optimization techniques, is crucial for industrial, scientific, and engineering applications. Its importance stems from the fact that it works as a decision maker that chooses the suitable values of many variables so that a goal (maximum profit, best resource allocation, … etc.) is achieved while satisfying a set of constraints that are specified as mathematical equalities/inequalities [5].

Solution of most practical real world problems involves the manipulation of very large sparse matrices which makes it difficult to harness the processing power of modern processors compared to that achieved when performing dense computations. One of the reasons for this difficulty is the low computation to communication ratio of sparse computation. Another reason is the overhead of indirect indexing required to load non-zero values whose distribution doesn’t follow any known pattern. Finally, sparse computation is irregular, and therefore, it can’t benefit a lot from the SIMD capabilities of modern processors.

Fortunately, some blocking methods have been suggested to overcome the sparsity overhead for sparse Cholesky factorization and allow exploiting different capabilities of modern processors and utilize BLAS3 operations[3,4]. Blocking used in Cholesky is based on the concept of “supernode”, which is a sequence of columns where each two consecutive columns share the same sparsity structure below the diagonal element. In [2], we implemented a parallel Cholesky factorization that is based on the blocking method suggested by Rothberg and Gupta in [3] and used it to develop a parallel LP solver on the Cell processor.

In this paper, we try to enhance the performance of our LP solver in [2] by using mixed precision technique to exploit the high single-precision computing power while maintaining double precision accuracy. This is quite important for most modern processors [1] as their single precision peak is higher than their double precision peak. In fact, the peak performance single/double is about 14 for the Cell BE. Another enhancement is the implementation of the parallel sparse matrix-matrix multiplication of the form ADA\(^T\) where A is a sparse matrix and D is a diagonal matrix so that more acceleration can be achieved especially for problems where a considerable time is spent performing the multiplication. To our knowledge, mixed-precision sparse Cholesky factorization has never been implemented on the Cell processor and used to enhance the performance of an LP solver.

The paper is organized like the following: algorithms of parallel Cholesky and parallel matrix multiplication are described in section 2. Then, mixed precision technique is explained in section 3 and related work in section 4. Experimental results are discussed in section 5, and finally, conclusion and future work are presented in section 6.

II.2 Parallel sparse LP solver

Our LP solver is built based on the IPM implementation used in GLPK [8, 9]. We parallelized both Cholesky factorization and matrix multiplication since the solver spent
most of the time doing these computations [10]. The following sections explain the parallelization of both kernels.

### A. Parallel Cholesky Factorization

We have implemented a sparse LP solver on the Cell BE [2] that is based on a parallel Cholesky factorization algorithm developed by Rothberg and Gupta [3].

\[
\begin{align*}
1. & \text{ for } k = 1 \text{ to } n \text{ do} \\
2. & \quad L_{kk} := \text{Factor}(L_{kk}) \\
3. & \quad \text{for } i = k + 1 \text{ to } N \text{ with } L_{ik} \neq 0 \text{ do} \\
4. & \quad \quad L_{ik} := L_{ik}(L_{kk})^{-1} \\
5. & \quad \text{for } j = k + 1 \text{ to } N \text{ with } L_{jk} \neq 0 \text{ do} \\
6. & \quad \quad \text{for } i = j \text{ to } n \text{ with } L_{ik} \neq 0 \text{ do} \\
7. & \quad \quad \quad L_{ij} := L_{ij} - L_{ik}(L_{jk})^T
\end{align*}
\]

**Figure 1: Serial block factorization**

The algorithm describes a blocking algorithm that divides the matrix into a set of two dimensional blocks which have different sizes. Being dense, the operations on the result blocks allow utilizing cache, SIMD processors, and the BLAS library [3]. The blocking algorithm is based on the concept of supernodes [4] where each supernode is a set of consecutive columns that share sparsity structure below the diagonal element. The matrix is partitioned vertically into a set of supernodes, then it is partitioned horizontally such that horizontal partitions have the same size as vertical partitions.

A serial block factorization algorithm is shown in Fig. 1. It is based on applying three different mathematical operations to the blocks namely BFACT, BDIV, and BMOD shown in lines 2, 4, and 7 respectively.

The fact that many blocks have only zero-values makes the algorithm more parallelizable. To make use of this fact, the parallel algorithm computes the number of BMOD operations required for each block and keeps it updated during the factorization. When a block receives all block modifications and gets multiplied by the inverse of the diagonal block, it is sent to all processors that own blocks that could be modified by it. The receiver processor performs all related modifications to the blocks it owns. When all block modifications to a certain block are performed, the block is either factorized if it is a diagonal block or multiplied by the inverse of the diagonal block and then sent to other processors. More details about parallel block Cholesky factorization on the Cell BE processor can be found in [2].
B. Parallel Sparse Matrix Multiplication

GLPK requires a matrix multiplication of the form \( S = A D A^T \) where \( D \) is a diagonal matrix stored as one dimensional array and \( S \) and \( A \) are sparse matrices stored in compressed row storage (CRS) [6]. This multiplication is performed using two phases: a symbolic ADA\(^T\) and a numeric ADA\(^T\). The symbolic phase is performed once and is used to determine the non-zero structure of \( S \) allowing efficient computation of the numeric phase. The numeric phase determines the numeric values of \( S \) and is performed every iteration. Fig. 2 shows the storage scheme of sparse matrices and Fig. 3 the pseudo code of the serial implementation of performing ADA\(^T\) in GLPK.

\[
A = \begin{bmatrix}
0 & 3 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 \\
0 & 5 & 9 & 2 & 0 \\
6 & 0 & 0 & 5 & 3 \\
0 & 0 & 5 & 9 & 0
\end{bmatrix}
\]

\( n = 5 \), \( nz(A) = 12 \).

The CRS data structure for \( A \) is:

<table>
<thead>
<tr>
<th>A_val</th>
<th>3 1 4 1 5 9 2 6 5 3 5 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_ind</td>
<td>1 4 0 1 1 2 3 0 3 4 2 3</td>
</tr>
<tr>
<td>A_ptr</td>
<td>0 2 4 7 10 12</td>
</tr>
</tbody>
</table>

Figure 2: Matrix A and its CRS sparse storage

1. for each row \( r \) of \( A \) (i.e. \( A[r] \))
2. use \( A\_ptr \) to determine the start and end indexes of \( A[r] \) in \( A\_ind \) and \( A\_val \)
4. use \( S\_ptr \) to determine the start and end indexes of \( S[r] \) in \( S\_ind \) and \( S\_val \)
5. for each element \( e \) in row \( r \) of \( S \)
6. use \( S\_ind \) to find the column index \( c \) of \( e \)
7. use \( A\_ptr \) to find the start and end indexes of \( A[c] \) in \( A\_ind \) and \( A\_val \)

Figure 3: Serial implementation of serial ADAT using CRS data structure

In general, sparse matrix-matrix multiplication is hard to parallelize efficiently due to indirect indexing, difficult utilization of SIMD, sparsity storage, and load balancing problems [7]. In our implementation, we make use of the symbolic phase to reduce the load balancing problem by assigning each SPE a horizontal partition of \( S \) to compute such that all partitions have almost equal numbers of non-zero values.
The main challenge in the algorithm is that the size of the local storage in SPE is very small to have all required data. We tried to overcome this problem by having only two rows of A in the local store at a time, but this strategy was not successful due to the existence of some dense rows in some problems and because of the need to have all values of D in order to multiply two rows of A. We ended with allocating fixed-size buffers for transferring two rows (or parts of them) of A and gather the associated values of D using DMA lists. Fig. 4 shows the pseudo code of parallel ADA_T executed by each SPE.

1. for each row r of my partition of S “S[r]”
2. S[r,r] = 0
3. for each element e with index c in row r of S
4. S[r,c] = 0;
5. divide A[r] into n parts based on size of A[r]
6. for p= 1 to n
8. load/gather values of Dp associated with Ap[r] Elements
10. for each element e with index c in row r of S
13. write S[r] to main memory

Figure 4: The algorithm executed by each SPE to compute its part of ADAT

The algorithm in Fig. 4 hides some of the details that exist in the actual implementation to make it simple and readable. One of the issues that is not shown is the partition of both S_ptr and A_ptr for problems with large number of rows as they wouldn't fit into the local store. Moreover, the computation of one row of S is also divided into several phases when a row r of S “S[r]” exceeds the allocated buffer. In addition, implementation of lines 11 and 12 include loading parts of row A[c] until all values of Ap[r] are multiplied with their associated values of A[c]. Finally, real implementation uses double buffering mainly to interleave the loading of Ap[c] in line 11 and the computation in line 12.

II.3 Mixed precision

Single floating point operations on modern architectures are usually at least twice as fast as double precision operations[1]. IPM Opteron 246, IBM PowerPC 970, and Intel Xeon 5100 are all examples where single precision peak is twice the double precision one. The single/double gap on some other architectures is much greater, for example, double precision peak is 14 times slower than single precision peak on the Cell BE processor.

For some applications, like most practical linear programming (LP) problems, single precision arithmetic lead to inaccurate results and numerical stability problems. As a result, such applications can not benefit from the high single-precision processing power. Fortunately, a technique called mixed-precision allows performing expensive computation
parts in single precision and refine the results using less expensive double precision operations obtaining the required 64-bit accuracy.

In this paper, we implement a LP solver using the mixed precision technique. More specifically, we implement a mixed-precision algorithm for solving a sparse positive definite symmetric system of the form $Ax = b$ using Cholesky factorization. The algorithm is described in Fig. 5. Step 1 represents Cholesky factorization of $A$ into $LL^T$ after applying permutation $P$ for reducing the fill-in produced in the factor $L$. Steps 2 and 3 (and steps 5 and 6) represents forward and backward solvers. Vector $x$ is the solution to the system and it gets refined through steps 4..7. Note that only steps 4 and 7 are performed in double precision while other steps are performed in single precision. It is important to mention that the overall performance depends on number of refinement iterations and the ability of single precision operations to utilize the available single processing power.

\[
\begin{align*}
1: & \quad LL^T \leftarrow PA \quad (S) \\
2: & \quad \text{solve } Ly = Pb \quad (S) \\
3: & \quad \text{solve } L^T x_0 = y \quad (S) \\
\text{do } k = 1, 2, \ldots & \\
4: & \quad r_k \leftarrow b - Ax_{k-1} \quad (D) \\
5: & \quad \text{solve } Ly = Pr_k \quad (S) \\
6: & \quad \text{solve } L^T z_k = y \quad (S) \\
7: & \quad x_k \leftarrow x_k + z_k \quad (D) \\
\text{check convergence} & \\
\text{done}
\end{align*}
\]

**Figure 5: Mixed-precision**

**II.4 Related Work**

Up to our knowledge, mixed-precision Cholesky for solving sparse matrices has never been implemented on the Cell Processor. In the following, we present work that is highly related to our work.

The authors in [1] presented the performance gap between single and double precision on modern processors and discussed the concept of mixed-precision technique to exploit the relative high performance of single precision. In addition they explained the usage of mixed-precision technique to solve a system of linear equations whether it is dense or sparse, symmetric positive definite or non-symmetric, using both direct and iterative methods. They also showed that the maximum value of the condition number should be less than the inverse of the of the used lower precision for iterative refinement to work well. They tested the technique on several platforms, but mentioned that they only tested dense systems on the Cell processor because of the lack of Cell BE software libraries to perform sparse computations.
Mixed-precision technique has been used for implementing dense Cholesky factorization on the Cell processor [13]. In terms of performance, the authors reported excellent results as their implementation achieved a maximum of 156 Gflops on 8 SPEs, i.e. 76% of the Cell BE single precision peak. On the other hand, they used well conditioned problems that require only two steps of refinement to achieve double precision or more accuracy.

Vishwas and others have implemented sparse Cholesky factorization on the Cell processor, but they used single precision only [14]. Their implementation achieved a maximum of 81.5 Gflops when running on a two-node 3.2 GHz Cell Blade i.e. 20% of the SPE single precision peak.

Our implementation is different from the above mentioned work mainly because it has to deal with sparse matrices that can suffer from much numerical instability. These requirements are imposed by the fact that our implementation is part of a linear programming solver that aims at solving practical real-world problems and that we used Netlib problems which are known to cause solving difficult degenerate matrices.

### Table I: Data sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size (rows x cols.)</th>
<th># of non-zeros</th>
<th>Chol. time</th>
<th>ADA(T) time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFL001</td>
<td>6072x12230</td>
<td>41873</td>
<td>99%</td>
<td>0.2%</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>3137x9408</td>
<td>144848</td>
<td>93%</td>
<td>5%</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>2172x5167</td>
<td>35674</td>
<td>87%</td>
<td>6%</td>
</tr>
<tr>
<td>PILOT</td>
<td>1442x3652</td>
<td>43220</td>
<td>79%</td>
<td>10%</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1504x1818</td>
<td>26230</td>
<td>69%</td>
<td>25%</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>245x2594</td>
<td>70216</td>
<td>5%</td>
<td>89%</td>
</tr>
<tr>
<td>CZPROB</td>
<td>930x3523</td>
<td>14173</td>
<td>4%</td>
<td>84%</td>
</tr>
<tr>
<td>WOODW</td>
<td>1099x8405</td>
<td>37478</td>
<td>23%</td>
<td>63%</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>1152x5427</td>
<td>21597</td>
<td>17%</td>
<td>52%</td>
</tr>
</tbody>
</table>

### II.5 Experimental Results

Our application is based on GLPK C routines that solve LP problems using a serial implementation of Mehrotra’s predictor-corrector method [9], a famous primal-dual interior point method that is used in many practical IPM-based LP solvers. On the other hand, the application is implemented on the Sony’s PlayStation 3 (PPE + 6 SPEs). In addition, all PPU code is compiled using PPU GNU C compiler and all SPU code is compiled using SPU GNU C compiler which both are parts of the IBM SDK 3.1. Moreover, the optimization level was set to 3 and both SIMDMATH and BLAS libraries were used in the SPE implementation.

Experiments have been conducted using the Netlib data sets [11] shown in Table I. The first 5 data sets spend most of the time doing Cholesky factorization and are chosen such that it relatively takes long time to execute using serial GLPK. Other data sets spend a considerable amount of time doing matrix multiplication of the form ADA\(T\), therefore, they represent good candidates to study the efficiency of parallel ADA\(T\). The last two columns
show the time spent by serial GLPK performing Cholesky factorization and sparse matrix multiplication relative to the total execution time.

In the following subsections we demonstrate the performance of parallel sparse matrix multiplication, mixed-precision Cholesky performance, and the overall performance of the LP solver.

A. Parallel Sparse Matrix Multiplication

The performance of parallel sparse multiplication ($\text{ADA}^T$), which is implemented in double precision, is shown in table II. The speedup is calculated by dividing $\text{ADA}^T$ execution time on PPE to that on the Cell processor (PPE + 6 SPEs). It can be noticed that the speedup is better for datasets which take long relative time performing serial $\text{ADA}^T$. Note the second column is explained earlier (last column in table I).

<table>
<thead>
<tr>
<th>Data Set</th>
<th>ADA$^T$ time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CZPROB</td>
<td>84%</td>
<td>7.0</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>89%</td>
<td>4.7</td>
</tr>
<tr>
<td>WOODW</td>
<td>63%</td>
<td>2.9</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>25%</td>
<td>2.4</td>
</tr>
<tr>
<td>PILOT</td>
<td>10%</td>
<td>2.0</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>52%</td>
<td>2.0</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>5%</td>
<td>1.7</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>6%</td>
<td>1.6</td>
</tr>
</tbody>
</table>

B. Mixed-Precision Cholesky Factorization

In this section, we present the overall LP solver speedup caused by using mixed-precision Cholesky factorization compared to solving it using double precision. It is important to notice that mixed-precision solver may produce less accurate results as it can not be used to factorize ill-conditioned matrices produced in final iterations on most big Netlib data sets, therefore, the speedup is calculated taking into account that results have same accuracy by stopping the double precision solver when the mixed-precision solution is achieved (usually same number of iterations produce same accuracy). The LP solver is accelerated by a maximum of 2.9 by using mixed-precision technique. Relative to the double – single peak performance gap of the Cell processor, this humble speedup is due to the following reasons:

- An efficient backward/forward solver, which is executed several times during iterative refinement, is not implemented for the Cell processor. Therefore, the overall gain of using mixed-precision technique is reduced dramatically. Moreover, each iteration of Mehrotra’s predictor-corrector method solves two linear systems using the same Cholesky factorization (i.e. the steps 4..7 of Fig. 5 are repeated twice) which increases
the negative effect of slow backward and forward solvers. Column 4 of table III shows the percentage of time spent doing iterative refinement (steps 4..7 of Fig. 5) relative to total time spent by all steps in the same figure. The overhead is shown clearly when solving MAROS-R7 data set.

- The blocking algorithm causes the creation of many very thin supernodes (small blocks). This cause too much overhead sending its metadata to the SPEs for each related operation and exchanging lots of messages between PPE threads. We reduce the overhead by amalgamating supernodes [16] so that wider supernodes are created by storing some zeros as if they were non-zeros. The average supernode width after using the amalgamation technique (column 3 in table III) is calculated by dividing number of columns of the matrix to be factorized by the number of supernodes after using the amalgamation technique. Since our SPE implementation can handle blocks of sizes up to 76x76, it is clear that current blocking is far from achieving good utilization of SPE resources. The effect of small data blocks on the speedup can be noticed in table III for all data sets. MAROS-R7 is an exception since it suffers more backward and forward overhead that is discussed in previous section.

Table III: Mixed-precision speedup

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Mixed-prec. time (Sec.)</th>
<th>Avg. SN. width</th>
<th>Backward/forward overhead</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFL001</td>
<td>15.5</td>
<td>20</td>
<td>13%</td>
<td>2.90</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>8.71</td>
<td>44</td>
<td>34%</td>
<td>1.15</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>2.67</td>
<td>14</td>
<td>17%</td>
<td>1.64</td>
</tr>
<tr>
<td>PILOT</td>
<td>3.96</td>
<td>10</td>
<td>19%</td>
<td>1.41</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1.84</td>
<td>7</td>
<td>14%</td>
<td>1.17</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>3.29</td>
<td>4</td>
<td>42%</td>
<td>0.90</td>
</tr>
<tr>
<td>CZPROB</td>
<td>1.10</td>
<td>5</td>
<td>15%</td>
<td>1.00</td>
</tr>
<tr>
<td>WOODW</td>
<td>2.19</td>
<td>22</td>
<td>31%</td>
<td>1.19</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>0.76</td>
<td>7</td>
<td>25%</td>
<td>0.91</td>
</tr>
</tbody>
</table>

- Accelerating Cholesky factorization causes the whole LP solver to accelerate at a rate proportional to the percentage of Cholesky factorization time relative to the total execution time (Amdahl's law).

- It is hard to accelerate solving data sets like SHIP12L and CZPROB since they can be solved quickly using serial GLPK. This is due to the large relative overhead of blocking and deblocking, exchanging messages between PPE threads, and sending blocks and their metadata to SPEs.

C. Overall Performance

In this section we compare the performance of the parallel LP solver after parallelizing both Cholesky and matrix multiplication to the performance of serial double precision GLPK when data sets are solved to the same precision. The parallel LP is executed on Playstation 3 (1 PPU and 6 SPEs, 3.2 GHz) and the serial solver is executed on an Intel processor (Intel Core 2 Duo T7300, 2.00GHz) used in Dell830 laptops.
Table IV: Overall Performance

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Serial time of one iteration (Sec.)</th>
<th>Avg. SN. width</th>
<th>RPI</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFL001</td>
<td>6.37</td>
<td>20</td>
<td>1.1x10^{-3}</td>
<td>4.10</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>2.96</td>
<td>44</td>
<td>5.0x10^{-6}</td>
<td>3.73</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>0.15</td>
<td>14</td>
<td>1.1x10^{-5}</td>
<td>1.14</td>
</tr>
<tr>
<td>PILOT</td>
<td>0.26</td>
<td>10</td>
<td>8.0x10^{-4}</td>
<td>1.10</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>0.17</td>
<td>7</td>
<td>1.1x10^{-3}</td>
<td>0.83</td>
</tr>
<tr>
<td>WOOD1P</td>
<td>0.14</td>
<td>4</td>
<td>3.2x10^{-9}</td>
<td>1.12</td>
</tr>
<tr>
<td>CZPROB</td>
<td>0.02</td>
<td>5</td>
<td>3.9x10^{-8}</td>
<td>0.75</td>
</tr>
<tr>
<td>WOODW</td>
<td>0.08</td>
<td>22</td>
<td>1.8x10^{-7}</td>
<td>1.07</td>
</tr>
<tr>
<td>SHIP12L</td>
<td>0.01</td>
<td>7</td>
<td>1.4x10^{-7}</td>
<td>0.30</td>
</tr>
</tbody>
</table>

To give an indication of the achieved accuracy achieved by mixed-precision, the relative primal infeasibility (RPI) [15] of the last iteration is included in table IV. Results show that more speedup is achieved for large data sets that take relatively long time to solve using serial GLPK. Another factor that affects speedup is sizes of the blocks.

II.6 Discussion and future work

As shown in the previous section, mixed-precision technique increases the performance of an LP solver implemented on the Cell BE processor. Moreover, the same technique can be used on other modern processors as most of them execute single precision operations faster than double precision ones.

The existence of efficient sparse backward/forward solvers is important for better utilization of this technique. It is even more important when the matrix to be factorized is not well-conditioned, as more iterative refinement will then be needed. We are planning to study the possibility of building parallel backward/forward solvers to enhance the overall performance of our solver.

We did our experiments using sparse realistic problems which usually cause factorization of ill-conditioned matrices as the algorithm gets closer to more accurate results. Therefore, mixed-precision may not be enough to solve LP problems to the precision required by LP solver-dependent applications. This problem may be solved by switching to a double precision Cholesky once the matrices to be factorized in last iterations start to have high condition numbers.

In addition to the two issues discussed above, current blocking algorithm may result in the creation of very small blocks, which cause much performance degradation. This problem has been reduced by amalgamating consecutive supernodes, but it is not enough to overcome this challenge and it causes sacrificing much of the sparsity benefits. We plan to
use another blocking method where the matrix is partitioned vertically into a group of supernodes and amalgamation can be achieved between nonconsecutive supernodes.
References


Paper III

IPM based sparse LP solver on a heterogeneous processor

M. Eleyat and L. Natvig
Computational Management Science, 2012
Abstract

We present the parallelization of a linear programming solver using a primal-dual interior point method on one of the heterogeneous processors, namely the Cell BE processor. Focus is given to Cholesky factorization as it is the most computationally expensive kernel in interior point methods. To make it easier to develop and port to other heterogeneous systems, we propose a two-phase implementation procedure where we first develop a shared-memory multithreaded application that executes only on the main processor, and then offload the compute-intensive tasks to execute on the synergistic processors (Cell accelerator cores). We used parent-child supernode amalgamation to increase sizes of the blocks, but we noticed that the existence of many small blocks cause significant performance degradation. To reduce the overhead of small blocks, we extend the block fan-out algorithm such that small blocks are aggregated into large blocks without adding extra zeros. We also use another type of amalgamation that can merge any two consecutive supernodes and use it to avoid having very small blocks in a composed block. The suggested block aggregation method is able to speedup the whole LP solver of up to 2.5 when compared to using parent-child supernode amalgamation alone.
III.1 Introduction

Linear programming (LP) has been the focus of much research because many industrial and scientific applications are based on solving LP problems (Luenberger 2007). In addition, solving practical real-world LP problems is a compute-intensive process that can be accelerated using the processing power of emerging multi-core processors.

Our study uses an Interior Point Method (IPM), called Mehrotra’s predictor-corrector method (Mehrotra 1992), to solve LP problems and it is implemented serially as part of the GNU Linear Programming Kit (GLPK) (Makhorin 2008). Most of the time is spent factorizing sparse symmetric positive definite matrices; therefore, we focus on an efficient parallel implementation of sparse Cholesky factorization. The parallel implementation is based on a method suggested by (Rothberg and Gupta 1994), it is called block fan-out method and it is based on supernode oriented blocking of the Cholesky factor.

We have published two papers on LP implementation on the Cell BE Processor: the first describes a general implementation of a parallel IPM-based LP solver (Eleyat and Natvig 2010), while the later exploits the technique of mixed precision to enhance the LP solver efficiency (Eleyat and Natvig 2010). This paper describes the programming model that we used to make it easier to port the parallel solver to other heterogeneous processors. In addition, it extends the blocking method of (Rothberg and Gupta 1994) such that bigger blocks are formed, reducing the overhead of scheduling small blocks and allowing better utilization of the multi-core cache system (local stores in the case of the Cell processor). We refer to the new formed blocks as composed blocks since they are composed of one or more blocks (sub-blocks) that are formed using original blocking.

We are not aware of any LP solvers implemented on the Cell Processor. Vishwas and others (2009) have implemented Cholesky factorization on the Cell processor. However, their design didn’t take portability into consideration. Kurzak, Buttari and Dongarra (2008) developed Cholesky factorization on the Cell BE processor, but their implementation is intended for dense Cholesky factorization where a straight forward blocking results in dense blocks of same size.

Supernode based blocking and supernode amalgamation have been widely used in sparse Cholesky factorization (Lee et al. 2003; Rozin and Toledo 2005; Rothberg and Schreiber 1999) to overcome the overhead of manipulating small blocks and allow exploiting cache, SIMD processing power, and Level 3 BLAS routines. Smelyanskiy and others (2007) used supernode-based blocking without use of amalgamation. Instead, they show, using a cycle accurate simulator, that the hardware support for low overhead task queues proposed by Kumar (2007) can be used to accelerate the scheduling of small tasks. More specifically, the tasks are stored in hardware queues, and are prefetched to the cores so that each core can start a new task as soon as it finishes its current one.

The paper is organized as follows: We explain the architecture of the Cell BE processor in section 2. Then, we discuss the implementation and the programming model in section 3. After that, we introduce a new blocking method in section 4. Finally, we present performance results and conclude in sections 4 and 5 respectively.
III.2 The Cell BE processor - a heterogeneous processors

The Cell Broadband Engine (Cell BE) is a heterogeneous processor that has one main core (Power Processing Element, PPE) and 8 accelerator cores (called Synergistic processing elements (SPEs)) as shown in figure 1. The cores and an on-chip memory controller are linked together by an element interconnection bus (EIB) (Kahle et al. 2005; Shi and others 2009). The main core (PPE) is a 64-bit Power processor with vector processing extensions and two levels of hardware managed caches, a 32 KB L1 data cache and a 512 KB L2 cache. In addition, it is a dual-issue, dual-threaded processor that has a single precision peak of 25.6 GFLOPS and a double precision peak of 6.4 GFLOPS.

The 8 SPEs are SIMD cores which each possesses a 256 KB local store (LS) for storing data and instructions, a 128 x 128-bit register file and a Memory Flow Controller (MFC). MFC has the capability to move code and data between main memory and local stores using a direct memory access (DMA) controller. Moreover, each SPE has a single precision peak of 25.6 GFLOPS and a double precision peak of only 1.83 GFLOPS.

The main core (PPE) is usually responsible for running the operating system and controlling the other cores (SPEs); it can start, stop, interrupt, and schedule processes running on them. In fact, SPEs achieve their work only by followings PPE commands. The PPE can read and write the main memory and the local memories of SPEs through the standard load/store instructions. However, data movement to and from an SPE (local store) is achieved explicitly using DMA commands which poses a major challenge to software development on the Cell BE processor.

A revised variant of the Cell BE processor called the PowerXCell 8i was announced by IBM in 2008 and made available in IBM QS22 Blade servers. The SPEs in the new variant have a much better double precision floating-point peak performance (102.4 GFLOPS) compared to the previous one (14.64 GFLOPS). In addition, PowerXCell 8i has been used in several supercomputers (Green500 list).

Figure 1: The architecture of the Cell BE processor
III.3 Parallel Cholesky factorization on the Cell BE processor

Solving an LP problem using IPM involves repeating a set of matrix operations until finding the optimal solution. In each iteration, most time is spent factorizing a sparse symmetric positive definite matrix. An important fact about this matrix is that it has the same sparsity structure, i.e. location of non-zero values, through all iterations. This fact means that symbolic factorization, the process of identifying the sparsity structure of the matrix and its factors, is performed once. Therefore, supernodes and the structures of the blocks used in parallel implementations are also identified once and used through all IPM iterations.

III.3.1 Two-phase implementation

Heterogeneous processors are multi-core processors where one or more cores are different from other cores. In addition, efficient programming of a heterogeneous processor is a challenging task that requires deep knowledge of its specific architecture and the produced software is hard to port to other multi-core architectures. Taking the Cell processor as an example, optimized SPE code is not portable and the SPE-PPE communication is specific to the Cell architecture.

Our main goal is to utilize the existence of at least one traditional powerful core and build an LP solver that can be ported to other future heterogeneous multi-core systems, without major rewrite of the solver. The target multi-cores systems are those that have at least one traditional core (main core) which supports Pthreads, such as the PPE core in the Cell BE processor.

We achieved our goal using a two-phase development process. The first phase implies building a multi-threaded shared memory LP solver where all threads execute on the main processor (PPE) and all data is stored in main memory. The threads in this phase execute a set of compute-intensive functions which are, as will be seen later, the matrix operations required to perform block Cholesky factorization. The second phase implies executing the compute-intensive functions on the accelerator cores. Instead of calling the real function, each thread on the main core calls an adapter function that communicates with an accelerator to perform the real function.

In general, the first phase produces a relatively easy to debug code and it can be used as a starting point for porting to other heterogeneous multi-core systems. In fact, the first phase code should run on the main core of any heterogeneous processor without any change. The porting process can be performed as shown in the following:

1. start with the phase 1 code and make sure that it can run on the main core without any problems
2. tune the functions code so that it can run on the accelerators and exploit their computation power.
3. build adapter functions that encapsulate communication between the main core threads and the accelerators.
III.3.2 Parallel block Cholesky factorization

The main idea of parallelizing Cholesky factorization is to divide the matrix to be factorized into two-dimensional blocks and determine the block operations that can be performed in parallel by accelerators. Blocking is carried out according to the algorithm introduced by (Rothberg and Gupta 1994) which is based on the concept of a supernode. It is performed by the main thread executing on the main processor (PPE).

Three types of operations (tasks) are performed on the blocks: BFAC, BDIV, and BMOD which are respectively the lines 2, 4, and 7 of the serial block factorization shown in algorithm 1 for the purpose of explaining these operations. The dependency between block operations can be summarized as in the following:

1. BFAC operation on a diagonal block can’t be performed before all BMOD operations on the same block are performed.
2. BDIV operation on a block can’t be performed before all BMOD operations on the same block are performed. In addition it can only be performed after the BFAC operation on the diagonal block of the same column.
3. only after applying a BDIV operation to a block, can it be used to modify other blocks (BMOD).
4. there is no dependency on zero blocks, which means that more parallelism is introduced by the existence of zero blocks.

Algorithm 1: Serial block factorization

1. for k = 1 to N do
2. \( L_{kk} := \text{Factor}(L_{kk}) \)
3. for i = k+1 to N with \( L_{ik} \neq 0 \) do
4. \( L_{ik} := L_{ik}(L_{kk})^{-1} \)
5. for j = k+1 to N with \( L_{jk} \neq 0 \) do
6. for i = j to N with \( L_{ik} \neq 0 \) do
7. \( L_{ij} := L_{ij} - L_{ik}(L_{jk})^T \)

Blocks are statically mapped to the threads as suggested by (Rothberg and Gupta 1994) and operations on a certain block are performed only by the “owner” thread. In other words, each thread is assigned statically a group of tasks (task queue), however, the execution order of tasks in one queue is determined by the order of their dependency satisfaction. To help the thread to decide what tasks are ready to be performed, the number of required BMOD operations is computed for each block in advance. The number is decremented each time a BMOD is performed on a block.

Each thread is required to update other threads right after it finishes an operation, indicating what operation it has just performed and on what block. The thread is also required to get updates from other threads. To allow such kind of communication, each thread maintains another queue, which we refer to as message queue, where received messages are stored before the thread process each of them in order. Figure 2 shows an example of a thread fetching messages from its message queue. As shown in the figure, when thread \( n \) reads that block \( t \) is factorized, it decides to apply the BDIV operation to block \( m \) (block \( m \) is mapped...
to thread \( n \) and it is in the same column as \( t \) and then update other threads. The reader is referred to (Eleyat and Natvig 2010) for more details about parallel sparse Cholesky factorization.

### III.4 Small blocks challenge

The blocking algorithm suggested by (Rothberg and Gupta 1994) is based on supernodes. A supernode is a set of consecutive columns where each two consecutive columns have the same non-zero structure below diagonal element (Ng and Peyton 1993). Resulting supernodes are called fundamental supernodes. However, when applied to Netlib LP data sets (Gay 1985), thin supernodes, and consequently small blocks, are generated. Unfortunately, small blocks cause execution of small tasks (matrix operations) that degrade the performance of the LP solver. The degradation is due to the fact that the time spent exchanging messages between message queues and transferring the blocks between main memory and local stores is relatively long compared to the execution time of block operations performed by accelerator cores.

#### III.4.1 Parent-child supernode amalgamation

To overcome the small blocks problem in the Multifrontal method, (Ashcraft and Grimes 1989) suggested storing some zeros as if they were non-zeros so that some parent-child fundamental supernodes in the supernode elimination tree can be merged into what is called “relaxed supernodes”. However, merging is only allowed if the number of extra added/stored zeros introduced into a relaxed supernode resulted by amalgamation doesn’t exceed a user-defined parameter MAX_NZ.

![Figure 2: Thread n fetches a message from its message queue and takes an action](image)

When testing their method on the Cray X-MP/24 system, the authors chose MAX_NZ by trying several values and choosing the one that cause best performance. The reader is referred to (Ng and Peyton 1993) to read more about supernodes and their dependency represented by supernode elimination tree.
For ease of implementation, we only merge consecutive supernodes. We navigate fundamental supernodes from left to right and allow parent-child consecutive supernodes to be amalgamated as long as the size of the relaxed supernode doesn’t grow beyond a specified size, determined by the size of the SPE local store in the case of the Cell BE processor. Figure 3 shows the non-zero structure of the lower left Cholesky factor of a matrix (3a) partitioned into 4 fundamental supernodes. It also shows the corresponding supernode elimination tree (3c). Storing two zeros (represented by *) in supernode 1 (child) allows merging it with supernode 2 (parent) and both become one supernode in figure 3b which also represented as one shaded area in figure 3c. In the following two sections we introduce two ways of overcoming the problem of small supernodes.

III.4.2 New blocking method

We introduce a new method for producing coarser blocks out of small blocks without storing explicit zeros as in the case of supernodes amalgamation. The new method is an extension to the blocking method suggested by (Rothberg and Gupta 1994). In their method, a block can only be part of one supernode as shown in figure 4a. However, our modified blocking allows the block to be composed of several parts of several adjacent supernodes as shown in figure 4b. The new blocking can be thought of as if small blocks in figure 4a are aggregated to make larger blocks in figure 4b. The new blocks are composed of other blocks (sub-blocks), therefore we choose to refer to the new generated blocks as composed blocks and the small blocks as sub-blocks. Note the sub-blocks are the same blocks generated by the original (Rothberg and Gupta 1994) blocking method. Both supernode amalgamation and the new blocking method can be used to generate coarser blocks, however, the later doesn’t need to store extra zeros and it allows any group of adjacent supernodes to be treated as one supernode that is divided into one column of composed blocks, making it easier to eliminate thin supernodes (small blocks). In terms of storage, the new blocking method requires storing a reference to the beginning of each sub-block including zero sub-blocks. However, the communication overhead is reduced using the scatter/gather SPE capability performed using the DMA list command. To perform
Cholesky factorization using the new blocking method, the same block fan-out shown in algorithm 1 is used, however, the 3 operations BFAC, BDIV, and BMOD need to be replaced by three equivalent versions that work on composed blocks.

To distinguish them from operations on sub-blocks, we denote the 3 required operations on composed blocks as CBFAC, CBDIV, and CBMOD. Fortunately, an operation on a composed block can be performed by performing a sequence of the 3 original operations, BFAC, BDIV and BMOD, on its sub-blocks. In fact, factorization using composed block operations (CBFAC, CBMOD, and CBDIV) involves performing the same sequence of BFAC, BDIV and BMOD operations that would be performed on sub-blocks by algorithm 1.

Suppose that M is the matrix to be factorized and A, B, C and D are composed blocks represented as matrices of sub-blocks and that A is a diagonal block as shown in figure 5 below. The 3 operations on composed blocks can be performed as explained in the following:

**Factorization of a composed diagonal block (CBFAC)**

To factorize a composed block such as block A, we use the same algorithm shown in algorithm 1, however, BFAC, BMOD, and BDIV operations are applied to A sub-blocks. In other words, \(L_{xy}\) in algorithm 1 is just replaced by \(a_{xy}\) that is shown in figure 5.

![Figure: 4 Block aggregation](image-url)
Modifying a composed block (CBMOD)

It is straightforward to perform this operation on a composed block since block matrix multiplication is well-known in mathematics. For example, to compute $\mathbf{D} = \mathbf{D} - \mathbf{C}\mathbf{B}^T$, $\mathbf{B}^T$ is multiplied with $\mathbf{C}$, and the result sub-blocks are subtracted from their correspondent sub-blocks of $\mathbf{D}$.

Multiplying a composed block by the inverse of a diagonal composed block (CBDIV)

The algorithm to compute $\mathbf{B}\mathbf{A}^{-1}$ is shown below:

Algorithm 2: $\mathbf{B} = \mathbf{B}\mathbf{A}^{-1}$ on composed blocks

\[
\begin{align*}
\text{for } k &= 1 \text{ to } n \\
&\times \text{ for } i = 1 \text{ to } m \\
&\times \quad b_{ik} := b_{ik}a_{kk}^{-1} \\
&\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \time
branching in CBFAC and CBMOD. The following is a description of blind amalgamation, a technique that we used to avoid the creation of very small sub-blocks.

Blind amalgamation can merge any group of adjacent supernodes even if they don’t have the parent-child relationship (and that’s why we call it blind). Similar to traditional supernode amalgamation, two adjacent supernodes can be merged by explicitly storing some zero elements. We traverse the fundamental supernodes from left to right and keep merging adjacent supernodes under the constraint that the ratio of number of extra stored zeros to the size of the resulted supernode (number of columns multiplied by number of rows) doesn’t exceed a user-specified parameter, which we refer to as amalgamation parameter. We also don’t allow the width of resulted supernode to exceed a user specified value.

Similar to parent-child amalgamation, the method can be used on its own, i.e. independently from composed blocks, to produce large supernodes, and consequently, large blocks. However, this type of amalgamation doesn’t take dependency among original supernodes into consideration and can, therefore, reduce the number of parallel tasks that can be executed in parallel. For example, suppose that A, B and C denote to three supernodes. Suppose also that A doesn’t depend on C and it is amalgamated with B that depends on C. Consequently, the resulted supernode (A+B) depends on C. Before amalgamation, operations on any two blocks that are made from A and C can be performed in parallel, however, such parallelism is reduced between operation on blocks from A+B and C.

### III.5 Performance results

The application is implemented on the Sony PlayStation 3 (PPE + 6 SPEs) using double precision arithmetic. All PPE code is compiled using PPE GNU 4.3 and all SPE code is compiled using SPE GNU 4.3 C compiler that are parts of the IBM SDK 3.1. Moreover, the optimization level was set to 3 and both SIMDMATH and BLAS libraries were used in the SPE implementation. Experiments have been conducted using Netlib data shown in Table 1. We run our application on data sets that relatively take long time to execute using serial GLPK and that spends most of the time performing Cholesky factorization. The last column show the time spent by serial GLPK performing Cholesky factorization relative to the total execution time.

Our SPE application can perform operations on blocks of up to 64x64 doubles (32 KB); therefore maximum supernode width is set to 64. However, the average width of fundamental supernodes, shown in column 2 of table 2, is very small compared to the maximum allowed width. Blocking based on fundamental supernodes would produce blocks that mostly have less than 4 non-zero values. It will also cause a very low performance because of scheduling, transferring and performing operation on many small blocks, and unbalanced load. Use of parent-child amalgamation of consecutive supernodes that allows adding any number of zeros increases the average width of supernodes as shown in column 3 of table 2, but supernodes are relatively thin for most of the data sets.

We have tested different alternatives for reducing the overhead of small blocks. In all cases, we have first applied parent-child amalgamation and then used the resulted relaxed
supernodes as input to other techniques (blind amalgamation and/or the new blocking method). Moreover, the execution time of Cholesky factorization using parent-child amalgamation was used as a baseline when computing the speedup of using several alternatives. In addition, amalgamation was only allowed when the ratio of added zeros to the size of the amalgamated supernode was less than a user-defined parameter that had been varied from 0.005 to 0.050. The parameter range and increments were selected based on practical tests and reported results in table 2 correspond to the best obtained results. Columns 4, 5 and 6 shows the speedup obtained when using blind amalgamation, block aggregation, and a hybrid of both methods respectively. The following are some comments about the results:

### Table 1. Datasets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size rows x cols.</th>
<th># of non-zeros</th>
<th>Chol. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PILOT87</td>
<td>3608 x 8038</td>
<td>21322</td>
<td>85%</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1504x1818</td>
<td>26230</td>
<td>69%</td>
</tr>
<tr>
<td>PILOT</td>
<td>1442x3652</td>
<td>43220</td>
<td>79%</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>2172x5167</td>
<td>35674</td>
<td>87%</td>
</tr>
<tr>
<td>QAP12</td>
<td>3192x8856</td>
<td>38304</td>
<td>99%</td>
</tr>
<tr>
<td>BNL2</td>
<td>2325 x 3489</td>
<td>16124</td>
<td>87%</td>
</tr>
<tr>
<td>CYCLE</td>
<td>1904 x 2857</td>
<td>21322</td>
<td>80%</td>
</tr>
<tr>
<td>DFL001</td>
<td>6072x12230</td>
<td>41873</td>
<td>99%</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>3137x9408</td>
<td>144848</td>
<td>93%</td>
</tr>
</tbody>
</table>

- Both blind amalgamation and block aggregation (composed blocks) were able to enhance the performance of the whole LP solver except for MAROS-R7 where parent-child amalgamation was enough to produce relatively large blocks. In addition, both methods limit the number of parallel tasks that can be executed in parallel, so there could be a different behavior if many more accelerator cores are used.

- The new blocking method (Block aggregation) has the advantage of not storing extra zeros and that explains why its performance is better than blind amalgamation in most cases. On the other hand, moving data between the SPE local store and memory is more efficient when it is represented as big contiguous chunks. Block aggregation of many small sub-blocks cause the overhead of transferring many small chunks of data and executing a higher number of loop branches introduced in the algorithms of operations on composed blocks explained in section 3.2. This behavior can be seen for PILOT87, which has the smallest average width of relaxed supernode and it explains why blind amalgamation has better performance. Remember that parent-child amalgamation is always included and therefore, sub-blocks are generated based on relaxed supernodes shown column in 3.

- Use of both methods means that sub-blocks are generated based on the supernodes resulted from blind amalgamation of relaxed supernodes. This increases the performance of composed blocks since blind amalgamation reduces number of thin supernodes and it requires a smaller amalgamation factor to produce the best results when compared to using blind amalgamation alone.
III.6 Conclusion

We suggest a two-phase LP solver implementation procedure to be used on heterogeneous processors. The main benefit is that it allows easier porting of the LP solver to other heterogeneous processors. We also propose a method to overcome the overhead of small data blocks by aggregating them and forming large composed blocks.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Avg. width fund. SNs</th>
<th>Avg. width relaxed SNs</th>
<th>Speedup blind</th>
<th>Speedup composed</th>
<th>Speedup blind+composed</th>
</tr>
</thead>
<tbody>
<tr>
<td>PILOT87</td>
<td>1.4</td>
<td>3.0</td>
<td>2.3</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>DEGEN3</td>
<td>1.7</td>
<td>5.5</td>
<td>1.7</td>
<td>1.8</td>
<td>1.9</td>
</tr>
<tr>
<td>PILOT</td>
<td>1.4</td>
<td>5.1</td>
<td>1.5</td>
<td>1.5</td>
<td>1.7</td>
</tr>
<tr>
<td>QAP12</td>
<td>3.0</td>
<td>20.9</td>
<td>1.4</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>BNL2</td>
<td>1.4</td>
<td>7.8</td>
<td>1.1</td>
<td>1.4</td>
<td>1.5</td>
</tr>
<tr>
<td>D2Q06C</td>
<td>1.8</td>
<td>8.8</td>
<td>1.2</td>
<td>1.3</td>
<td>1.4</td>
</tr>
<tr>
<td>CYCLE</td>
<td>2.1</td>
<td>10.5</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
</tr>
<tr>
<td>DFL001</td>
<td>1.5</td>
<td>15.9</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
</tr>
<tr>
<td>MAROS-R7</td>
<td>7.0</td>
<td>41.3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

It is wise to use parent-child supernode amalgamation because it takes supernodes dependency into consideration and so it doesn’t limit the number of concurrent tasks than can be executed in parallel. This method increases the average size of supernodes and therefore the efficiency of factorization. Unfortunately, the sizes of the blocks can still be too small, causing the overhead related to scheduling and executing operations performed on small blocks. This problem can be noticed clearly when solving most of the Netlib problems. To produce bigger blocks, we introduced two methods that combine thin supernodes and produce bigger ones. The new methods can combine any sequence of consecutive supernodes without taking the parent-child dependency into consideration which can reduce the number of tasks that can be executed in parallel. In other words, the new methods can result in a smaller number of big but more interdependent tasks.

Block aggregation, the main method introduced in this paper, encapsulates small blocks into large composed blocks. This method has the advantage of storing no extra zeros and it can gather any group of consecutive supernodes as long as their aggregate width doesn’t exceed the maximum allowed width. However, the method wouldn’t perform well if sub-blocks belonging to a composed block are very small because of the overhead of transferring significant amount of sub-blocks metadata (width, no of rows, etc). In addition, small sub-blocks make it difficult to exploit SIMD processing power when their rows have only 1 or 2 elements.

Blind amalgamation can also combine any group of consecutive supernodes, however, it requires storing extra zeros but no additional metadata. Moreover, each large block can be represented as one chunk of data that can be transferred more efficiently and exploit SIMD processing power. We have tested several possibilities and showed that a hybrid method of
block aggregation and blind amalgamation resulted in a significant enhancement to the overall performance of the LP solver.
References


Paper IV

Cache-Aware Matrix Multiplication on Multicore Systems for IPM-based LP Solvers

Mujahed Eleyat, Lasse Natvig and Jørn Amundsen
In Computer Science and Information Systems (FedCSIS), 2011
Abstract

We profile GLPK, an open source linear programming solver, and show empirically that the form of matrix multiplication used in interior point methods takes a significant portion of the total execution time when solving some of the Netlib and other LP data sets. Then, we discuss the drawbacks of the matrix multiplication algorithm used in GLPK in terms of cache utilization and use blocking to develop two cache-aware implementations. We apply OpenMP to develop parallel implementations with load balancing. The best implementation achieved a median speedup of 21.9 when executed on a 12-core AMD Opteron.
IV.1 Introduction

During recent years, processor designers have moved away from uniprocessor systems to multicore systems. This shift is mainly due to manufacturers inability to continue enhancing the performance of single-core processors [8]. Increasing clock speeds requires higher voltage and causes, consequently, too much heat to dissipate. On the other hand, using deeper pipelines and other advanced architectural techniques have yielded decreasing improvements. In addition, and due to the speed gap between main memory and the processing cores, there has been more demand for an efficient cache system to allow exploiting the collective processing power [21]. As a result, multicore programmers need not only to provide a parallel implementation of the application, but they also have to take cache utilization into consideration for efficient utilization of the multicore system. Techniques to reduce cache and TLB misses depend on the application memory access pattern, for example, tiling/blocking [14] is the most popular method for applications with poor exploitation of temporal locality.

A Linear Programming (LP) solver is one of many compute-intensive applications that could benefit from the high multi-core performance. It works as a decision maker that chooses values of many variables to achieve a goal (maximum profit, best resource allocation, etc.) while satisfying a set of constraints that are specified as mathematical equalities and inequalities [15]. If we have $m$ constraints and $n$ variables, the LP-problem in standard form can be written as:

$$\begin{align*}
\text{minimize } & \quad z = c^T x, \\
\text{subject to } & \quad Ax = b, \quad x \geq 0,
\end{align*}$$

where $x$ is an $n$-dimensional column vector, $c^T$ is an $n$-dimensional row vector, $A$ is a $m \times n$ matrix, and $b$ is an $m$-dimensional column vector.

Solving LP problems in an efficient way is crucial for industrial and scientific fields, especially since an application might need to solve large problems and/or a long sequence of problems. For example, Miriam Regina, a network gas flow simulator developed by Miriam AS [3], solves thousands of LP problems to make a single allocation of gas flow in the network. On the other hand, it needs to solve bigger LP instances for the simulation to cover large networks that span the national boundaries.

The motivation for investigating matrix multiplication in interior point methods (IPM) [15, 16] is that it takes a large fraction of the total computation time when solving some of the data sets. In addition, it is a special form of multiplication of the form $ADA^T$, where $A$ is a sparse matrix, $D$ is a diagonal matrix, and $A^T$ is the transpose of $A$. Moreover, sparse multiplication is a form of irregular computation that is much more challenging to accelerate than dense multiplication. On the other side, the structure of the multiplication result is constant through all IPM iterations, a fact that may be used to enhance its computation performance.

In this paper, we profile serial GLPK [2], an open source LP solver, and present empirical results showing that matrix multiplication takes a relatively long time to compute for some Netlib and miscellaneous problems [4, 1]. We also analyse memory access patterns of sparse multiplication and develop cache-aware algorithms that reduce the rate of cache and TLB misses. Moreover, a parallel version is also provided while trying to exploit the cache hierarchy of the multicore system.
The paper is organized as follows: Section IV.2 gives a brief overview of the AMD Opteron compute node used. Then, compute-intensive parts of the LP solver are introduced in section IV.3. Section IV.4 explains GLPK implementation of sparse matrix multiplication while section IV.5 describes techniques to enhance cache utilization. We present related work in section IV.6 and conclude with experimental results and future work.

IV.2 Multi-core hardware

Introduced in 2009, the 64-bit Istanbul processor is the first 6-core AMD Opteron® processor and is available for 2-, 4-, and 8-socket systems, with clock speeds ranging from 2.0 to 2.8 GHz [11].

Fig. IV.1 shows a simplified block diagram. The processor has six cores, three levels of cache, a crossbar connecting the cores, the System Request Interface, the Memory controller, and three HyperTransport 3.0 links. The memory controller supports DDR2 memory with a bandwidth of up to 12.8 GB/s. In addition, the HyperTransport 3.0 links provide an aggregate bandwidth of 57.6 GB/s and are used to allow communication between different Istanbul processors. Each core has two levels of cache, a 512 KB L2 cache, 64 KB data cache and 64 KB instruction cache. However, all cores share a 6 MB L3 cache.

AMD Opteron multiprocessor systems are based on the cache coherent Non-Uniform Memory Access (ccNUMA) architecture. Each processor is connected directly to its own dedicated memory banks and it uses HT links to communicate with I/O buses and the other processor(s). Fig. IV.2 shows a block diagram of a 2-socket system.

IV.3 GLPK and IPM computational kernels

The GLPK (GNU Linear Programming Kit) package is a set of ANSI C routines contained into a callable library and intended for solving large-scale linear programming, mixed integer programming, and other related problems [2]. GLKP has routines for solving LP problems using either simplex or one of the primal-dual interior point methods (IPMs), namely the Mehrotra’s predictor-corrector method [16]. This method, as well as other primal-dual interior point methods, keeps repeating a set of matrix operations, until it converges to an optimal solution. Every iteration of the algorithm includes the following computations [19]:

1. Sparse matrix-matrix multiplication of the form $S = PAD(PA)^T$, where $P$ is a permutation matrix stored as a single dimensional array, $A$ is the sparse working constraint matrix stored using the CRS format, $D$ is a diagonal matrix stored using a single dimensional array, and $(PA)^T$ is the transpose of matrix $PA$. The output matrix $S$ is a symmetric positive definite matrix.

2. Cholesky factorization of a symmetric sparse matrix $S$, the result of step 1, into $LL^T$ where $L$ is the lower factor matrix and $L^T$ is the transpose of $L$.

IV.3.1 Compressed row storage CRS

Since most practical problems are very sparse, GLPK uses compressed row storage (CRS) to store the constraint matrix and other matrices used in the IPM algorithm. CRS is a general storage format that makes no assumptions about the sparsity structure of the matrix [18]. As shown in Table IV.1, CRS uses three contiguous arrays to store a sparse matrix $A$: $A_{\text{val}}$ stores all nonzero elements row by row, $A_{\text{ind}}$ holds the column indices of the nonzeros, and $A_{\text{ptr}}$ holds the offset of each row into $A_{\text{val}}$. CRS is usually used to access the matrix row by row, while another format called compressed column storage (CCS) is used when a column by column access is needed. Similar to CRS, CSS uses three arrays to store the matrix, however, it stores the nonzeros column by column and $A_{\text{ptr}}$ have pointers to the start of columns in $A_{\text{val}}$.

IV.3.2 Time analysis of IPM computational kernels

Time analysis of serial IPM-based GLPK has been performed when solving big data sets taken from Netlib and the BPMPD website [1]. Size, number of nonzeros, and sparsity, fraction of nonzero elements of the matrix, of each data set are shown in Table IV.2. Results of time analysis are presented in Table IV.3 and they show that Cholesky factorization and sparse matrix multiplication of the form $ADA^T$ are the two most computationally expensive tasks of the

![Simplified block diagram of an AMD Opteron Istanbul processor.](image-url)
solver. However, a few of the data sets, namely BPMPD, CZPROB, and NEMSEMM1, show that a considerable amount of time is spent executing other parts of the code. These results show the importance of accelerating Cholesky factorization and sparse multiplication in order to enhance the performance of IPM based LP solvers.

IV.4 Original implementation of GLPK matrix multiplication and cache problems

As mentioned earlier, the sparse matrix product $S$ in IPM has the form $S = PAD(PA)^T$. This product is computed in two phases: symbolic and numeric. The symbolic phase is performed once and is used to determine the nonzero structure of $S$ for use in the numeric phase.

The numeric phase, which is implemented based on Gustavson’s algorithm [10], is executed every iteration to determine the numeric values of $s_{ij}$ of $S$. Algorithm 1 shows the pseudocode of the GLPK serial implementation of $S = PAD(PA)^T$ where $A$ is an $m \times n$ matrix and $P$

![Block diagram of a 2-socket system](image)

Fig. IV.2. Block diagram of a 2-socket system

Table IV.1. A matrix $A$ and its CRS storage

| $A_{val}$ | 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 | 9 |
| $A_{ind}$ | 2 | 5 | 1 | 2 | 3 | 4 | 1 | 4 | 5 | 3 | 4 |
| $A_{ptr}$ | 1 | 3 | 5 | 8 | 11 | 13 (nonzero count + 1) |

98
Algorithm 1 Serial implementation of $S = PAD(PA)^T$, using $A_\alpha$ for row $\alpha$ of $A$.

1: for $i = 1 \rightarrow m$ do  
2: $i_p = \pi(i)$  
3: $w = (A_{i_p})^T$  
4: for $j = i \rightarrow m$ & $s_{ij} \neq 0$ do  
5: $j_p = \pi(j)$  
6: $s_{ij} = A_{j_p}Dw$  
7: end for  
8: end for

▷ see Algorithm 2

Algorithm 2 Decompression of a row of matrix $A$ into $w$.

1: for $k = A_{\text{ptr}}(i_p) \rightarrow A_{\text{ptr}}(i_p + 1)$ do  
2: $l = A_{\text{ind}}(k)$  
3: $w(l) = A_{\text{val}}(k)$  
4: end for

is stored in a permutation vector $\pi$. The matrix product is computed row by row (line 1) and the permutation is applied at lines 2 and 5. If row $k$ of $S$ has $n_{nz}$ nonzeros, then its computation requires multiplying row $i$ ($i_p$ after permutation) of $A$ by $D$ and by other $n_{nz}$ rows of $A$. Since rows have different sparsity, row $i_p$ is decompressed into a vector $w$ (line 3) as illustrated in Algorithm 2. Each nonzero in row $i$ of $S$ is finally obtained by performing a dot product between $Dw$ and a row of $A$ (line 6).

The GLPK implementation of $S = PAD(PA)^T$ suffers from the following problems with regard to cache utilization:

- Because of the sparsity of $A$, a small fraction of the values in $D$ and $w$ need to be read for the computation of each nonzero of $S$. However, these values are scattered irregularly over large vectors, $D$ and $w$, that don’t have room in L1 and L2 cache. Such irregular access pattern will cause a high cache miss rate.

- Although the rows of $A$ have small number of values that are stored contiguously, the permutations makes it difficult to benefit from data locality and might cause much TLB misses for matrices with high number of nonzeros [12].

<table>
<thead>
<tr>
<th>Problem name</th>
<th>Column count</th>
<th>Row count</th>
<th>Nonzeros</th>
<th>Sparsity $\times 10^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT2D</td>
<td>10525</td>
<td>21024</td>
<td>150042</td>
<td>6.8</td>
</tr>
<tr>
<td>CZPROB</td>
<td>929</td>
<td>3333</td>
<td>10022</td>
<td>32.4</td>
</tr>
<tr>
<td>NEMSEMM1</td>
<td>5668</td>
<td>74151</td>
<td>1036227</td>
<td>24.7</td>
</tr>
<tr>
<td>WORLD</td>
<td>47259</td>
<td>79053</td>
<td>220891</td>
<td>0.6</td>
</tr>
<tr>
<td>NSCT2</td>
<td>23003</td>
<td>37563</td>
<td>697738</td>
<td>8.1</td>
</tr>
<tr>
<td>BPMPD</td>
<td>33841</td>
<td>1144020</td>
<td>3450992</td>
<td>0.9</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>11144</td>
<td>22977</td>
<td>108562</td>
<td>4.2</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>9872</td>
<td>14286</td>
<td>596697</td>
<td>42.3</td>
</tr>
<tr>
<td>DFL001</td>
<td>6084</td>
<td>12243</td>
<td>35658</td>
<td>4.8</td>
</tr>
<tr>
<td>QAP12</td>
<td>3192</td>
<td>8856</td>
<td>38304</td>
<td>13.5</td>
</tr>
<tr>
<td>QAP15</td>
<td>6330</td>
<td>22275</td>
<td>94950</td>
<td>6.7</td>
</tr>
</tbody>
</table>

Table IV.2. Information about the test data sets
Table IV.3. Profiling serial GLPK

<table>
<thead>
<tr>
<th>Problem name</th>
<th>ADAT (%)</th>
<th>Cholesky (%)</th>
<th>Bck/fwd solver (%)</th>
<th>Others (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT2D</td>
<td>98.8</td>
<td>0.3</td>
<td>0.1</td>
<td>0.8</td>
</tr>
<tr>
<td>CZPROB</td>
<td>69.1</td>
<td>7.7</td>
<td>2.2</td>
<td>21.0</td>
</tr>
<tr>
<td>NEMSEMM1</td>
<td>66.3</td>
<td>13.7</td>
<td>1.0</td>
<td>19.0</td>
</tr>
<tr>
<td>WORLD</td>
<td>6.3</td>
<td>81.4</td>
<td>4.8</td>
<td>7.5</td>
</tr>
<tr>
<td>NSCT2</td>
<td>6.1</td>
<td>92.6</td>
<td>0.4</td>
<td>0.9</td>
</tr>
<tr>
<td>BPMPD</td>
<td>34.9</td>
<td>13.8</td>
<td>1.5</td>
<td>49.8</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>33.4</td>
<td>57.5</td>
<td>3.0</td>
<td>6.1</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>11.4</td>
<td>85.8</td>
<td>0.8</td>
<td>2.0</td>
</tr>
<tr>
<td>DFL001</td>
<td>0.2</td>
<td>98.7</td>
<td>0.8</td>
<td>0.3</td>
</tr>
<tr>
<td>QAP12</td>
<td>0.1</td>
<td>99.2</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>QAP15</td>
<td>0.0</td>
<td>98.7</td>
<td>0.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>

IV.5 Cache-aware matrix multiplication

Trying to exploit cache and avoid the problems mentioned in the previous section, we use 1D and 2D partitioning and develop techniques to avoid the overhead of accessing zero blocks and zero block rows. Both extensions of the original algorithm avoid the negative effect of permutation by performing it during the blocking phase, i.e. the rows are permuted in memory before they are split into several blocks. This allows more uniform access to rows of partitions during multiplication. The new algorithms are explained in the following subsections.

IV.5.1 1D partitioning of the matrix A

The method is based on a vertical partitioning of \( PA \) into blocks \( A^{(1)}, A^{(2)}, \ldots, A^{(v)} \), where \( v \) is the number of vertical partitions. Moreover, partitioning is made once since \( A \) is constant and only \( D \) changes through the IPM iterations. In addition, each of the blocks is stored in memory as an independent matrix using CRS. Algorithm 3 shows the pseudocode of the 1D algorithm. \( D \) and \( w \) are accessed in smaller chunks \( D^{(1)}, D^{(2)}, \ldots, D^{(v)} \) and \( w' \) whose size depends on the width of \( A \)-partitions. The goal is to have \( D^{(j)} \)'s and \( w' \) that can fit into L1/L2 cache and be reused through loop iterations at line 5.

One of the drawbacks of 1D partitioning is that many of the partitioned rows have no elements (zero rows) which waste cycles on loading and comparing \( A_{\text{ptr}} \) values. Fig. IV.3A shows an example of a vertical partitioning where 60% of the blocked rows have no elements. In fact, the percentage of zero rows is much higher in real problems as the matrices are much more sparse than the one shown in Fig. IV.3A. Another drawback is the extra storage of ptr array of each partition.

To avoid wasting time on zero partition rows, a higher level of compressed storage is used to efficiently access the nonzero rows of partitions. The matrix, as shown in Fig. IV.3A, is treated as an \( m \times v \) matrix and a new second level of CRS structure (only ptr and ind) is added and used to access nonzero partition rows. This adds more to storage requirements, but has a good
effect on performance. The added Parts\textsubscript{ptr} and Parts\textsubscript{ind} are shown Fig. IV.3B, and the associated multiplication algorithm is shown in Algorithm 4.

### IV.5.2 2D partitioning of the matrix A

Data blocking is a well known technique to utilize data spatial locality [13] and it is well suited for dense matrix multiplication. We try to apply the same technique to sparse matrix multiplication by dividing matrix $A$ into $M \times N$ blocks and consequently matrix $S$ into $M \times M$ blocks all stored using CRS. Computation of an $S$ block is achieved by multiplying two rows of $A$ blocks as shown in Algorithm 5. Blocks have different sparsity and many $A$ and $S$ blocks may have no elements (zero blocks). Different sparsity of different blocks is a reason why blocking is not as efficient as when dealing with dense matrices. However, trying to exploit the existence of zero blocks we add extra information to $S$ blocks as shown in the following:

- Each block of $S$ has an array that stores the indices of nonzero rows.

**Algorithm 3** Vertically partitioned implementation of $S = PAD(PA)^T$, using $A^{(p)}_\alpha$ for row $\alpha$ of partition $A^{(p)}$.

**Require:** $A \leftarrow PA$

1: for $i = 1 \rightarrow m$ do
2: for partition $p = 1 \rightarrow v$ do
3: \[ w' = (A^{(p)})^T \]
4: for $j = i \rightarrow m \& s_{ij} \neq 0$ do
5: \[ s_{ij} + = A^{(p)}_j D^{(p)} w' \]
6: end for
7: end for
8: end for

![Diagram A: Blocking A into 3 vertical partitions](image)

![Diagram B: Using a second level of CRS (ptr and ind) for fast access of nonzero rows of partitions](image)

**Fig. IV.3.** 1D partitioning of matrix $A$
Algorithm 4 Extension of Algorithm 3 with second level CRS

Require: \( A \leftarrow PA \) \hfill \triangleright \text{performed when partitioning}

1: for \( i = 1 \rightarrow m \) do
2: \quad for \( t = \text{Parts}_{\text{ptr}}(i) \rightarrow \text{Parts}_{\text{ptr}}(i + 1) \) do
3: \quad \quad partition \( p = \text{Parts}_{\text{ind}}(t) \)
4: \quad \quad \( w' = (A_i^{(p)})^T \)
5: \quad \quad for \( j = i \rightarrow m \) \& \( s_{ij} \neq 0 \) do
6: \quad \quad \quad \( s_{ij} += A_j^{(p)} D^{(p)} w' \)
7: \quad \quad end for
8: \quad end for
9: end for

Algorithm 5 2D partitioning of matrix \( A \)

1: for \( I = 1 \rightarrow M \) do
2: \quad for \( J = 1 \rightarrow M \) do
3: \quad \quad for \( K = 1 \rightarrow N \) do
4: \quad \quad \quad \( S_{I,J} += A_{I,K} A_{J,K} \)
5: \quad \quad end for
6: \quad end for
7: end for

- Each block of \( S \) has an array of indices of participating pairs of \( A \) blocks.

The goal of the first point is to allow utilization of the already known \( S \) matrix structure. However, the second point aims at avoiding accessing \( A \) blocks that don’t participate into computation of an \( S \) block. Two pairs of \( A \) blocks participate in the computation if their product produces one or more nonzeros, which is simply determined by checking if they have at least one common index of nonzero columns. We determine participating blocks just after the symbolic phase and use it through all IPM iterations. Suppose that matrix \( A \) has \( M \times N \) blocks then \( S \) has \( M \times M \) blocks as shown in Algorithm 5.

IV.5.3 Parallel computation of \( S \) and load balancing

Parallelization of original GLPK implementation and our implementations of the sparse matrix multiplication have been achieved with OpenMP, mainly for loop parallelization. The original GLPK implementation is parallelized by parallelizing the for loop shown at line 1 in Algorithm 1, causing each core to compute a chunk of \( S \) rows. Similarly, 1D partitioning of \( A \) also uses the same principle as the outer loop iterates over rows of \( S \). Finally, multiplication based on 2D partitioning of \( A \) is parallelized by parallelizing the for loop shown at line 1 of Algorithm 5, causing each core to compute a chunk of rows of \( S \) blocks.

Due to different levels of sparsity within the same matrix in general, and as will be seen in the performance results section, our parallel implementation suffers from load imbalance. To address this, we divide \( S \) into a number of shares that equals the number of cores, and assign one core to each share. An ideal share would be any share whose number of nonzeros equals the number of nonzeros in \( S \) divided by the number of cores. Therefore, we try to assign shares such that they differ as less as possible from an ideal share. To force a core to compute one share, we add an outer loop over shares and apply the `omp parallel for` construct to the new added loop as shown in Listing 1.
Listing IV.1. Parallel block based matrix multiplication

```c
#pragma omp parallel for
for row = 1 to M
    for col = 1 to M
        ...
```

A. Parallel 2D algorithm without load balancing

```c
shares = assign_shares (num_cores, num_nonzeros[]);
#pragma omp parallel for
for i = 1 to num_cores
    for row = shares[i].from to shares[i].to
        for col = 1 to M
            ...
```

B. Parallel 2D algorithm with load balancing

A share is composed of a number of consecutive \( S \) rows in the original and 1D algorithms, but it is made of a number of consecutive rows of blocks in the 2D algorithm, making it harder to determine shares that are close to an ideal share. For the 2D algorithm, we try to determine shares that are bigger than an ideal share within a specified tolerance. If \( t \) denotes tolerance and \( d \) denotes number of nonzeros in an ideal share, then a share can have up to \((1 + t)d\) nonzeros. In our implementation, we start trying a 5% tolerance and decide all the shares except the last one. If the size of the last share doesn’t satisfy the tolerance constraint, we keep increasing the tolerance by 5% and repeat the algorithm until all shares respect the tolerance constraint.

IV.6 Related work

Our implementations, although not intended for general sparse matrix multiplication, are based on the classical Gustavson algorithm [10] using compressed row storage of matrices. That algorithm is also used in Csparse [7] and MATLAB [9] and is proven to be optimal with respect to number of operations and storage space of general sparse matrices.

Algorithms for sparse multiplication are developed with focus on optimizing number of operations and storage requirements, however they only perform better than Gustavson’s algorithm when working on certain class of matrices. For example, Park et al. [17] built an efficient algorithm that is based on a compact storage of banded and triangular matrices. On the other hand, Buluç et al. [6] introduced what they called the doubly compressed sparse column (DCSC) which uses less space than compressed column storage (CSC) for storing hypersparse matrices, matrices where number of nonzeros is less than the dimension of the matrix. Such matrices may be the result of a 2D partitioning of sparse matrices for parallel processing.

To our knowledge, Sulatycke et al. are the only researchers who presented sparse matrix algorithms that take efficiency of caches into consideration [20]. Their cache aware algorithms are based on interchanging loops of a standard multiplication algorithm. Moreover, they presented a parallel version that is based on static and dynamic splitting of matrix rows among several threads. However, their experiments were conducted on up to 1000 x 1000 10% sparse matrices, which are much smaller and less sparse than those tested in this paper.
Most recent research about sparse matrix multiplication have been performed by Buluç et al. In [6], they discussed the scalability limitations of matrix multiplication on thousands of processors. Moreover, they developed a sequential hypersparse matrix multiplication algorithm using the DCSC sparse storage to overcome the presented limitations. Parallel implementation was simulated by dividing input matrices using 2D blocking decomposition, excluding other costs like updates and parallelization overheads. Based on their work in [5], load imbalance, hiding communication costs, and additions of submatrices, are the main challenges of parallelizing sparse multiplication. In addition, they have also analysed the scalability of using 1D and 2D block decomposition to divide the work among the processors and show analytically and experimentally that the 2D based algorithms are more scalable than those based on 1D blocking.

IV.7 Performance results and conclusion

IV.7.1 Cache aware matrix multiplication

Our experiments are performed on a 2 x 6 cores AMD Opteron (Istanbul 2431) compute node. All code including GLPK 4.43 is compiled with GCC 4.4.3, with optimization level 3 (-O3). Moreover, execution time of only the first IPM iteration has been measured when solving each of the data sets because iterations caused by solving one data set take the same amount of execution time. Table IV.4 reports the execution time of original GLPK implementation and the new two implementations of the serial matrix multiplication, executed on a single core. Speedup is calculated taking the original implementation as a baseline. The following can be concluded:

1. The new 1D and 2D algorithms execute faster than the original one for all data sets. However, FIT2D is accelerated much more than other data sets. This can be explained by the unique nonzero structure of its $PA$ as shown in Fig. IV.4. The figure is created by placing a dot in the location of each nonzero element, i.e. the horizontal thin bar in the figure represents a group of adjacent dense rows in the matrix. The nonzero structure of this matrix is special because most rows have two values while the last few rows are dense. The original implementation is slow because it accesses most of the $D$ values when one of the dense rows in the bottom of $PA$ is involved, causing much L1 and L2 cache misses. However, the 1D and 2D impletations utilize the cache and improve locality of access as explained in section IV.5.

2. Different data sets are accelerated by different values due to the difference in sparsity. Moreover, the distribution of nonzeros is different among different data sets.

3. 2D avoids accessing nonzero blocks and blocks whose multiplication doesn’t result in any nonzeros, while 1D avoids accessing nonzero rows of partitions. 2D cause more performance when nonzeros are concentrated in chunks causing a lot of nonparticipating blocks to be avoided.
IV.7.2 Size of partitions/blocks

Table IV.5 shows sizes of partitions/blocks that cause optimal speedup for both implementations and for different data sets. The results show that the optimal dimensioning of block/partitions are more related to the distribution of nonzeros than to the sparsity of data sets. If we fix partition size in the 1D implementation to 100 and the block size in the 2D implementation to $100 \times 100$, the speedup of NSCT2 and BAS1LP is reduced by 16% and 14% respectively. However, the speedup of 2D partitioning for CZPROB, NEMSEMM1, NSCT2, BPMPD, OLIVIER, and BAS1LP is reduced by an average of 8%.

IV.7.3 Parallel sparse matrix multiplication

The performance of the parallel matrix multiplication for original implementation and our implementations before and after load balancing is shown in Figs. IV.5-IV.9. The results show the

<table>
<thead>
<tr>
<th>Problem name</th>
<th>Orig. [s]</th>
<th>1D [s]</th>
<th>2D [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT2D</td>
<td>2.455</td>
<td>0.0261</td>
<td>0.0227</td>
<td>94.1</td>
</tr>
<tr>
<td>CZPROB</td>
<td>0.004</td>
<td>0.0006</td>
<td>0.0007</td>
<td>6.6</td>
</tr>
<tr>
<td>NEMSEMM1</td>
<td>0.279</td>
<td>0.0783</td>
<td>0.0684</td>
<td>3.6</td>
</tr>
<tr>
<td>WORLD</td>
<td>0.049</td>
<td>0.0292</td>
<td>0.0414</td>
<td>1.7</td>
</tr>
<tr>
<td>NSCT2</td>
<td>2.015</td>
<td>1.6290</td>
<td>1.2490</td>
<td>1.2</td>
</tr>
<tr>
<td>BPMPD</td>
<td>0.433</td>
<td>0.1047</td>
<td>0.1211</td>
<td>4.1</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>0.088</td>
<td>0.0164</td>
<td>0.0180</td>
<td>5.4</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>0.992</td>
<td>0.7724</td>
<td>0.5933</td>
<td>1.3</td>
</tr>
</tbody>
</table>
high importance of the load balancing. In addition, they show that problems that have longer serial execution time scale better than those which have relatively lower execution time.

Although both implementations show a comparable speedup when executed serially, the later has better speedup when both are executed in parallel. The median speedup of the 1D and 2D implementations is 12.0 and 21.9. Table IV.6 shows the speedup achieved when executing the implementations on 10 cores with the execution time of the original serial algorithm as a baseline. We chose to show the results on 10 cores since some data sets show bad performance when executed on 11 and 12 cores.

To have a more clear view, we show the parallel performance of NEMSEMM1 as an example, in Fig. IV.10. The figure shows that speedup doesn’t increase smoothly with increasing number of cores. This behaviour is due to two main reasons. First, a strange varying OpenMP overhead is observed. It is measured as the difference between the matrix multiplication time and the execution time of the thread that takes most time to finish computing its share. Second, because nonzeros can be concentrated in a small part(s) of the matrix, using nonzeros to divide the shares among threads doesn’t always guarantee that load balancing will be improved. For example, in the 2D algorithm, one thread might be responsible for computing many very sparse blocks, while a second one might be responsible for computing a much lower number of dense blocks. The overhead caused by these two reasons can have a relatively big effect on performance as shown when using 11 and 12 cores.

<table>
<thead>
<tr>
<th>Problem name</th>
<th>Sparsity $\times 10^{-4}$</th>
<th>1D width</th>
<th>2D width x height</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT2D</td>
<td>6.8</td>
<td>100</td>
<td>100 x 100</td>
</tr>
<tr>
<td>CZPROB</td>
<td>32.4</td>
<td>100</td>
<td>100 x 50</td>
</tr>
<tr>
<td>NEMSEMM1</td>
<td>24.7</td>
<td>100</td>
<td>100 x 50</td>
</tr>
<tr>
<td>WORLD</td>
<td>0.6</td>
<td>100</td>
<td>100 x 100</td>
</tr>
<tr>
<td>NSCT2</td>
<td>8.1</td>
<td>1500</td>
<td>100 x 50</td>
</tr>
<tr>
<td>BPMPD</td>
<td>0.9</td>
<td>100</td>
<td>100 x 70</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>4.2</td>
<td>100</td>
<td>600 x 625</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>42.3</td>
<td>400</td>
<td>200 x 50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem name</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIT2D</td>
<td>270.3</td>
</tr>
<tr>
<td>CZPROB</td>
<td>15.6</td>
</tr>
<tr>
<td>NEMSEMM1</td>
<td>19.7</td>
</tr>
<tr>
<td>WORLD</td>
<td>7.4</td>
</tr>
<tr>
<td>NSCT2</td>
<td>8.3</td>
</tr>
<tr>
<td>BPMPD</td>
<td>7.0</td>
</tr>
<tr>
<td>OLIVIER</td>
<td>18.8</td>
</tr>
<tr>
<td>BAS1LP</td>
<td>6.6</td>
</tr>
<tr>
<td>Median Speedup</td>
<td>12.0</td>
</tr>
</tbody>
</table>
An efficient LP solver is crucial for many scientific and industrial applications. However, most research has been focused on an efficient Cholesky factorization since it is the most expensive computation in interior point methods. We showed that, similar to Cholesky factorization, sparse matrix multiplication in IPM-based solvers use a relatively high percentage of the total execution time when solving some big data sets, and proposed two cache-aware implementations of the sparse multiplication algorithm used in GLPK. Moreover, we used OpenMP to parallelize the multiplication and developed a simple, but efficient technique for load balancing.

Due to many zero rows of very sparse blocks, CRS and CCS are not optimal wrt. space for storing very sparse blocks, but we had to use them for two reasons,

- Block sparsity varies a lot even in the same data set
- Our algorithms requires very fast access to rows/columns.

It might be possible to use different storage and computation mechanisms for blocks based on their sparsity. One approach to accomplish this is to use 2D partitioning but using larger blocks, and then choose the appropriate storage mechanism and block multiplication procedure based on the sparsity level. In case of dense (or close to dense blocks), another level of blocking can be performed to better utilize the cache.

Since blocking is our main technique of exploiting cache, it is interesting to try our algorithms on other multicore systems that have different cache systems.
Fig. IV.6. Parallel original GLPK implementation with load balancing.

Fig. IV.7. Parallel 1D algorithm with load balancing.
Fig. IV.8. Parallel 2D without load balancing.

Fig. IV.9. Parallel 2D with load balancing.
Fig. IV.10. NEMSEMM1 with load balancing. Original serial execution time is used as a baseline.
References


Paper V

The maximum flow problem with minimum lot sizes

Dag Haugland, Mujahed Eleyat and Magnus Lie Hetland
In Computational Logistics, 2011
Abstract

In many transportation systems, the shipment quantities are subject to minimum lot sizes in addition to regular capacity constraints. This means that either the quantity must be zero, or it must be between the two bounds. In this work, we consider a directed graph, where a minimum lot size and a flow capacity are defined for each arc, and study the problem of maximizing the flow from a given source to a given terminal. We prove that the problem is NP-hard. Based on a straightforward mixed integer programming formulation, we develop a Lagrangian relaxation technique, and demonstrate how this can provide strong bounds on the maximum flow. For fast computation of near-optimal solutions, we develop a heuristic method that departs from the zero solution and gradually augments the set of flow-carrying (open) arcs. The set of open arcs does not necessarily constitute a feasible solution. We point out how feasibility can be checked quickly by solving regular maximum flow problems in an extended network, and how the solutions to these subproblems can be productive in augmenting the set of open arcs. Finally, we present results from preliminary computational experiments with the construction heuristic.
V.1 Introduction

In transportation systems, as well as in production and manufacturing, some operations might be effective only when the processed quantity lies above a given threshold. Reasons for such restrictions are of diverse nature. Operations might require lots to be large in order to be cost effective, the products appear only in batches of a minimum size, or underlying mechanical and chemical processes require a minimum level of operation.

Operations involving setup costs are somehow related to those involving minimum lot sizes. Their resemblance is reflected by the corresponding optimization models, which in both cases imply the introduction of binary variables representing the decision of whether or not to activate the operation. An obvious distinction is that negative effects of a yes-decision in the case of setup costs are confined to the objective function, whereas in the case of minimum lot sizes, also feasibility is affected. As a consequence, solution approaches that work well in the former case may be non-trivial to translate to the latter.

Production planning models respecting lot size decisions and constraints appear abundantly in the operations research literature. Most of these are extensions of the classic capacitated lot sizing problem (CLSP), of which several surveys [2, 6, 8] are found. CLSP amounts to optimize the lot sizes over a multi-period horizon, when, in addition to holding and production costs, a setup cost is incurred in each period of production. In its simplest form, the CLSP model does not include lower bounds on the lot sizes. As suggested e.g. in the model by Voß and Woodruff [11], minimum lot sizes may replace or come in addition to setup costs, and the model updates are straightforward.

Network flow models for optimizing transportation decisions are also often extended by binary variables in order to reflect setup costs. The fixed charge network flow problem [5, 9] is a direct extension of the minimum cost flow problem, where a fixed cost for utilizing the arc is added to the flow proportional term. Another extension of the minimum cost flow problem is the minimum cost circulation problem [10], in which lower flow bounds are defined on the arcs. Contrary to the fixed charge network flow problem, the circulation problem is modeled as a linear program. The minimum lot sizes are considered as hard constraints in the sense that it is not an option to put the flow to zero in case it is infeasible or suboptimal to respect the bound. Flow models acknowledging this option do not seem to be well studied in the scientific literature. However, for the reasons given in the opening paragraph, we believe that such models have relevance. For instance, the Miriam Regina system [1] for testing production availability and deliverability in the process industry and in energy applications, incorporates a flow model where processes can be operated in a semi-continuous range (either they are turned off or they are assigned a capacity above a non-zero bound).

In this article, we study network flow optimization subject to minimum lot size constraints. For reasons of simplicity, and consistent with the approach taken in the Miriam Regina system, we choose flow maximization as the underlying model. The purpose of the work is to suggest efficient computational methods that, despite the proven intractability of the problem, are able to produce near-optimal solutions with modest computational effort.

The rest of the article is organized as follows. In the next section, we introduce some nomenclature and mathematical notation, and define our network flow model in rigorous terms. Section
V.3 provides a proof of NP-hardness of the problem. In Section V.4, we formulate the problem as a mixed integer programming problem, and we show how Lagrangian relaxation can be applied in order to compute upper bounds on the maximum flow. We also give a fast method for translating the (possibly infeasible) relaxed solutions into feasible ones. The result in Section V.3 encourages a study of possibly inexact solution methods, and in Section V.5, we develop a construction heuristic based on an augmenting path approach. Computational results with this heuristic are given in Section V.6.

V.2 Problem definition

Let $G = (N, A)$ be a directed graph with node set $N$ and arc set $A$, and non-negative integer vectors $\ell$ and $u$ of respectively lower and upper flow bounds. We assume that $G$ contains a unique source $s \in N$ and a unique sink $t \in N$, and that $A$ contains a circulation arc $(t, s)$ with $\ell_{ts} = 0$ and $u_{ts} = \infty$ from the sink to the source. Let $\ell_{ts} = 0$ and $u_{ts} = \infty$.

We consider the problem of maximizing the flow from the source through the network to the sink, such that the flow at each arc is either zero or between the two bounds. This is equivalent to maximizing the flow recycled along arc $(t, s)$. By defining the set of circulations in $G$ as

$$ F(G) = \left\{ x \in \mathbb{R}_+^A : \sum_{j:(i,j) \in A} x_{ij} - \sum_{j:(j,i) \in A} x_{ji} = 0 \forall i \in N \right\}, $$

the problem is expressed as:

$$ \max_{x \in F(G), \ x_{ij} \in \{0\} \cup [\ell_{ij}, u_{ij}]} x_{ts}, \quad (V.1) $$

Henceforth, we say that arc $(i, j)$ is closed if $x_{ij} = 0$, and open otherwise. We let $X \subseteq A$ be the set of open arcs, and let $\bar{X} = A \setminus X$ be the set of closed arcs. Let $G_X$ denote the subgraph with node set $N$ and arc set $X$. We say that $X$ is feasible if there exists some flow vector $x \in F(G)$ satisfying $\ell_{ij} \leq x_{ij} \leq u_{ij}$ for all $(i, j) \in X$, and $x_{ij} = 0$ for all $(i, j) \in \bar{X}$. Let $z(X)$ denote the maximum value $x_{ts}$ can take under these conditions, and let $z(\emptyset) = -\infty$ if $X$ is infeasible. Observe that the empty set is feasible with $z(\emptyset) = 0$.

V.3 Computational complexity

**Proposition 1.** Problem (V.1)–(V.3) is NP-hard.
Proof. The proof is by a polynomial reduction from the \textsc{Subset Sum} problem, which is known to be NP-complete [3, p. 951]. Given a finite set \{a_0, a_1, \ldots, a_n\} of positive integers, the problem is to decide whether there exists a subset \(S \subseteq \{1, \ldots, n\}\) such that \(\sum_{i \in S} a_i = a_0\). Define the digraph with node set \(N = \{s, t, v_0, \ldots, v_n\}\) and arc set \(A = \{(s, v_1), \ldots, (s, v_n), (v_1, v_0), \ldots, (v_n, v_0), (v_0, t)\}\), and let the flow bounds be \(\ell_{v_0t} = u_{v_0t} = a_0\) and \(\ell_{sv_i} = \ell_{v_0v_i} = u_{sv_i} = u_{v_0v_i} = a_i\) \((i = 1, \ldots, n)\). Hence, if there does exist a subset \(S\) as requested, then an optimal solution to (V.1)–(V.3) is to send \(a_0\) units from \(s\) to \(t\) via nodes \(v_i, i \in S\). Otherwise, the only feasible solution is \(x = 0\). \(\Box\)

\section*{V.4 Integer programming model and Lagrangian relaxation}

By defining the decision vector \(y \in \{0, 1\}^A\), where \(y_{ij} = 1\) if \(x_{ij} \in [\ell_{ij}, u_{ij}]\) and \(y_{ij} = 0\) if \(x_{ij} = 0\), we arrive at the mixed integer linear program

\[
\begin{align*}
\max & \ x_{ts}, \\
\text{subject to} & \ x \in F(G), \\
& x_{ij} - \ell_{ij}y_{ij} \geq 0, \ (i, j) \in A, \\
& x_{ij} - u_{ij}y_{ij} \leq 0, \ (i, j) \in A, \\
& y \in \{0, 1\}^A.
\end{align*}
\]

(V.4)–(V.8)

Define \(F(G, u) = \{x \in F(G) : x_{ij} \leq u_{ij} \forall (i, j) \in A\}\), and the maximum flow out of node \(i \in N\) when the lower flow bounds are relaxed as \(M_i = \max \left\{ \sum_{j \in A} x_{ij} : x \in F(G, u) \right\}\). Then any feasible solution to (V.4)–(V.8) satisfies \(\sum_{j : (j, i) \in A} \ell_{ji}y_{ji} \leq M_i\), stating that a set of arcs entering node \(i\) cannot be opened if the sum of their lower flow bounds exceeds the maximum flow out of the node.

Let \(\lambda \in \mathbb{R}_+^A\) and \(\mu \in \mathbb{R}_+^A\), where \(\lambda_{ts} = \mu_{ts} = 0\), and consider the Lagrangian relaxation of (V.4)–(V.8):

\[
L(\lambda, \mu) = \max \sum_{(i, j) \in A} (h_{ij}x_{ij} + d_{ij}y_{ij}),
\]

subject to \(x \in F(G, u)\),

(V.9)–(V.10)

\[
\sum_{j : (j, i) \in A} \ell_{ji}y_{ji} \leq M_i, \quad i \in N,
\]

(V.11)

\[
y \in \{0, 1\}^A.
\]

(V.12)

where \(h_{ij} = \begin{cases} 1, & (i, j) = (t, s) \\ \lambda_{ij} - \mu_{ij}, & (i, j) \neq (t, s) \end{cases}\), and \(d_{ij} = \mu_{ij}u_{ij} - \lambda_{ij}\ell_{ij}\). That is, we have applied the Lagrangian multipliers \(\lambda_{ij}\) and \(\mu_{ij}\) to (V.6) and (V.7), respectively. To strengthen the relaxation, the upper flow bounds are kept in (V.10) and the valid inequalities (V.11) are added.

Consequently, we arrive at a minimum cost flow problem in the \(x\)-variables. In the \(y\)-variables, the problem is decomposed into a set of \(|N|\) knapsack problems, and is hence solvable in pseudopolynomial time.
V.4.1 A heuristic for finding feasible solutions

Since (V.9)–(V.12), the optimal solution of which is denoted \((x^L, y^L)\), is a capacitated minimum cost flow problem in the \(x\)-variables, the arcs \( \{(i, j) \in A : 0 < x^L_{ij} < u_{ij} \} \) define no cycles in \(G\). Let this arc set be contained in some \(A_T \subseteq A\) where \((N, A_T)\) is a tree (arc directions disregarded). That is, we have \(x^L_{ij} \in \{0, u_{ij}\}\) for all \((i, j) \in A_T = A \setminus A_T\).

With the purpose of increasing the number of feasible arc flows, each iteration of the following heuristic sends flow along a cycle consisting of arcs in \(A_T\) and exactly one arc, \(k\), in \(\bar{A}\). As a result, arc \(k\) replaces some cycle arc \((i, j) \in A_T\), which in its turn is assigned flow equal to either 0, \(\ell_{ij}\) or \(u_{ij}\). Throughout the heuristic, we thus have that all non-tree arcs have either zero flow or flow on either of their bounds. Initially, none of them have flow on their lower bound. The tree arcs are assigned flow values between zero and the upper bound. If no arc has flow strictly between zero and the lower bound, a feasible solution is found, and the heuristic terminates.

For all \(k \in \bar{A}\), let \(C_k\) denote the arc set of the unique cycle consisting of arc \(k\) and arcs in \(A_T\).

We accept to let \(k\) replace some tree arc if and only if, for some number \(f\), both the following conditions hold when \(f\) units of flow are sent along \(C_k\):

1. All currently feasible arc flows remain feasible.
2. At least one currently infeasible arc flow becomes feasible.

An arc in \(C_k\) is said to be a forward (backward) arc if it is (not) directed consistently with \(k\). To remove ambiguity, we assume that the \(f\) flow units are sent in the forward direction. That is, all forward arcs in \(C_k\) have their flow values increased (decreased) if \(f > 0\) (\(f < 0\)), and the converse is true for the backward arcs. If no arc meeting the conditions can be found, the heuristic terminates and fails to find a feasible solution.

**Proposition 2.** The set of \(f\)-values satisfying condition 1 consists of a closed interval and at most \(|C_k|\) distinct values.

**Proof.** Consider a traversal of the cycle starting by arc \(k = (i_0, i_1)\), and let \(i_0, \ldots, i_{|C_k|−1}\) denote the nodes hence visited. Let \(i_{|C_k|} = i_0\), and let \(F_m\) denote the set of \(f\)-values that make the flow on all arcs on the path \((i_0, \ldots, i_m)\) feasible \((m = 0, \ldots, |C_k|)\). Assume \(F_m = [lb_m, ub_m] \cup P_m\), \(m < |C_k|\), where \(|P_m| \leq m\), and observe that the assumption holds for \(m = 0\) (let \(lb_0 = -\infty\), \(ub_0 = +\infty\)). If \(k_m = (i_m, i_{m+1})\) is a forward arc then the set \(S_m\) of \(f\)-values rendering its flow value feasible is \(S_m = I_m \cup \{p_m\}\), where

\[
(I_m, p_m) = \begin{cases} 
(\ell_{k_m}, u_{k_m}), x_{k_m} = 0, \\
([0, u_{k_m} - \ell_{k_m}], -\ell_{k_m}), x_{k_m} = \ell_{k_m}, \\
(\ell_{k_m} - u_{k_m}, 0), x_{k_m} = u_{k_m}.
\end{cases}
\]

Otherwise, we have \(S_m = I_m \cup \{p_m\}\), where

\[
(I_m, p_m) = \begin{cases} 
(0, -u_{k_m}), x_{k_m} = 0, \\
(\ell_{k_m} - u_{k_m}, \ell_{k_m}), x_{k_m} = \ell_{k_m}, \\
([0, u_{k_m} - \ell_{k_m}], u_{k_m}), x_{k_m} = u_{k_m}.
\end{cases}
\]
This yields $F_{m+1} = F_m \cap S_m = ([lb_m, ub_m] \cap I_m) \cup P_{m+1}$, where $P_{m+1}$ consists of all values in $P_m$ contained in $S_m$ and also $p_m$ if $p_m \in F_m$. It follows that $F_{m+1}$ is composed of an interval and at most $m + 1$ singular values, and the result follows by induction. \[\square\]

The proposition covers the special case where $f = 0$ is the only value satisfying the condition, in which case the interval in question is empty.

The proof of Proposition 2 is constructive in giving an algorithm for computing the feasible flow assignments to $C_k$. Once we have found that condition 1 is met by $f \in [lb, ub] \cup \{p_1, \ldots, p_{|C_k|}\}$, we check whether any of the values $lb, ub, p_1, \ldots, p_{|C_k|}$ satisfies the second condition. Among all such values, we select the one maximizing $x_{ts}$ if $(t, s) \in C_k$. Otherwise, a value maximizing the number of feasible arc flows is chosen. It is straightforward to see that by this choice of $f$-value, at least one arc in $C_k$ will be assigned zero flow or flow on one of its bounds, and is hence qualified to leave $A_T$.

The above procedure is repeated for all $k \in \bar{A}_T$ in an arbitrary order, and interrupted when a feasible flow is found or when the two conditions cannot be met.

V.4.2 Variable fixing

Combining a feasible flow and the upper bound $L(\lambda, \mu)$ on the maximum flow may help to fix the value of certain $y$-variables. Assume the heuristic above or any other method has identified some feasible flow vector $x^H$, and assume there exists some other feasible solution $(x^1, y^1)$, where $y^1_k = 1$ for some $k \in A$. Let $(x^0, y^0)$ be identical to $(x^1, y^1)$, except that $y^0_k = 0$. Since $(x^0, y^0)$ is feasible in (V.9)–(V.12), we have $L(\lambda, \mu) \geq \sum_{(i,j) \in A} (h_{ij} x^0_{ij} + d_{ij} y^0_{ij}) = \sum_{(i,j) \in A} (h_{ij} x^1_{ij} + d_{ij} y^1_{ij}) - \mu_k u_k + \lambda_k \ell_k \geq x^1_{ts} - \mu_k u_k + \lambda_k \ell_k$. It follows that if $x^1_{ts} > x^H_{ts}$ then $L(\lambda, \mu) + \mu_k u_k - \lambda_k \ell_k > x^H_{ts}$. Hence, if $\mu_k u_k - \lambda_k \ell_k \leq x^H_{ts} - L(\lambda, \mu)$, opening arc $k$ cannot yield solutions that are superior to $x^H$, and consequently, we fix $y_k = 0$. Since $x = 0$ is feasible, we obtain as a special case that all arcs $k$ for which $\mu_k u_k - \lambda_k \ell_k \leq -L(\lambda, \mu)$ can be closed. This holds regardless of how $\lambda \in \mathbb{R}_+^A$ and $\mu \in \mathbb{R}_+^A$ are chosen, and the tighter the upper bound $L(\lambda, \mu)$ is, the more variables can be fixed.

V.4.3 The Lagrangian dual

In order to have the tightest possible upper bound on the maximum flow, and thereby to be able to fix a maximum number of variables, it is desirable to solve the Lagrangian dual problem $\min_{\lambda, \mu \in \mathbb{R}_+^A} L(\lambda, \mu)$, e.g. by the popular subgradient (dual descent) algorithm.

Since the relaxed problem in the $y$-variables does not have integrality property, the minimum bound $\min_{\lambda, \mu \in \mathbb{R}_+^A} L(\lambda, \mu)$ has a potential to dominate the bound provided by the LP-relaxation of (V.4)–(V.8). It is easily seen that the latter is simply the max flow in $G$ when the lower flow bounds are neglected. Therefore, the improved bound comes for the cost of solving $|N|$ knapsack problems in each iteration of the subgradient algorithm, in place of solving a single maximum flow problem. The size of any knapsack problem is however proportional to the in-degree of the corresponding node, and for sparse graphs this represents a modest computational cost.
V.5 Construction heuristic

In this section, we give a heuristic method that starts with $X$ empty, and then gradually extends $X$ as long as extensions can be found. This will produce a sequence of arc sets, some of which will be feasible while others are infeasible. Whenever $X$ is feasible, we attempt to extend it such that the max flow is improved. Otherwise, we look for extensions that reduce the constraint violations. The best feasible $X$ hence encountered is output from the algorithm.

This algorithmic idea is indicated by Algorithm 6.

**Algorithm 6 Idea**($G,\ell,u,(t,s))$

1. $X \leftarrow \{(t,s)\}$ // The initial set of open arcs consists uniquely of the circulation arc
2. $z^* \leftarrow 0$, $X^* \leftarrow X$ // best solution ever

repeat

1. Check whether $X$ is feasible
2. if feasibility check positive then
   1. Let $x$ be a flow allocation in $G_X$ maximizing $x_{ts}$
   2. if $z(X) > z^*$ then
      1. $z^* \leftarrow z(X)$, $X^* \leftarrow X$ // best solution ever
   end if
3. $S \leftarrow$ extension of $X$ suggested to increase $x_{ts}$
4. else
   1. $S \leftarrow$ extension of $X$ suggested to reduce constraint violation
5. end if
6. $X \leftarrow X \cup S$

until $S = \emptyset$ return $X^*$

V.5.1 Checking feasibility

Finding a feasible arc set $X$ is not a trivial problem. Neither is it trivial to find a good extension $S$ of an already feasible $X$, and we may encounter that $X \cup S$ is inferior to $X$ and even infeasible. Checking whether any arc set is feasible, is however accomplished by solving a standard max-flow instance in an auxiliary network defined as follows [8, Section 10.2]:

Let $G_X' = (N', A')$ be a digraph with node set $N' = N \cup \{(s', t')\}$ and arc set $A' = X \cup \{(t', s')\} \cup \{(s', i) : i \in N\} \cup \{(i, t') : i \in N\}$. That is, we add to $G_X$ an auxiliary source $s'$ and an auxiliary sink $t'$ and a circulation arc between them. We also add arcs from $s'$ to each node in $N$, and from each node in $N$, we add an arc to $t'$.

With each arc $(i, j)$ in the extended digraph, we associate a flow capacity (an upper flow bound) $c_{ij}$, but we do not define lower flow bounds in this network (or we can assume they are all 0). We let $c_{ij} = u_{ij} - \ell_{ij}$ for all $(i, j) \in X$ and $c_{t's'} = \infty$. For the new arcs joining the auxiliary source with other nodes, the capacities are defined as $c_{s'i} = \sum_{j : (j,i) \in X} \ell_{ji}$ for all $i \in N$. Likewise, capacities on arcs entering $t'$ are defined as $c_{it'} = \sum_{j : (i,j) \in X} \ell_{ij}$ for all $i \in N$. This means that the capacity of arc $(s', i)$ becomes the sum of lower bounds on arcs in $G_X$ entering node $i$, while the capacity of arc $(i, t')$ is the sum of lower bounds on arcs in $G_X$ leaving node $i$. Observe that

$$\sum_{i \in N} c_{s'i} = \sum_{i \in N} c_{it'} = \sum_{(i,j) \in X} \ell_{ij}.$$
Proposition 3. If \( x' \in \mathcal{Z}^G \) is a flow allocation in \( G'_X \) such that \( x'_{t's'} \) is maximized, then the arc set \( X \) is feasible if and only if
\[
x'_{t's'} = \sum_{(i,j) \in X} \ell_{ij}.
\]

Proof. See Theorem 10.2.1 in [8]. □

It follows that feasibility of \( X \) can be checked by solving a standard (that is, without lower flow bounds) max-flow problem in \( G'_X \). If the maximum flow equals the total capacity of the arcs leaving \( s' \) (and thereby also the total capacity of the arcs entering \( t' \)), then \( X \) is feasible. Otherwise, \( X \) is infeasible.

We solve the standard max-flow problem in \( G'_X \) by an augmenting-path algorithm. In each iteration, flow is sent along a path from \( s' \) to \( t' \) in the residual network of \( G'_X \), and the algorithm terminates when no such path exists.

V.5.2 Allocating flow to a set of feasible arcs

Assume we have computed \( x' \) by the augmenting-path algorithm, and verified that the condition in Proposition 3 holds. We conclude that \( X \) is feasible. Actually, the flow satisfying (V.2)–(V.3) is given as \( x_{ij} = x'_{ij} + \ell_{ij} \) for all \( (i,j) \in X \). This is however not necessarily the flow allocation that yields \( z(X) = x_{ts} \). To maximize \( x_{ts} \), we go on searching for augmenting paths in the residual network of \( G'_X \), but now the paths go from \( s \) to \( t \). All arcs incident to \( s' \) or \( t' \) are ignored, because the flow here must be unchanged. The maximum flow in \( G_X \) is finally found by adding \( \ell_{ij} \) to \( x'_{ij} \) for all \( (i,j) \in X \).

V.5.3 Extending a feasible solution

If \( x \) is the feasible flow allocation to \( G_X \) that maximizes \( x_{ts} \), then the residual graph of \( G_X \) has no path from \( s \) to \( t \). Hence, an extension of \( X \) should produce such a path in order to open for more flow from \( s \) to \( t \). We therefore consider the residual of the entire network \( G \), where also currently closed arcs are included, and search for a flow-augmenting path using one of the criteria explained in Section V.5.5. If such a path is found, its intersection with \( \bar{X} \) becomes the desired extension \( S \).

V.5.4 Extending an infeasible solution

Assume now that we have verified that \( x' \) does not satisfy the condition in 3. We conclude that \( X \) is infeasible, and we start the search for an extension of \( X \) that hopefully reestablishes feasibility.

It is easily seen that the suggested transformation \( x_{ij} = x'_{ij} + \ell_{ij} \) for all \( (i,j) \in X \) produces a solution violating the flow conservation constraints (V.2) for at least two nodes, whereas the flow bounds (V.3) are respected. More precisely, we can identify (at least) one node \( i^+ \) with excess entering flow, and (at least) one node \( i^- \) with shortage of entering flow. That is, we can find \( i^+ \) and \( i^- \) such that
\[
\sum_{j:(i,j) \in X} x_{ji} - \sum_{i:(j,i) \in X} x_{ij} > 0 \quad \text{and} \quad \sum_{j:(j,i) \in X} x_{ji} - \sum_{i:(i,j) \in X} x_{ij} < 0.
\]
The idea is now to extend $X$ in such a way that flow can be sent from $i^+$ to $i^-$. To this end, we search in $\overline{X}$ for arcs that create new paths joining the two unbalanced nodes.

Let $G'$ be the digraph consisting of all nodes and arcs in $G$ and $G'_X$, i.e., an extension of $G'_X$ where all $(i,j) \in \overline{X}$ are added. Define the capacities $c_{ij} = u_{ij} - \ell_{ij}$ also for all $(i,j) \in \overline{X}$. Assign the flow $x'$ (see Section V.5.1) to the network $G'$, and find a path $P = (N_P, A_P)$ with node set $N_P$ and arc set $A_P$ from $i^+$ to $i^-$ in the residual network $G'(x')$. If there is no such path, we conclude that $X$ cannot be extended. Otherwise, let $S = A_P \cap \overline{X}$ denote the set of new arcs along the path.

In general, $G'(x')$ may contain several flow-augmenting paths from $i^+$ to $i^-$. Care must be shown when selecting $P$, because once $S$ is accepted as extension of $X$, the new arcs will never leave again. A necessary condition for feasibility of $X \cup S$ is that the arcs $(i,j) \in S$ introduce a flow-augmenting path from $i^+$ to $i^-$ when added to the residual network $G'_X(x')$. This is however not a sufficient condition, because the lower bounds of the new arcs may provoke new infeasibilities. Taking the lower bounds into account when introducing new arcs seems to be hard to accomplish in rigorous terms, and will therefore be dealt with heuristically as explained in Section V.5.5.

It is not guaranteed that $X \cup S$ is feasible. This will be checked in the next iteration of Algorithm 6, which makes a call to the procedure discussed in Section V.5.1.

V.5.5 Finding paths in residual graphs

We have developed three methods for finding paths from $s$ to $t$ to extend a feasible solution, and from $i^+$ to $i^-$ to extend an infeasible solution. Their computational burdens are modest as they can be accomplished by a simple rewrite of Dijkstra’s shortest-path algorithm.

The ideal path is one where the smallest upper bound is large and the largest lower bound is small. Our path finding methods vary in the way they are adapted to this observation.

Maximizing the largest lower flow bound:

In the first method, we simply find a path $P = (N_P, A_P)$ for which $\max_{(i,j) \in A_P} \ell_{ij}$ is minimized. Excess flow (see Section V.5.4) and upper bounds do not affect the choice of path. The method is implemented by replacing the summation of path and arc lengths (lower flow bounds) in Dijkstra’s algorithm by a maximum operation over the two arguments.

Large difference between the smallest upper and the largest lower bounds:

In the second method, the ambition is to find a path $P$ with $\min_{(i,j) \in A_P} u_{ij} - \max_{(i,j) \in A_P} \ell_{ij}$ large, while also taking excess flow into account. Let $e_i$ denote the excess flow at node $i$, and for any $i \in N_P$, let $L_i$ and $U_i$ denote respectively the minimum and maximum amounts of flow node $i$ has to receive if all arcs in $A_P$ are opened. Then, for all $(i,j) \in A_P$, we have (let $L_s = -\infty$, $U_s = +\infty$) $L_j = \max \{ L_i + e_i, \ell_{ij} \}$ and $U_j = \min \{ U_i + e_i, u_{ij} \}$. As an approach
to making $U_i^+ - L_i^+$ (or $U_i - L_i$) large, we apply these recursive formulae in a heuristic inspired by Dijkstra as shown in Algorithm 7.

### Algorithm 7 LargeCapacityDifference($i^+, i^-$)

\[
\begin{align*}
    &d_i \leftarrow -\infty, \quad L_i \leftarrow 0, \quad U_i \leftarrow \infty, \quad \forall i \in N \\
    &d_{i^+} \leftarrow 0, \quad N_P \leftarrow \{i^+\}, \quad A_P \leftarrow \emptyset, \quad i \leftarrow i^+
\end{align*}
\]

repeat

\[
\begin{align*}
    &\text{for } \forall j \in N \setminus N_P : (i, j) \in A \\
    &\quad \text{do} \\
    &\quad L_j' \leftarrow \max\{L_i + e_i, \ell_{ij}\}, \quad U_j' \leftarrow \min\{U_i + e_i, u_{ij}\}
\end{align*}
\]

\[
\begin{align*}
    &\text{if } U_j' - L_j' > d_j \text{ then} \\
    &\quad L_j \leftarrow L_j', \quad U_j \leftarrow U_j', \quad d_j \leftarrow U_j' - L_j', \quad p_j \leftarrow p_i \\
    &\quad // \text{Best known path to } j \text{ goes via } i
\end{align*}
\]

\[
\begin{align*}
    &\text{end if}
\end{align*}
\]

\[
\begin{align*}
    &\text{end for}
\end{align*}
\]

\[
\begin{align*}
    &\text{Find } i \in \arg \max \{d_j : j \in N \setminus N_P\}
\end{align*}
\]

\[
\begin{align*}
    &N_P \leftarrow N_P \cup \{i\}, \quad A_P \leftarrow A_P \cup \{(p_i, i)\}
\end{align*}
\]

until $i^- \in N_P$

return $(N_P, A_P)$

---

**A hybrid method:**

Finally, we suggest a method combining the ideas of the two first methods. Consider the if-statement of Algorithm 7. Determining whether a new best path to $j$ has been found, is in the hybrid method based on a comparison either between $U_j' - L_j'$ and $d_j$ or between $L_j'$ and $L_j$. When $|L_j' - L_j|$ is small in comparison to $|U_j' - L_j' - (U_j - L_j)|$, then the decision is made as in Algorithm 7, otherwise we compare $L_j'$ to $L_j$. In our implementation, we have applied $|L_j' - L_j| < 3|U_j' - L_j' - (U_j - L_j)|$ as the criterion for using the rule of Algorithm 7.

**V.6 Computational experiments**

We have evaluated the construction heuristic in Section V.5 by applying it to a set of input graphs. The instances were obtained by first generating four base graphs $GW6$, $GW7$, $GW$ and $GL$, using the RMFGEN-generator of Goldfarb and Grigoriadis [4], which takes four parameter values $a$, $b$, $c_1$, and $c_2$ as input. The actual parameter values and graph sizes are shown in Table V.1. However, the generator was modified so that the capacities of the in-frame arcs are also generated randomly in the range $[c_2, c_2a^2]$. We generated 6 instances of each graph that differ by the percentage of arcs that have nonzero lower flow bounds. For example, $GL-40$ is generated such that 40% of the arcs have nonzero lower flow bounds. These arcs were selected randomly by drawing from the entire arc set $A$, and for each of them, $\ell_{ij}$ was randomly generated in the range $[u_{ij}/4, u_{ij}]$.

In Table VI.1, we give the optimal flow, produced by supplying the mixed integer programming formulation of Section V.4 to CPLEX, the upper bound given by the LP-relaxation of the same model, and the flow obtained by the three variants (see Section V.5.5) of the heuristic. The variants are in the table denoted 1, 2, and 3, consistent with the order in which they are introduced in Section V.5.5.
As seen from the table, the LP-bound is for all four base graphs tight when no more than 50% of the arcs have lower flow bounds. This means that in such instances, the minimum lot size constraints do not affect the maximum flow. For more constrained instances, the maximum flow drops below the bound, and the construction heuristic struggles to find other solutions than the trivial zero solution. When the proportion of arcs with a minimum lot size is small, the heuristic seems to have a fair chance to find good solutions. The results are inconclusive in the choice of variant of the heuristic.

We believe that the heuristic’s lack of success in highly constrained instances is due to its greedy nature: When the arcs along some path are opened, excess flow will be induced by their lower flow bounds. Whether it in later iterations will be possible to open paths to account for this excess flow seems difficult, and if judged incorrectly, the heuristic will be left with excessive flow for which there is insufficient capacity.
A more sophisticated procedure for arc selection could improve the construction heuristic. With many arcs subject to minimum lot size constraints, it becomes unlikely to find long paths where the largest lower flow bound is smaller than the smallest upper bound. Making a feasible flow allocation then requires that the flow is split at one or more nodes. Consequently, the heuristic should investigate more general subgraphs than paths when considering new arcs to be opened.

V.7 Conclusions

We have introduced the maximum flow problem in directed graphs with minimum lot size constraints on the arcs, imposing either zero flow or flow between the lower and upper capacity. The disjunctive nature of the new constraints makes the problem NP-hard. Based on a mixed integer programming formulation, we have shown how the problem can be approached by Lagrangian relaxation. This approach involves a method for computing strong upper bounds on the maximum flow, and a method for fixing binary variables based on the upper bounds. We have also suggested a construction heuristic for the problem, and presented results from some computational experiments.

This work will be followed up by experimental evaluation of the Lagrangian relaxation technique. Approaches for strengthening the construction heuristic will also be developed and tested numerically.
References


Paper VI

Parallel algorithms for the maximum flow problem with minimum lot sizes

Mujahed Eleyat, Dag Haugland, Magnus Lie Hetland and Lasse Natvig
In Operations Research Proceedings, 2011
Abstract

In many transportation systems, the shipment quantities are subject to minimum lot sizes in addition to regular capacity constraints. This means that either the quantity must be zero, or it must be between the two bounds. In this work, we prove that the maximum flow problem with minimum lot-size constraints on the arcs is strongly NP-hard, and we enhance the performance of a previously suggested heuristic. Profiling the serial implementation shows that most of the execution time is spent on solving a series of regular max flow problems. Therefore, we develop a parallel augmenting path algorithm that accelerates the heuristic by an average factor of 1.25.
VI.1 Introduction

Planning models for production, storage and transportation of goods often have to reflect minimum lot size constraints. The essence of such constraints is that if the activity level is not above some given lower bound, the operation in question must be inactive. Frequently, the computational consequence is that an otherwise tractable model becomes NP-hard.

In this work, we study the maximum flow problem subject to minimum lot size constraints. The purpose of the work is to suggest efficient computational methods that, despite the proven intractability of the problem, are able to produce near-optimal solutions with modest computational effort. To that end, we suggest a heuristic method based on a parallel implementation of a max flow algorithm.

It is well known that the maximum flow problem and its variants are challenging to parallelize. In fact, the maximum flow is P-complete, even for acyclic graphs [5, p. 152], which means that finding a highly parallel algorithm is very unlikely. This does not mean, however, that achieving speedups with a concurrent solution is impossible. Indeed, several such solutions exist. These algorithms tend to avoid the augmenting path approach, which at least intuitively seems inherently sequential. The other common approach, of pushing a preflow through the network [9] seems more amenable to parallelism. For example, already in 1982, Shiloach introduced a special-purpose concurrent max flow algorithm (with cubic sequential running time) based on this idea [11]. The classic algorithm of Goldberg [3] was also intended to be parallel even in its original version, and its parallel execution has since been improved [1]. The heuristic method suggested in the current work is based upon a parallel augmenting path algorithm.

VI.1.1 Problem definition

We let $G = (N, A)$ be a directed graph with node set $N$ and arc set $A$, and we let $\ell$ and $u$ be non-negative integer vectors of respectively lower and upper flow bounds. We assume that $G$ contains a unique source $s \in N$ and a unique sink $t \in N$, and that $A$ contains a circulation arc $(t, s)$ with $\ell_{ts} = 0$ and $u_{ts} = \infty$ from the sink to the source.

Let $x \in \mathbb{Z}^A_+$ be a flow vector, and consider the problem of maximizing $x_{ts}$ such that for all $(i, j) \in A$, either $x_{ij} = 0$ or $x_{ij} \in [\ell_{ij}, u_{ij}]$.

Define $F(G) = \{ x \in \mathbb{Z}^A_+ : \sum_{j:(i,j) \in A} x_{ij} - \sum_{j:(j,i) \in A} x_{ji} = 0 \ \forall i \in N \}$. For any arc set $X \subseteq A$, let $F_X(G) = \{ x \in F(G) : x_{ij} \in [\ell_{ij}, u_{ij}] \ \forall (i, j) \in X, x_{ij} = 0 \ \forall (i, j) \in \bar{X} \}$, where $\bar{X} = A \setminus X$. The problem is then expressed as

$$[P] \max_{X,x} \{ x_{ts} : X \subseteq A, x \in F_X(G) \}$$

We say that $X$ is feasible if $F_X(G) \neq \emptyset$. We let $G_X$ denote the subgraph with node set $N$ and arc set $X$. 

133
VI.2 Computational complexity

That network flow problems with semi-continuous flow variables are NP-hard is folklore. A formal proof was given in [6], and in this section, we improve the result by demonstrating strong NP-hardness.

**Proposition 4.** Problem \([P]\) is strongly NP-hard.

**Proof.** The proof is by a polynomial reduction from EXACT COVER BY 3-SETS (X3C), which is strongly NP-complete [2]. For the reduction to confer NP-hardness in the strong sense, we must also make sure that all numbers are polynomially bounded in the problem size and maximal numerical magnitude of the original (trivially satisfied in the following). Given a finite set \(Y = \{y_1, \ldots, y_n\}\), where \(n/3 = q\) is an integer, and a set \(C = \{C_1, \ldots, C_m\}\) of subsets of \(Y\), where \(|C_1| = \cdots = |C_m| = 3\), the X3C problem is to decide whether there exists some \(C' \subseteq C\) of pairwise disjoint subsets such that \(C'\) covers \(Y\). Consider any X3C-instance, and define the directed graph with node set \(N = \{s, v^c_1, \ldots, v^c_m, v^y_1, \ldots, v^y_n, t', t\}\). Define the arc sets: \(A_1 = \{(s, v^c_i) : y_j \in C_i \}$ for all \(i, j\), \(A_2 = \{(v^c_i, v^y_j) : y_j \in C_i \}$ for all \(i, j\), \(A_3 = \{(v^y_i, t') : y_j \in C_i \}$, and \(A_4 = \{(t', t)\}\), and \(A = A_1 \cup A_2 \cup A_3 \cup A_4\). An illustration of \(G = (N, A)\) is given by Parmar in his discussion of the subject [10], where it is used to prove NP-hardness of the EQUAL-SPLIT NETWORK FLOW PROBLEM. All arcs in \(A_1\) have lower and upper flow bounds 1, whereas all other arcs, except \((t', t)\), have zero lower flow bound and upper flow bound equal to \(\frac{q}{2}\). Arc \((t', t)\) has zero lower flow bound and capacity \(q\). It is easily verified that the X3C-instance is a yes-instance if and only if the optimal solution to problem \([P]\) applied to \(G\) is to send \(q\) units of flow along arc \((t', t)\). Hence, there exists a polynomial reduction from X3C to the decision version of \([P]\), and the proof is complete. \(\square\)

VI.3 A heuristic method

Since Proposition 4 discourages an exact algorithm, we will rather apply a heuristic solution method that may fail to find the optimal solution. The method is discussed in detail in [6], and reviewed briefly here. The idea is to start with \(X = \{(t, s)\}\), and gradually extend the set. Finding a larger, feasible \(X\) is non-trivial, and we will consider two kinds of extensions of \(X\): (1) Extending an infeasible \(X\) in order to reduce constraint violations, and (2) extending a feasible \(X\) in order to increase \(x_{ts}\).

Checking whether \(X\) is feasible, is accomplished by defining the digraph \(G'_X = (N', A')\), with the node set \(N' = N \cup \{s', t'\}\) consisting of all nodes in \(G\) and a new source \(s'\) and a new terminal \(t'\). The arc set \(A'\) consists of the arcs in \(X\), the circulation arc \((t', s')\), and the arcs \((s', i)\) and \((i, t')\) for all \(i \in N\). Define the arc capacities \(c_{ij} = u_{ij} - t_{ij}\) for all \((i, j) \in X\), \(c_{s't'} = +\infty\), \(c_{s'i} = \sum_{j:(i,j) \in X} \ell_{ji}\), and \(c_{it'} = \sum_{j:(i,j) \in X} \ell_{ij}\) for all \(i \in N\).

**Proposition 5.** Assume \(x' \in \mathbb{Z}^{A'}\) is a solution to the MAXIMUM FLOW PROBLEM in \(G'_X\) with source \(s'\), sink \(t'\), and arc capacities \(c\). Then the arc set \(X\) is feasible if and only if \(x'_{s't'} = \sum_{(i,j) \in X} \ell_{ij}\).
Proof. See Theorem 10.2.1 in [8]. □

For feasibility checking, Proposition 5 suggests that a max flow problem is solved in each iteration of our heuristic (Algorithm 8). If $X$ is feasible, we find an $x \in F_X(G)$ maximizing $x_{ts}$, and compare it to the best solution so far. We also try to improve $x$ by searching for a flow-augmenting path in $G$. If $X$ is infeasible, then $x'_{t's'} < \sum_{(i,j) \in X} \ell_{ij}$, and we search for a path from $s'$ to $t'$ in $G'$ that can increase $x'_{t's'}$. In conclusion, steps 3, 5, 9 and 11 involve computation of augmenting paths in $G$ or $G'$. For a fast implementation of the heuristic, we give a parallel augmenting path algorithm in the next section.

**Algorithm 8 Heuristic**($G, \ell, u, (t, s)$)

1: $X \leftarrow \{(t, s)\}$, $z^* \leftarrow 0$, $X^* \leftarrow X$
2: repeat
3: Compute $x'$ as given in Proposition 5.
4: if $x'_{t's'} = \sum_{(i,j) \in X} \ell_{ij}$ then
5: Compute $x \in \arg \max \{x_{ts} : x \in F_X(G)\}$
6: if $x_{ts} > z^*$ then
7: $z^* \leftarrow x_{ts}$, $X^* \leftarrow X$
8: end if
9: Let $B$ be the set of arcs of a flow-augmenting path from $s$ to $t$ in $G$
10: else
11: Let $B$ be the set of arcs of a flow-augmenting path from $s'$ to $t'$ in $G'$
12: end if
13: $X \leftarrow X \cup B$
14: until $B = \emptyset$
15: return $X^*$

**VI.4 Parallel implementation**

We chose to parallelize the augmenting path method and not the push-relabel one because the latter can not be used to perform step 5 of Algorithm 8. Another reason is that recent research [7] shows that parallel push-relabel methods have low speed-up in sparse instances.

Algorithm 9 shows our parallelization technique. In step 1, a set $S \subseteq N$ of start nodes is selected such that they have the same distance from the source. The selection is achieved using a modified version of breadth first search (BFS). In step 3, the paths from the source to $m \leq |S|$ start nodes are found using one BFS operation, and paths to the sink are found using $m$ BFS operations.

The efficiency of the algorithm is greatly dependent on the residual capacities of arcs shared by two or more different paths. Ideally, the residual capacity of each arc $a$ should be no smaller than the sum of the flow capacities of augmenting paths intersecting $a$. 

135
Algorithm 9 Parallel path augmentation max flow algorithm

1. Main thread: Select a set $S$ of start nodes in the residual graph $G(x)$
   repeat
   2. Main thread: Randomly choose $m$ nodes from $S$
   3. In parallel: main thread finds the shortest paths from the source to the $m$ nodes while other threads find the shortest paths from the $m$ nodes to the sink
   4. Main thread: Perform $m$ flow augmentations through the shortest paths
   until no more paths

VI.5 Performance and conclusion

We implemented the parallel algorithm using C++ and Pthreads and executed it on $2 \times 6$ Opteron processor. The instances were obtained using the RMFGEN-generator of Goldfarb [4] and 40% of the arcs were selected randomly to have nonzero lower bounds.

The speedup, calculated using the execution time of the serial algorithm as a baseline, is shown in Table VI.1. The speedup is not high mainly due to the fact that the paths found in parallel usually share an arc that gets saturated by augmenting flow through one of the paths. Moreover, and because we need to synchronize the threads every time they perform the small task of finding paths in a small sparse subgraph, parallel overhead is another reason for little speedup, especially when solving relatively small problems. In fact, this is the reason we chose not to parallelize BFS, which requires synchronizing the threads many times while finding a single path.

The performance can be enhanced by finding paths that don’t share arcs with small residual capacity. Finding such paths may not be easy and might require a lot of synchronization overhead. However, heuristic that let threads select different groups of arcs may also lead to better performance without increasing parallel overhead.

<table>
<thead>
<tr>
<th>Table VI.1. Performance Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>$G1$</td>
</tr>
<tr>
<td>$G2$</td>
</tr>
<tr>
<td>$G3$</td>
</tr>
<tr>
<td>$G4$</td>
</tr>
</tbody>
</table>
References


To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed is the main author of this paper and he did most of the work. His supervisor, Prof. Natvig, provided so many valuable comments that helped improving the paper.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

____ Trondheim 9 Mai 2014_________

Lasse Natvig
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/imre/search/phd/forms).

Candidate’s described contribution to:


Mujahed is the main author of this paper and he did most of the work. His supervisor, Prof. Natvig, provided so many valuable comments that helped improving the paper.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

__________
Trondheim 9 Mai 2014

[Signature]
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos. thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/imr/research/phd/forms).

Candidate's described contribution to:


Mujahed is the main author of this paper and he did most of the work. His supervisor, Prof. Natvig, provided so many valuable comments that helped improving the paper.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Trondheim 9 Mai 2014

Lasse Natvig
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed was the main author of this paper and he did most of the work. He discussed the approaches and results with the other authors and got many valuable comments.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Trondheim 9 Mai 2014

Lasse Natvig
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat's PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate's described contribution to:


Mujahed was the main author of this paper and he did most of the work. He discussed the approaches and results with the other authors and got many valuable comments.

Statement by the co-author:
I hereby confirm that the doctoral candidate's contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

[Signature]

JØRN AMUNDESEN
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/imr/research/phd/forms).

Candidate’s described contribution to:


Mujahed has implemented and tested two heuristic algorithms suggested by Dr. Haugland. In addition, he analyzed the results in cooperation with the other authors. He has also suggested and tested a hybrid algorithm that combines the two main algorithms.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Bergen, May 7, 2014

Dag Haugland

DAG HAUGLAND
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed has implemented and tested two heuristic algorithms suggested by Dr. Haugland. In addition, he analyzed the results in cooperation with the other authors. He has also suggested and tested a hybrid algorithm that combines the two main algorithms.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Trondheim, May 8th, 2014

Magnus Lie Hetland
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed was the main author of this paper and he did most of the work. He discussed the approaches and results with the other authors and got many valuable comments.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Bergen, May 7, 2014

Dag Haugland

Dag Haugland
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed was the main author of this paper and he did most of the work. He discussed the approaches and results with the other authors and got many valuable comments.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Trondheim, May 8th, 2019

[Signature]
MAGNUS LIE HEITLAND
To the evaluation committee

Co-authorship regarded publication included in Mujahed Eleyat’s PhD dr.philos thesis
(cf. the PhD regulations § 7.4, section 4 and the dr.philos regulations § 3, section 5, http://www.ntnu.edu/ime/research/phd/forms).

Candidate’s described contribution to:


Mujahed is the main author of this paper and he did most of the work. He discussed the approaches and results with the other authors and got many valuable comments.

Statement by the co-author:
I hereby confirm that the doctoral candidate’s contribution to this paper is correctly identified above, and I consent to Mujahed Eleyat including it in his PhD dr.philos. dissertation.

Trondheim 9 Mai 2014

Lasse Natvig