Performance Analysis of Cache-Aware Multicore Parallelization with Application to Optimization Theory

Kristoffer Stensen

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Jørn Aslak Amundsen, IDI
Co-supervisor: Mujahed Eleyat, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The master thesis looks at sparse matrices originating from optimization problems and tuning storage formats for cache-efficient access. It applies low level instrumentation tools for the development and analysis of cache-aware sparse matrix multiplication and cholesky decomposition in interior point methods. If time permits, it should be further looked into parallelization.

Abstract

In previous work, a cache-aware sparse matrix multiplication for linear programming interior point methods was proposed. The serial implementations achieved speedups ranging from 1.2 to 108.0 over the implementation in GLPK, an open-source linear programming solver. In this work, the same ideas and data structures are used to develop a cache-aware sparse cholesky decomposition as it is implemented in GLPK. The serial implementation achieves a speedup of up to 2.5 on the problem set considered. The matrix multiplication and cholesky decomposition are analysed by use of performance counters on both an AMD-based and an Intel-based system. The analysis shows that the applied blocking techniques reduce the number of floating point operations performed, and that this effect is even more important than the achieved cache utilization to produce speedup for some problems.

Preface

This thesis is submitted as fulfilment of the master's degree in Computer Science at the Department of Computer and Information Science at the Norwegian University of Science and Technology.

I would like to thank my supervisor Jørn Amundsen for guidance throughout the project, and especially for his dedication in making performance counters available for my work. I would also like to thank co-supervisor Mujahed Eleyat for providing the source code from his work on sparse matrix multiplication.

Trondheim, June 18, 2012

Kristoffer Stensen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Linear programming is widely applied in economics and business, and is involved in a significant portion of all scientific computations [11]. GNU Linear Programming Kit (GLPK) is an open-source library intended to solve large-scale linear programming problems, mixed integer programming problems, and other related problems [2]. Included in GLPK is a serial implementation of an interior point method for linear programming. A central part of the interior point method is solving a linear equation system of the form $(ADA^T)x = b$. Eleyat [10] proposed a cache-aware implementation of the multiplication ADA^T using blocking techniques, and also a load balance scheme for parallel execution.

In this work, the same idea and data structures developed for the blocked matrix multiplication are used to develop a blocked version of the cholesky decomposition, as it is implemented in the interior point method in GLPK. The cache behaviour of both the matrix multiplication and the cholesky decomposition are analysed by use of performance counters.

The report is structured as follows. Chapter 2 introduces linear programming, the interior point method and sparse matrix representations. The theory of numerical linear equation solving is given, and the blocked matrix multiplication is presented. Chapter 3 describes the blocked cholesky decomposition. Chapter 4 presents and discusses the results and cache-analysis. Conclusions and suggestions to future work are given in Chapter 5.

Chapter 2

Background

Linear programming problems often introduce large, sparse matrices, which are stored in special data structures. One such structure is the compressed row storage format, which is used in the interior point method of GLPK and is central in the development of the blocked sparse matrix multiplication [10]. The interior point method in GLPK solves linear programming problems by performing numerical linear algebra computations, such as matrix multiplication and cholesky decomposition, on such sparse matrices.

2.1 Linear Programming

Linear programming (or linear optimization) is a tool for finding the optimal value of a linear function, given a set of linear constraints. Linear programming uses a mathematical model, which in standard form can be expressed as

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b, \\ & x \geq 0, \end{aligned} \tag{2.1}$$

where $c^T x$ is the objective function to be minimized, x is the decision variables to be found, A is the constraint matrix, and c and b are constant vectors. The most common application is resource allocation, but any problem that fits the mathematical model, such as certain network flow problems, can be solved by linear programming.

2.2 Interior Point Method

The constraints in Equations 2.1 form a convex region that defines the set of feasible solutions to the problem. Contrary to the simplex method (see for example [6]), which finds the optimal solution by moving along the boundaries of the feasible region, the interior point method converges to an optimal solution through a path in the interior of the region. The interior point method is found to be superior to the simplex method for some large linear programming problems [11].

GLPK implements the primal-dual interior point method proposed by Mehrotra [13]. Figure 2.1 shows the computations of an iteration of the algorithm. The vectors x_k, y_k and z_k represent the estimate of solutions to Equations 2.1 and its dual problem. The dual problem is the equivalent problem *maximize* $z = b^T y$, *subject to* $A^T y + z$ *and* $z \geq 0$. X_k and Z_k are diagonal matrices with elements of x_k and z_k along their diagonals. The vector e represents $[1\,1\ldots1]^T$.

An iteration of the interior point method spends most of its time in a limited set of linear algebra kernels. Computation of lines 1.1 and 3.1 of Figure 2.1 involves solving linear equations of the form $(ADA^T)x = b$. As will be described in Section 2.4 this is done by performing matrix multiplication $S = ADA^T$, cholesky decomposition of S, followed by forward and backward substitutions. Other linear algebra routines involve matrix-vector multiplications of the form Ax and $A^T x$, and various vector-vector operations.

2.3 Sparse Matrix Representations

The straight-forward representation of a matrix saves the entries of the matrix contiguously in memory, either by rows or by columns. An element, a_{ij}, can then be directly accesses by row index i and column index j. For sparse

```
1. Compute affine scaling direction
   1.1 $dy_{\text{aff}} = (ADA^T)^{-1}(AZ_k^{-1}(X_k(c - A^T y_k - z_k) + X_k Z_k e) + b - Ax_k)$
   1.2 $dx_{\text{aff}} = Z_k^{-1}(X_k(A^T dy_{\text{aff}} - c + A^T y_k + z_k) - X_k Z_k e)$
   1.3 $dz_{\text{aff}} = X_k^{-1}(-X_k Z_k e - Z_k dx_{\text{aff}})$
2. Compute the centering parameter $\sigma$
   2.1 $\alpha_{\text{aff-p}} = \inf\{0 \leq \alpha \leq 1 | x_k + \alpha dx_{\text{aff}} \geq 0\}$
   2.2 $\alpha_{\text{aff-d}} = \inf\{0 \leq \alpha \leq 1 | z_k + \alpha dz_{\text{aff}} \geq 0\}$
   2.3 $\mu_{\text{aff}} = (x_k + \alpha_{\text{aff-p}} dx_{\text{aff}})^T (z_k + \alpha_{\text{aff-d}} dz_{\text{aff}})$
   2.4 $\sigma = (\mu_{\text{aff}}/\mu)^3$, where $\mu = x_k^T z_k / n$
3. Compute the centering direction
   3.1 $dy_{\text{cc}} = (ADA^T)^{-1}(AZ_k^{-1}(-(\sigma\mu e - X_k Z_k e)))$
   3.2 $dx_{\text{cc}} = Z_k^{-1}(X_k(A^T dy_{\text{cc}}) + (\sigma\mu e - X_k Z_k e))$
   3.3 $dz_{\text{cc}} = X_k^{-1}((\sigma\mu e - X_k Z_k e) - Z_k dx_{\text{cc}})$
4. Compute the combined direction
   4.1 $dx = dx_{\text{aff}} + dx_{\text{cc}}$
   4.2 $dy = dy_{\text{aff}} + dy_{\text{cc}}$
   4.3 $dz = dz_{\text{aff}} + dz$
5. Determine primal and dual step sizes
   5.1 $\alpha_p = \gamma \cdot \inf\{0 \leq \alpha \leq 1 | x_k + \alpha dx \geq 0\}$
   5.2 $\alpha_d = \gamma \cdot \inf\{0 \leq \alpha \leq 1 | z_k + \alpha dz \geq 0\}$, where $\gamma = 0.90$
6. Compute next point
   6.1 $x_{k+1} = x_k + \alpha_p dx$
   6.2 $y_{k+1} = y_k + \alpha_p dy$
   6.3 $z_{k+1} = z_k + \alpha_p dz$
```

Figure 2.1: Iteration k of the IPM algorithm

matrices, i.e. matrices primarily populated with zeroes, this storage scheme becomes inefficient, both in terms of memory requirements and operations performed on the matrix. For such matrices, only the non-zero elements need to be saved and trivial operations resulting in zero can often be avoided.

A wide variety of storage techniques for sparse matrices has been proposed [14]. Some storage formats are optimized for matrices with specific structures, such as banded matrices, while others are general and support matrices of any structure. The general idea is to store only the non-zero entries and determine their position in the matrix through an indexing structure. Hence, entries are not accessed directly by their row and column indices.

The cost of the savings in memory, memory referencing and computations is that algorithms become more complicated. Because of the indirect indexing pattern of sparse formats, these algorithms can be much more challenging to optimize for modern memory hierarchies than algorithms for the traditional matrix format.

2.3.1 Compressed Row Storage

The Compressed Row Storage (CRS) is a general storage format with no requirements for the matrix structure. An $m \times n$ matrix with n_{nz} non-zero elements is represented by three arrays, A_{val}, A_{ind} and A_{ptr}. A_{val} contains the n_{nz} non-zero entries, where rows are stored in order, and A_{ind} holds the column indices for each of the values in A_{val}. A_{ptr} contains the n starting positions of each row of A in A_{val} and A_{ind}, and the number of non-zeroes plus one, $n_{\text{nz}} + 1$, at the $(n+1)$th position. Within a row, the entries need not be stored in the same order as in A. The Compressed Column Storage (CSS) is a format similar to CRS where values are stored column-by-column and A_{ptr} contains column positions. A matrix A in CSS format equals its transpose, A^T, in CRS format. The CRS format reduces the memory requirement of mn storage locations for the full matrix to $2n_{nz} + n + 1$ storage locations. Figure 2.2 shows a matrix A and its CRS storage.

$$A = \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 3 & 1 & 0 \\ 8 & 0 & 3 & 0 \\ 7 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{aligned} A_{\text{val}} &= (2,3,1,3,8,7) \\ A_{\text{ind}} &= (4,2,3,3,1,1) \\ A_{\text{ptr}} &= (1,2,4,6,7) \end{aligned}$$

Figure 2.2: A matrix A in compressed row storage

2.4 Linear Solver

An equation of the form $Ax = b$ is solved by factorizing A into a product of a lower-triangular and and an upper-triangular matrix, $A = LU$. When A is symmetric ($A = A^T$) and positive definite ($c^T Ac > 0$ for all non-zero vectors c), L can be chosen so that $L = U^T$. This is the cholesky decomposition of A. The equation is now solved by first solving $U^T y = b$ (forward substitution) and then solving $Ux = y$ (backward substitution).

The interior point method spends most of the iteration time solving the linear equation system of the form $(ADA^T)x = b$. First, the matrix product

$$S = PAD(PA)^T \tag{2.2}$$

is calculated. P is a permutation matrix saved as a one-dimensional array π, such that row i of A is interchanged with row $\pi(i)$. P is calculated using some ordering algorithm and is used to reduce fill-in, i.e. entries that change from zero to a non-zero value during the cholesky decomposition [8]. Next, S is factorized into the product

$$S = U^T U. \tag{2.3}$$

Matrix A is saved in CRS format and vectors and diagonal matrix $D = XZ^{-1}$ are saved in full one-dimensional arrays, including their zero-entries. The upper-triangular part of matrices S and U are stored in CRS format, and their diagonal entries are stored in arrays S_{diag} and U_{diag}.

Both the matrix multiplication and the decomposition are preceded by a symbolic phase which finds the structures of S and U, respectively. The symbolic phases find S_{ind}, S_{ptr}, U_{ind} and U_{ptr} and are performed once, before the iterations of the interior point method are carried out. The numeric phases of the matrix multiplication and the decomposition, which calculate the actual values of S_{val} and U_{val}, are performed in each iteration.

2.5 Sparse Matrix Multiplication

For a matrix A saved in the traditional way, a multiplication of the form $S = PAD(PA)^T$ has a trivial implementation. For each element s_{ij}, the dot product, multiplied by entries of the diagonal matrix D, of row $\pi(i)$ and row $\pi(j)$ of A is computed:

$$s_{ij} = \sum_{k=1}^{n} a_{\pi(i),k} d_{k,k} a_{\pi(j),k} \tag{2.4}$$

An element can be directly accessed by indices i and j, so the iterator variables of three nested loops can be used for indexing.

When A and S are in CRS format, elements must be accessed in a way to avoid searching for entries. The beginning of a row can be accessed directly, all non-zero entries of a row can be iterated over, and for each of these entries its column index can be found directly. GLPK implements the multiplication row by row, as described in Algorithm 1. For each row i, the $\pi(i)$th row of A is decompressed into a full vector in an array w. Then, for each entry, s_{ij}, in row i, the dot product is computed between row $\pi(j)$ of A and w. The dot product is carried out by iterating over row $\pi(j)$ of A while reading the corresponding entry of row $p(i)$ out of w.

Algorithm 1 Computing $S = PAD(PA)^T$

```
 1: for $i = 1 \to m$ do
 2:     $i_p = \pi(i)$
 3:     for $k = A_{\text{ptr}}(i_p) \to A_{\text{ptr}}(i_p + 1)$ do
 4:         $w(A_{\text{ind}}(k)) = A_{\text{val}}(k)$
 5:     end for
 6:     for $k = S_{\text{ptr}}(i) \to S_{\text{ptr}}(i + 1)$ do
 7:         $j = S_{\text{ind}}(k)$
 8:         $j_p = \pi(j)$
 9:         for $l = A_{\text{ptr}}(j_p) \to A_{\text{ptr}}(j_p + 1)$ do
10:             $S_{\text{val}}(k) = S_{\text{val}}(k) + A_{\text{val}}(l) \cdot D(A_{\text{ind}}(l)) \cdot w(A_{\text{ind}}(l))$
11:         end for
12:     end for
13:     for $k = A_{\text{ptr}}(i_p) \to A_{\text{ptr}}(i_p + 1)$ do
14:         $S_{\text{diag}}(i) = S_{\text{diag}}(i) + A_{\text{val}}(k) \cdot D(A_{\text{ind}}(k)) \cdot A_{\text{val}}(k)$
15:     end for
16: end for
```

Eleyat [10] has suggested modifications to improve the performance of the multiplication based on the following observations: First, values of D and w needed to compute non-zero entries in S are scattered irregularly over large arrays and might cause a high cache miss rate. Second, the permutation of A makes it difficult to benefit from data locality and might cause translation lookaside buffer (TLB) misses. Two partitioning schemes were presented to avoid these problems, a one-dimensional and a two-dimensional partitioning.

2.5.1 One-dimensional Matrix Partitioning

The multiplication can be done in vertical blocks so that parts of w and D can be kept in memory during the multiplication of a block. Using block matrix multiplication, AA^T can be computed as

$$AA^T = \begin{bmatrix} A_1 & A_2 & \dots & A_k \end{bmatrix} \begin{bmatrix} A_1^T \\ A_2^T \\ \vdots \\ A_k^T \end{bmatrix} = \begin{bmatrix} A_1A_1^T + A_2A_2^T + \dots + A_kA_k^T \end{bmatrix}$$

where A is partitioned into to k vertical blocks. Matrix D is left out of the equation for simplicity. Since entries of A are not directly accessible, sparse block matrix multiplication requires a new structure for A to avoid having to search for the beginning of a block in a row. The matrix A is therefore divided into k matrices represented by separate CRS structures. In addition, another level of CRS indexing is introduced to avoid looking up zero-rows of a block. This higher-level CRS treats non-zero rows of a partition as entries and indexes the partition number instead of column number. The permutation of A is performed during the partitioning so that A is already permuted when multiplied.

2.5.2 Two-dimensional Matrix Partitioning

The vertical partitions of A can be further further divided to achieve a two-dimensional blocking of A. When partitioned into four sub-matrices, AA^T is computed as

$$AA^T = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{bmatrix} = \begin{bmatrix} A_{11}A_{11}^T + A_{12}A_{12}^T & A_{11}A_{21}^T + A_{12}A_{22}^T \\ A_{21}A_{11}^T + A_{22}A_{12}^T & A_{21}A_{21}^T + A_{22}A_{22}^T \end{bmatrix}.$$

A is partitioned into $M \times N$ blocks and S into $M \times M$ blocks. To exploit that the sparsity structure of S is known, each block of S keeps an array of indices of non-zero rows. In addition, S has an array of indices of participating pairs of blocks of A. Two blocks are participating if the result of their multiplication is non-zero. These arrays are calculated once, before the iterations of the interior point method begin.

2.5.3 Parallel Implementation

The matrix multiplication is computed in parallel by calculating rows of S concurrently. Since rows have different sparsity, dividing an equal amount of rows to each processor leads to load imbalance. Instead, S is divided into a number of shares that equals the number of processors. Ideally, all shares have the same number of non-zeros. A share is made up of consecutive rows in the vertical partitioning case, and a number of consecutive rows of blocks in the two-dimensional block case.

Chapter 3

Cache-aware Cholesky Decomposition

State of the art algorithms for sparse cholesky decomposition are based on supernodal and multifrontal methods that make use of optimized dense BLAS [12] routines to reach near peak performance of the hardware [7]. In the following, the ideas and structures developed for the sparse matrix multiplication are used to make a more cache-aware version of the cholesky decomposition as implemented in GLPK.

3.1 Original Algorithm

The factorization (Equation 2.3) is a form of Gaussian elimination performed row by row. For the $n \times n$ matrix S, the algorithm follows from

$$S = \begin{bmatrix} s_{11} & s_{12} \\ s_{12}^T & S_{22} \end{bmatrix} = \begin{bmatrix} u_{11} & 0 \\ u_{12}^T & U_{22}^T \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & U_{22} \end{bmatrix} = U^T U$$

where s_{11} is the upper-left entry of S, s_{12} is the row vector consisting of the remaining entries of the first row, and S_{22} is the remaining sub-matrix. Carrying out the matrix multiplication, gives the following equations:

$$s_{11} = u_{11} u_{11} \tag{3.1}$$

$$s_{12} = u_{11} u_{12} \tag{3.2}$$

$$S_{22} = u_{12}^T u_{12} + U_{22}^T U_{22}. \tag{3.3}$$

u_{11} is found as $\sqrt{s_{11}}$ and u_{12} equals s_{12}/u_{11}. Rearranging Equation 3.3 to $U_{22}^T U_{22} = S_{22} - u_{12}^T u_{12}$, U_{22} is found by recursively applying the algorithm on the matrix $S_{22} - u_{12}^T u_{12}$. The main work of the decomposition is updating the sub-matrix, U_{22}, by subtracting the outer product $u_{12}^T u_{12}$ (note that u_{12} is a row vector).

Entries of S are copied into U before the algorithm starts. The sparse implementation is performed row by row as shown in Algorithm 2. The transformed row i of U is unpacked into array w. For each non-zero entry in $u_{12}^T u_{12}$, the outer product between elements in w is subtracted from U_{22}.

Algorithm 2 Computing factorization $S = U^T U$

```
1:  for $i = 1 \to m$ do
2:      $U_{\text{diag}}(i) = \sqrt{U_{\text{diag}}(i)}$
3:      for $k = U_{\text{ptr}}(i) \to U_{\text{ptr}}(i+1)$ do
4:          $w(U_{\text{ind}}(k)) = U_{\text{val}}(k) = U_{\text{val}}(k)/U_{\text{diag}}(i)$
5:      end for
6:      for $k = U_{\text{ptr}}(i) \to U_{\text{ptr}}(i+1)$ do
7:          $j = U_{\text{ind}}(k)$
8:          for $l = U_{\text{ptr}}(j) \to U_{\text{ptr}}(j+1)$ do
9:              $U_{\text{val}}(l) = U_{\text{val}}(l) - w(j) \cdot w(U_{\text{ind}}(l))$
10:         end for
11:         $U_{\text{diag}}(j) = U_{\text{diag}}(j) - w(j) \cdot w(j)$
12:     end for
13: end for
```

3.2 Blocked algorithm

Blocking matrix operations to exploit reuse of data in cache is a frequently used technique for dense matrices [9]. The same strategy used in the cache-aware sparse matrix multiplication can be applied to the sparse cholesky decomposition. Similar to the multiplication case, a vector w is used throughout the computation, so that the original cholesky decomposition can suffer from the same cache issues mentioned in Section 2.5. Blocking S as

$$S = \begin{bmatrix} S_{11} & S_{12} \\ S_{12}^T & S_{22} \end{bmatrix} = \begin{bmatrix} U_{11}^T & 0 \\ U_{12}^T & U_{22}^T \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = U^T U$$

where S_{11} is $k \times k$, S_{12} is $k \times (n-k)$ and S_{22} is $(n-k) \times (n-k)$, gives the following equations:

$$S_{11} = U_{11}^T U_{11} \tag{3.4}$$

$$S_{12} = U_{11}^T U_{12} \tag{3.5}$$

$$S_{22} = U_{12}^T U_{12} + U_{22}^T U_{22} \tag{3.6}$$

Both U_{11} and U_{12} in Equations 3.4 and 3.5 can be found by performing Algorithm 2 on $[S_{11} S_{12}]$. Rearranging Equation 3.6 to $U_{22}^T U_{22} = S_{22} - U_{12}^T U_{12}$, U_{22} is found by recursively running the algorithm on matrix $S_{22} - U_{12}^T U_{12}$. Updating the sub-matrix now involves a matrix multiplication instead of the outer product in the unblocked algorithm. This matrix multiplication can be performed on vertical blocks in the same way as the vertically partitioned multiplication described in Section 2.5.1, so that the same data structure can be used for the blocked cholesky decomposition.

Algorithm 2 is used to compute U_{12}, which is needed to update sub-matrix U_{22}. Since matrices in CRS format are accessed in rows, it is better to have available its transpose, U_{12}^T, to efficiently compute $U_{12}^T U_{12}$. Therefore, U_{12} is transposed before the matrix update. This is equivalent to saying that U_{12} is transformed from compressed row storage to compressed column storage so that columns of U_{12} can be efficiently accessed. Algorithm 3 shows the steps to transpose a matrix A. Steps 1, 2 and 7 are performed once and are not repeated trough the iterations.

Algorithm 3 Find $n \times m$ matrix $B = A^T$, given $m \times n$ matrix A

```
 1: Determine row counts of $B$, $w(i)$=number of entries in row $i$ of $B$
 2: Find the cumulative sum, $w(i+1) = w(1) + \cdots + w(i)$
 3: for $i = 1 \to m$ do
 4:     for $k = A_{\text{ptr}}(i) \to A_{\text{ptr}}(i+1)$ do
 5:         $p = w(A_{\text{ind}}[k])$
 6:         $w(A_{\text{ind}}[k]) = w(A_{\text{ind}}[k]) + 1$
 7:         $B_{\text{ind}}(p) = i$
 8:         $B_{\text{val}}(p) = A_{\text{val}}(k)$
 9:     end for
10: end for
```

Since the pattern of U is known, the CRS structures of the transposed blocks can be decided before the interior point method iterations begin. This is done by first transposing the upper-triangular matrix U to a lower-triangular matrix L, and then dividing L into vertical partitions of size k.

3.3 Parallel Implementation

In each recursive step, the update of sub-matrix U_{22} can be parallelized, both in the original and the blocked version. Distribution of work between processors can be pre-calculated as done with the matrix multiplication in Section 2.5.3. Some complications occur when applying the same strategy for the cholesky decomposition. First, the shares need to be decided for each recursive step. Second, dividing the shares to include an equal amount of non-zeroes in U_{22} is not necessarily as effective, since the work performed for each non-zero entry can be varying from one recursive step to the next. Instead, rows of U_{22} are distributed such that each processor is assigned an equal amount of work in form of floating point operations.

Chapter 4

Results and Discussion

The sparse matrix multiplication and cholesky decomposition have been analysed by use of performance counters on two different hardware architectures, one of which is the same system as was used to develop the sparse matrix multiplication [10].

Some comments on the original results of Eleyat [10] are appropriate. When reproducing the results it was found that the parallel execution times reported [10, pp. 437, Figures 5-9] measure the initial point calculation in addition to the first iteration of the interior point method. Hence, the timings include two invocations of the multiplication. Also, results were of code compiled in 32-bit, which resulted in higher execution times on multiple cores than was the case for the same source code compiled in 64-bit. All results reported in this chapter are of code compiled in 64-bit.

4.1 Test Environments

The first system is a Linux Rocks cluster running CentOs 5.3. Each node is equipped with two 6-core 2.4 GHz AMD Opteron 2431 processors, adding up to 12 cores per node. As shown in Figure 4.1 each core has a 64KB L1 and a 512KB L2 cache and all cores share a 6MB L3 cache. Results are collected from a dedicated node with a custom kernel compiled with support for performance counters [1]. All code is compiled with GCC 4.4.3, optimization level 3. This is the same system as was used to develop the sparse matrix multiplication.

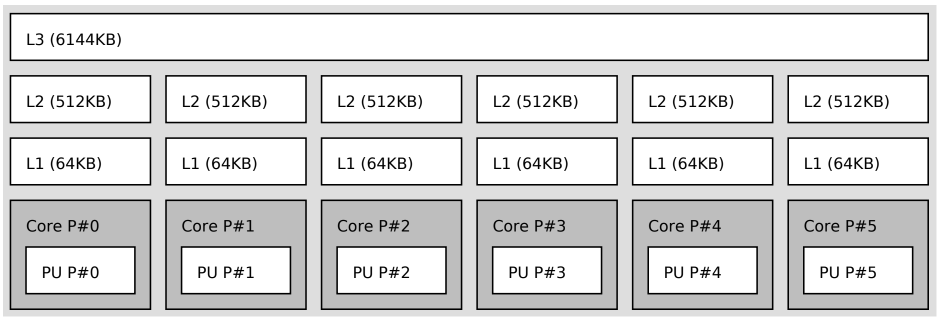


Figure 4.1: Cache hierarchy on the AMD Opteron 2431

The second system is running SUSE Linux Enterprise Server 11. The compute nodes are equipped with two 2.93GHz Intel Xeon X5570 (Nehalem) quad-core processors. Each core has a 32KB L1 and a 256KB L2 cache and all cores share a 8MB L3 cache, as shown in Figure 4.2. All code is compiled with GCC 4.3.4, optimization level 3.

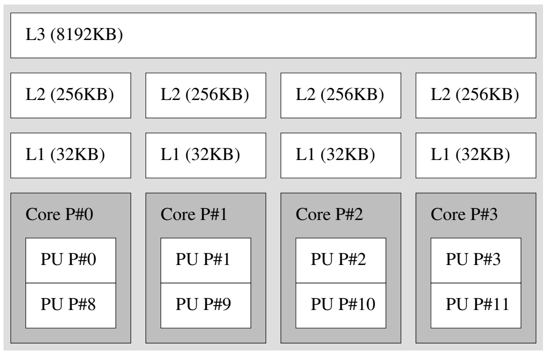


Figure 4.2: Cache hierarchy on the Intel Xeon X5570

Performance Application Programming Interface (PAPI) [5] has been used to read the performance counters on the processors. The AMD-based system has no support for L3-counters, and so only L1 and L2 results are discussed for this machine. The following PAPI-counters have been used in the results. PAPI_L1_DCA, PAPI_L2_DCA, PAPI_L1_DCM and PAPI_L2_DCM are used to read the L1 and L2 data cache accesses, and L1 and L2 data cache misses, respectively. PAPI_L3_TCA and PAPI_L3_TCM are used to read the total L3 cache accesses and misses on the Intel core. Floating point operations are counted with PAPI_FP_OPS.

4.2 Problem Set

The test problems are taken from the Netlib LP test problem set [4] and the BPMPD website [3]. The dimensions of constraint matrix A and the sparsity of A, S (see Equation 2.2) and U (see Equation 2.3) for each problem are shown in Table 4.1. FIT2D, NSCT2 and BAS1LP were also used to asses the matrix multiplication in the work of Eleyat [10]. The problems are selected to represent different sparsity structures.

Table 4.1: Dimensions and sparsity of the problem set matrices

Name	**Dimensions of A**	**Sparsity of A [%]**	**Sparsity of S [%]**	**Sparsity of U [%]**
FIT2D	10525×21024	0.07%	0.13%	0.13%
NSCT2	23003×37563	0.08%	0.79%	1.41%
BAS1LP	9872×14286	0.42%	1.94%	4.31%
DFL001	6084×12243	0.05%	0.12%	4.23%
BAXTER	28563×31855	0.01%	0.08%	0.89%

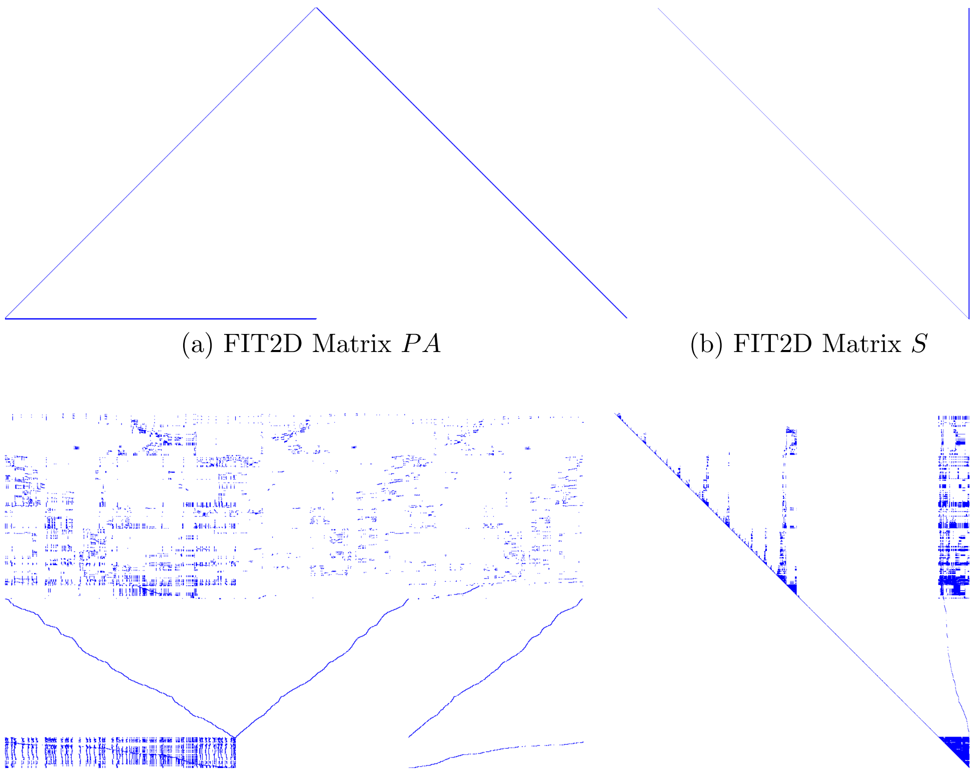
The sparsity patterns of the problem matrices are given in Figure 4.3. The figure depicts the non-zero structure of the permuted matrix PA and matrix S for each problem. Permutation matrix P for all problems is found by the minimum degree ordering algorithm [8], which is the default in GLPK.

4.3 Original GLPK

Table 4.2 shows the execution times of the original GLPK implementations of the matrix multiplication and the cholesky decomposition for one iteration on the AMD core. The numbers show that most time is spent computing the matrix multiplication and cholesky decomposition for these problems.

Table 4.2: Original GLPK execution times for matrix multiplication and cholesky decomposition together with the percentage it makes up of the total iteration time

Name	**Multiplication $\mathbf{S = PAD(PA)^T}$ [s]**	**Cholesky decomposition [s]**
FIT2D	2.35 (99.2%)	$7.32 \cdot 10^{-3}$ (0.3%)
NSCT2	1.85 (5.5%)	$3.18 \cdot 10^{1}$ (93.9%)
BAS1LP	$8.97 \cdot 10^{-1}$ (11.3%)	6.96 (87.5%)
DFL001	$5.68 \cdot 10^{-3}$ (0.2%)	3.03 (98.8%)
BAXTER	$1.42 \cdot 10^{-1}$ (0.7%)	$1.88 \cdot 10^{1}$ (98.5%)



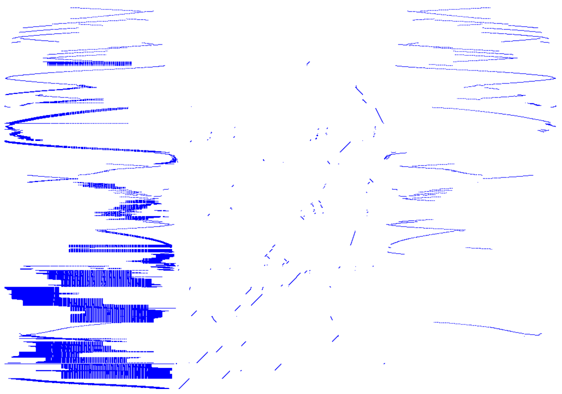
(a) FIT2D Matrix PA

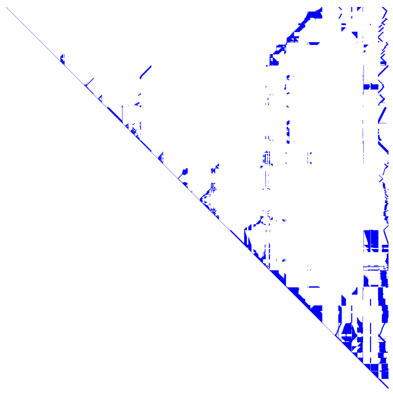
(b) FIT2D Matrix S

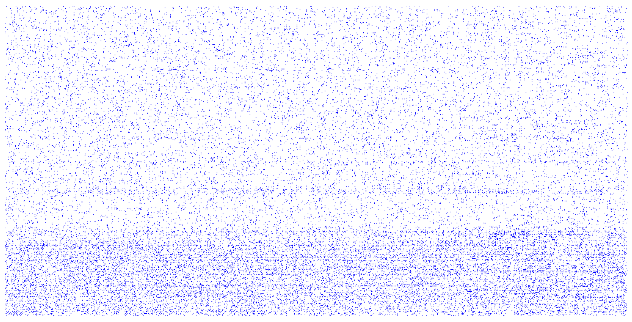
(c) NSCT2 Matrix PA

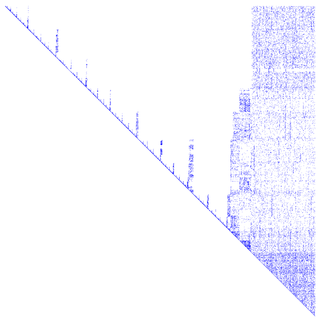
(d) NSCT2 Matrix S

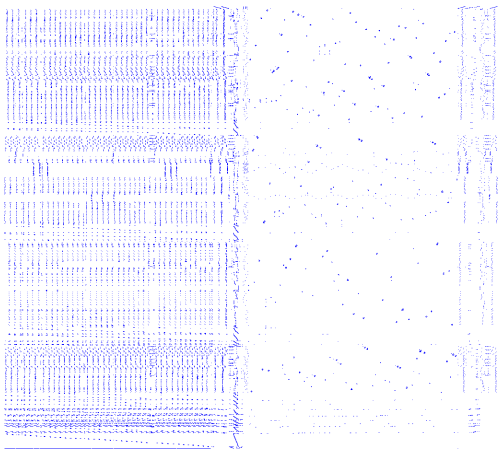
Figure 4.3: Sparsity patterns of matrix A after permutation and matrix S. Non-zero entries are coloured


(e) BAS1LP Matrix PA


(f) BAS1LP Matrix S


(g) DFL001 Matrix PA


(h) DFL001 Matrix S


(i) BAXTER Matrix PA

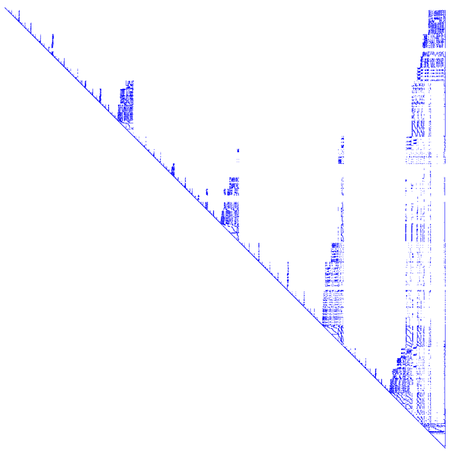

(j) BAXTER Matrix S

Figure 4.3: Sparsity patterns of matrix A after permutation and matrix S. Non-zero entries are coloured

4.4 Matrix Multiplication

Table 4.3 shows the speedups for the one-dimensionally (1D) and the two-dimensionally (2D) partitioned matrix multiplication when run on one core. Block sizes are fixed to 100 and 100 $\times$ 100 for the one dimensional and two dimensional versions, respectively. Speedup is achieved for FIT2D, NSCT2, BAS1LP and BAXTER. Results are comparable on the AMD core and Intel core. The preferred partitioning scheme varies from problem to problem. The remainder of this section will discuss the results of the one-dimensional blocking.

Table 4.3: Speedup achieved with the one-dimensionally and two-dimensionally partitioned matrix multiplication on the AMD and Intel core, and the ratio of floating point operations performed in the original GLPK implementation to the floating point operations performed in the blocked versions

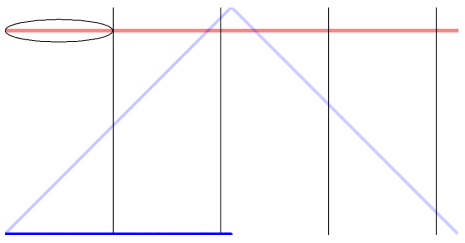
	Speedup AMD core		**Speedup Intel core**		**FP ops. ratio**	
Name	**1D**	**2D**	**1D**	**2D**	**1D**	**2D**
FIT2D	94.12	102.62	86.22	90.23	83.30	84.05
NSCT2	1.25	1.68	1.33	1.65	2.08	2.15
BAS1LP	1.19	1.55	1.28	1.50	1.61	1.65
DFL001	0.83	0.28	0.86	0.28	4.69	4.58
BAXTER	3.94	2.06	5.07	2.83	15.23	14.51

Also shown in Table 4.3 is the ratio of floating point operations in the original GLPK implementation to floating point operations in the blocked version. Interestingly, the ratios reveal that the blocked implementations perform much less floating point multiplications for some of the problems. Most extreme is the problem FIT2D for which over eighty times as many multiplications are performed for the original GLPK implementation, and speedups of 94 and 102 are achieved for the vertical and blocked partitioning, respectively, on the AMD core.

As can be seen from Figure 4.3a, FIT2D has a special sparsity structure. The matrix PA is mostly empty, except for entries along two diagonal bands and a few dense rows at the bottom. In the development of the blocked implementation, the high speedup was assumed to be caused by effective cache reuse in the dense rows [10, pp. 436]. However, the fact that the number of floating point operations is reduced, was not considered.

Figure 4.4 illustrates the computation of a row in the matrix multiplication of FIT2D. Non-zero entries are coloured blue, and the row being computed is coloured red. Denser areas of the matrices are coloured with higher opacity. In the original GLPK implementation, the entire row of PA in Figure 4.4a is copied into an array. Next, the non-zero entries in the same row of S in Figure 4.4b are traversed. Matrix S is empty except at its diagonal and a group of adjacent dense columns at the right edge. When calculating an entry of these dense columns, a dense row from PA is traversed and multiplied by the previously extracted array that mostly contains zeroes. Most of these multiplications result in zero.

Now, consider the vertical partitioning introduced as black lines in Figure 4.4a. Using the higher level CRS, the entire first partition is discarded when computing the row, avoiding the multiplication between part of the dense row and an array of zeroes (circled in Figure 4.4a). This saves a significant amount of multiplications, since almost every row of S involves multiplying the dense rows of A with an almost empty row in the original GLPK implementation.


(a) Matrix PA

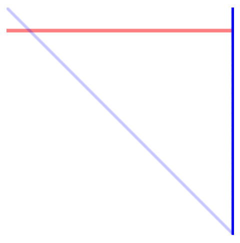

(b) Matrix U

Figure 4.4: Performing one row of matrix multiplication for FIT2D

Since the blocked versions perform less operations than the original GLPK implementation, their cache performances are not directly comparable. The blocked algorithms perform extra indexing for the blocks and thus introduce more memory accesses. Table 4.4 shows the ratio of memory accesses to floating operations performed for the original and the vertically partitioned matrix multiplication. For DFL001, for which speedup is not achieved, the blocked version performs a much higher number of memory accesses on average for each floating point operation.

Table 4.5 and Table 4.6 present the cache miss rates for the vertical partitioning on the AMD core and the Intel core, respectively. The blocked version has a similar effect on the cache misses on both systems. As expected, L1 cache misses are reduced. However, the blocked version does not seem to

Table 4.4: Ratio of memory accesses to floating point operations for the original GLPK matrix multiplication and the vertically partitioned matrix multiplication

Name	**Original**	**1D Partition**
FIT2D	1.45	1.46
NSCT2	1.40	2.06
BAS1LP	1.38	1.63
DFL001	1.71	8.62
BAXTER	1.42	3.07

exploit L2 and L3 cache as well as the original GLPK implementation, as the percentage of cache misses is increased for these levels.

Table 4.5: Level 1 and level 2 cache misses for the original GLPK matrix multiplication and the vertically partitioned matrix multiplication on the AMD core

	Original		**1D Partition**	
Name	**L1** [%]	**L2** [%]	**L1** [%]	**L2** [%]
FIT2D	2.25	1.26	0.37	20.10
NSCT2	2.18	9.53	1.26	21.13
BAS1LP	1.77	5.43	1.22	22.12
DFL001	22.27	3.04	12.36	35.18
BAXTER	3.08	0.72	3.96	12.37

Table 4.6: Level 1, level 2 and level 3 cache misses for the original GLPK matrix multiplication and the vertically partitioned matrix multiplication on the Intel core

	Original			**1D Partition**		
Name	**L1** [%]	**L2** [%]	**L3** [%]	**L1** [%]	**L2** [%]	**L3** [%]
FIT2D	12.85	11.53	0.04	1.10	18.00	10.20
NSCT2	8.22	17.35	2.30	4.66	19.06	3.81
BAS1LP	7.65	7.61	1.68	4.53	19.53	1.64
DFL001	31.47	18.57	5.14	12.82	61.82	12.73
BAXTER	12.35	5.83	0.76	6.76	35.44	19.77

A change in the cache miss rate of higher level caches affects the access pattern in the lower levels. Reducing misses in L1 cache reduces the L2 cache accesses, which again affects the cache miss rate of L2 cache. The same applies for L3 cache. Table 4.7 presents the percentage of cache misses relative to the total number of L1 accesses for each level of cache on the Intel core. Although cache misses are increased in L2 and L3 for most of the problems, the effect is not as significant as the effect in L1 cache. In the case of NSCT2, cache utilization is improved in all levels of cache.

Table 4.7: Level 1, level 2 and level 3 cache misses relative to the total (L1) cache accesses for the original GLPK matrix multiplication and the vertically partitioned matrix multiplication on the Intel core

	Original			**1D Partition**		
Name	**L1 [%]**	**L2 [%]**	**L3 [%]**	**L1 [%]**	**L2 [%]**	**L3 [%]**
FIT2D	12.85	1.06	3.66×10^{-4}	1.10	0.20	1.64×10^{-2}
NSCT2	8.22	1.66	2.33×10^{-2}	4.66	0.90	2.03×10^{-2}
BAS1LP	7.65	0.62	1.04×10^{-2}	4.54	0.89	1.03×10^{-2}
DFL001	31.47	6.58	2.20×10^{-1}	12.89	8.39	4.54×10^{-1}
BAXTER	12.35	1.42	1.33×10^{-2}	6.71	2.99	3.29×10^{-1}

The problems differ substantially in how much work is performed, and also in what amount of memory accesses is required for each unit of work, when comparing the original and blocked implementations. An interesting measure is how cache misses change from the original to the blocked implementation when considering the work performed. Table 4.8 shows the ratio of floating point operations to cache misses, i.e. how many multiplications that are, on average, computed without causing a cache miss. The numbers reveal that even though the number of operations are considerably reduced for all problems, the cost of each operation increases. For DFL001, for which vertical partitioning showed to reduce the number of multiplications by a factor of 4.7, the cache miss penalty is severe. The number of operations per cache miss has dropped considerably for each level of cache. FIT2D, on the other hand, which achieves a speedup greater than the factor of operations performed, has a much better cache utilization, increasing the number of operations per cache miss from 5.4 and 71.7 to 60.2 and 247.8 in L1 and L2 cache, respectively.

It is clear that the different test cases cause very different effects on the cache performance of the multiplication algorithms. More important than the size and sparsity of the matrices is the sparsity structure. The matrix PA of DFL001 (Figure 4.3g), which has its entries evenly distributed, causes higher cache miss rates compared to matrices where entries are collected in denser parts of the matrix, such as PA of BAS1LP (Figure 4.3e). L1 cache miss rates for DFL001 and BAS1LP are, on the Intel core, 7.65% and 31.47.%, respectively.

Table 4.8: Ratio of floating point operations to level 1, level 2 and level 3 cache misses for the original GLPK matrix multiplication and the vertically partitioned matrix multiplication on the Intel core

	Original			**1D Partition**		
Name	**L1**	**L2**	**L3**	**L1**	**L2**	**L3**
FIT2D	5.4	71.7	164132.0	60.2	347.8	2921.6
NSCT2	8.6	45.4	1865.6	10.4	52.8	1248.7
BAS1LP	9.5	111.7	6623.5	13.5	68.0	3911.9
DFL001	1.9	7.1	209.0	0.9	1.4	11.1
BAXTER	5.6	88.9	2517.3	4.9	10.3	61.1

4.5 Cholesky Decomposition

As described in Section 3.2, the blocked cholesky decomposition consists of three operations: The standard cholesky decomposition on blocks of the matrix, transposing of matrix blocks, and updating the remaining sub-matrix by matrix multiplication. Table 4.9 shows the time distribution for the blocked cholesky decomposition. For most of the problems the transposing of U takes less than one percent of the total execution time, while the block matrix multiplication accounts for around 90%. The exception is FIT2D which spends a relatively large proportion of time transposing U, and also less time in the sub-matrix update.

Table 4.9: Time distribution for blocked cholesky decomposition on the AMD core

Name	**Standard cholesky** [%]	**Transposing** [%]	**Sub-matrix update** [%]
FIT2D	14.6	22.4	63.0
NSCT2	5.5	0.5	94.0
BAS1LP	8.5	0.8	90.7
DFL001	8.3	0.7	91.0
BAXTER	9.9	1.0	89.2

Speedups for the blocked cholesky decomposition are presented in Table 4.10. Block widths of 100 and 50 are used on the AMD core and Intel core, respectively. Compared to the matrix multiplication, where speedup varies between 1.2 and 102.6, the effect of the blocking is not as varying for the different problems for the blocked cholesky decomposition. This could be linked to the fact that the structures of matrices S do not differ as much for the different problems as is the case for matrices PA, as can be seen from Figure 4.3.

Table 4.10: Speedup achieved with the blocked cholesky decomposition on the AMD and Intel core, and the ratio of floating point operations performed in the original GLPK implementation to the floating point operations performed in the blocked version

Name	Speedup AMD core	Speedup Intel core	FP ops. ratio
FIT2D	0.51	0.41	1.16
NSCT2	2.04	1.41	1.28
BAS1LP	1.61	1.45	1.32
DFL001	1.64	1.21	1.32
BAXTER	2.54	1.81	1.79

As in the case of blocked matrix multiplication, the blocked cholesky decomposition can avoid performing some zero-operations. Consider performing the outer product between the first row and column, $u_{ij} = u_{i1}u_{1j}$, on the matrix U in Figure 4.5. This outer product is subtracted from sub-matrix U_{22} as described in Section 3.1. A dot in Figure 4.5 represents a non-zero entry and an X represents a non-zero entry that leads to a zero-multiplication when computing the third row. In the original GLPK implementation, the first row is unpacked into a full array. When traversing the third row, an element corresponding to a zero in this array causes a multiplication by zero. This is not the case in the blocked version since the row (together with adjacent rows) is transposed into a separate matrix that is accessed by row-pointers in the matrix multiplication (Algorithm 1 in Section 2.5).

$$U = \left[\begin{array}{cccccc}
\bullet & & \bullet & & \bullet & \\
& \bullet & & & & \\ \hline
\bullet & & \bullet & \times & & \times \\ \hline
& & \bullet & \bullet & & \\
\bullet & & & & \bullet & \\
& & \bullet & & & \bullet
\end{array}\right]$$

Figure 4.5: Performing the outer product on the first row and column on matrix U. A dot represents a non-zero entry and an X represents a non-zero entry that leads to a zero-multiplication when computing the third row

As seen from Table 4.10, the reduction of operations for the blocked cholesky decomposition is not as substantial as for the blocked matrix multiplication. For most of the problems, the blocked cholesky decomposition achieves speedups higher than the factor of reduced operations, indicating better cache utilization, and that overhead associated with extra indexing of the blocks does not sabotage performance. Table 4.11 shows that the ratio of memory accesses to floating point operations is not as different between the original and blocked cholesky decomposition, as was the case for the matrix multiplication.

Table 4.11: Ratio of memory accesses to floating point operations for the original GLPK cholesky decomposition and the blocked cholesky decomposition

Name	**Original**	**Blocked**
FIT2D	3.05	5.46
NSCT2	2.03	2.11
BAS1LP	2.02	2.10
DFL001	2.02	2.16
BAXTER	2.02	2.14

Table 4.12 and Table 4.13 present the cache miss rates on the AMD core and the Intel core, respectively. Compared to the blocked matrix multiplication, the blocked cholesky decomposition does not achieve the same improvements in L1 cache. In fact, for all problems except NSCT2, L1 performance is degraded on the AMD core. There is, however, a consistent improvement in L3 cache on the Intel core. FIT2D is the exception for which cache miss rates are increased for all levels of cache.

Table 4.12: Level 1 and level 2 cache misses for the original GLPK cholesky decomposition and the blocked cholesky decomposition on the AMD core

	Original		**Blocked**	
Name	**L1** [%]	**L2** [%]	**L1** [%]	**L2** [%]
FIT2D	0.04	4.72	0.50	10.15
NSCT2	0.52	5.01	0.50	6.54
BAS1LP	0.35	5.18	0.70	7.36
DFL001	0.60	8.23	1.48	8.00
BAXTER	0.79	6.23	1.06	8.04

Taking the number of floating point operations performed into consideration, Table 4.14 shows the ratio of floating point operations to cache misses on the Intel core. The numbers confirm the improvement in L3 cache utilization. For NSCT2, the average number of floating point operations between each L3 cache miss is increased from 277.5 to 2839.1, a factor of 10.2.

Table 4.13: Level 1, level 2 and level 3 cache misses for the original GLPK cholesky decomposition and the blocked cholesky decomposition on the Intel core

	Original			**Blocked**		
Name	**L1 [%]**	**L2 [%]**	**L3 [%]**	**L1 [%]**	**L2 [%]**	**L3 [%]**
FIT2D	0.81	4.73	22.58	3.20	28.68	54.25
NSCT2	5.39	6.83	34.55	4.84	6.63	3.31
BAS1LP	4.74	6.54	25.55	4.64	9.47	2.22
DFL001	5.02	12.70	27.45	5.29	55.41	1.39
BAXTER	5.51	10.37	22.90	4.95	29.39	2.20

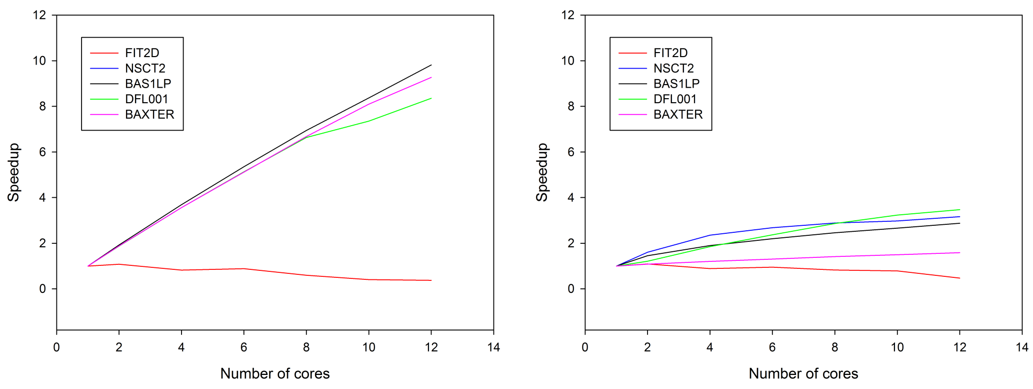
Table 4.14: Ratio of floating point operations to level 1, level 2 and level 3 cache misses for the original GLPK cholesky decomposition and the blocked cholesky decomposition on the Intel core

	Original			**Blocked**		
Name	**L1**	**L2**	**L3**	**L1**	**L2**	**L3**
FIT2D	46.1	1136.6	3103.3	4.1	13.7	16.2
NSCT2	8.9	120.3	277.5	9.8	109.9	2839.1
BAS1LP	10.2	153.0	351.6	10.5	164.6	4225.0
DFL001	9.6	76.8	275.7	7.6	16.4	765.6
BAXTER	8.7	85.2	260.5	8.9	29.8	920.7

4.6 Parallel Cholesky Decomposition

The sub-matrix update of the blocked cholesky decomposition has been parallelized in OpenMP. Attempts at parallelizing the remaining parts did not improve performance, and so were left to run in serial. This means that for a problem where 90% of the time is spent updating the sub-matrix, by Amdahls law, a maximum speedup of 10 can be achieved for the whole procedure.

Figure 4.6 shows the speedup for the parallelized sub-matrix update using the load balancing scheme described in Section 3.3, and using the guided scheduling option in OpenMP, on the AMD node. Guided scheduling dynamically assigns each thread with decreasing chunks of iterations and was the most effective among the scheduling options on the problem set. The load balancing clearly improves performance on multiple cores. Excluding FIT2D, speedup on 12 cores falls between 8 and 10 for all problems using the load balancing, compared to a speedup between 1.5 and 3.5 using the guided scheduling. The load balanced version provides consistently good results on all of these problems. FIT2D is slowed down in both cases. For this problem the interior point method spends time in the order of microseconds in cholesky decomposition, so that the overhead of parallelization sabotages the potential speedup.



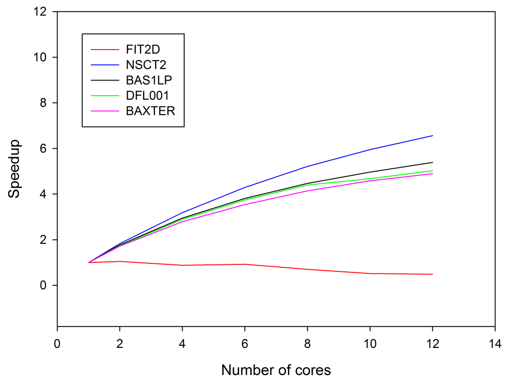
(a) Load balanced (b) Guided scheduling

Figure 4.6: Parallel Blocked cholesky sub-matrix update speedup on the AMD node

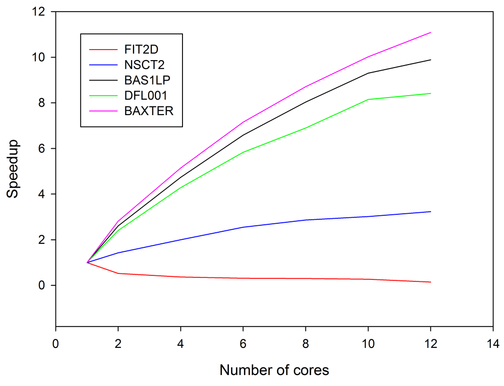
Figure 4.7 shows the speedup for the parallel blocked and the parallel original cholesky decomposition on the AMD node. As expected by Amdahls law, the speedup of the blocked version is decreasingly linear as the number of cores increases. The parallel original cholesky decomposition using the load balancing scheme achieves more varying speedups for the different problems than what was the case for the sub-matrix update in the blocked version (Figure 4.6a). For NSCT2, the blocked version shows a clear advantage over the original, achieving a speedup of 6.6 (10.2 for the block update), compared to a speedup of 3.2 for the original.

4.7 Block Sizes and Memory Considerations

Fixed block sizes have been used for all the results in this chapter. Since sparse matrices have varying sparsity patterns, one block size is not necessarily ideal for all problems. Hence, the choice of block size depends both on the computer architecture, to optimize for cache, and also on the problem to be solved. Eleyat found that by using a fixed block size, the speedup



(a) Blocked



(b) Original

Figure 4.7: Parallel Cholesky decomposition speedup on the AMD node

for NSCT2 was reduced by 16% compared to using block sizes adjusted for this problem [10]. However, finding the optimal block size required manual tuning for each problem.

Memory consumption has not been discussed in this work. Storing row pointers for each matrix partition requires extra memory over the plain CRS representation. Also, the blocked cholesky decomposition stores both U and its transpose, whereas only U is stored in the original implementation. For the problems considered, memory requirements have not been an issue. The most memory intensive problem, NSCT2, uses 141MB of memory for matrices A, S and U using double precision floating point numbers.

Chapter 5

Conclusions and Future Work

Matrix blocking can make sparse matrix multiplication and sparse cholesky decomposition more efficient. The effect differs significantly between problems, which can have matrices of very different sparsity structures. The degree of success for a given problem is determined by a complex combination of number of operations performed, the ratio of memory accesses to operations, and cache utilization. No single conclusion can be drawn for all problems. Although the motivation behind the blocked multiplication was to improve cache efficiency, analysis shows that avoiding zero-multiplications can be even more important for performance.

The blocked sparse cholesky decomposition, built on the same ideas and data structures as the blocked matrix multiplication, shows some differences from the multiplication in how the blocking affects performance. For the problems considered, the reduction of floating point operations performed was not as dramatic as was the case for the blocked matrix multiplication, and cache performance appears to be of more importance to achieve speedup. The precomputed division of work between processors is invaluable to achieve good load balance in the parallel cholesky decomposition. The blocked implementation better utilizes multiple processors for some problems.

As suggested for the matrix multiplication, future work can look at different storage formats and computation mechanisms for blocks based on their sparsity [10]. The Intel-based system used in this work is a temporary system for bridging operations to the new super-computer at NTNU. Future work can evaluate the performance of the algorithms on the new system. Also, it remains to fully parallelize the blocked cholesky decomposition.

References

[1] CentOS 5.4 PerfCTR and PAPI. `http://backedbyapenguin.wordpress.com/2010/03/05/centos-5-4-perfctr-and-papi`, June 2012.

[2] GNU Linear Programming Kit. `http://gnu.org/software/glpk/`, April 2012.

[3] Linear Programming Test Problems, BPMPD Home Page. `http://www.sztaki.hu/~meszaros/bpmpd/`, April 2012.

[4] The NETLIB LP Test Problem Set. `http://www.numerical.rl.ac.uk/cute/netlib.html`, April 2012.

[5] S. Browne, J. Dongorra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction To Algorithms, 2nd. ed.* MIT Press, 2001.

[7] T. A. Davis. *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2006.

[8] P. J. de Jonge. A comparative study of algorithms for reducing the fill-in during cholesky factorization. *Journal of Geodesy*, 66:296–305, 1992. 10.1007/BF02033190.

[9] J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):pp. 151–180, 1995.

[10] M. Eleyat, L. Natvig, and J. Amundsen. Cache-aware matrix multiplication on multicore systems for ipm-based lp solvers. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 431 –438, sept. 2011.

[11] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 2010.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979.

[13] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2(4):575–601, 1992.

[14] R. Shahnaz, A. Usman, and I.R. Chughtai. Review of storage techniques for sparse matrices. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1 –7, dec. 2005.