A Framework for Efficient Execution of Matrix Computations

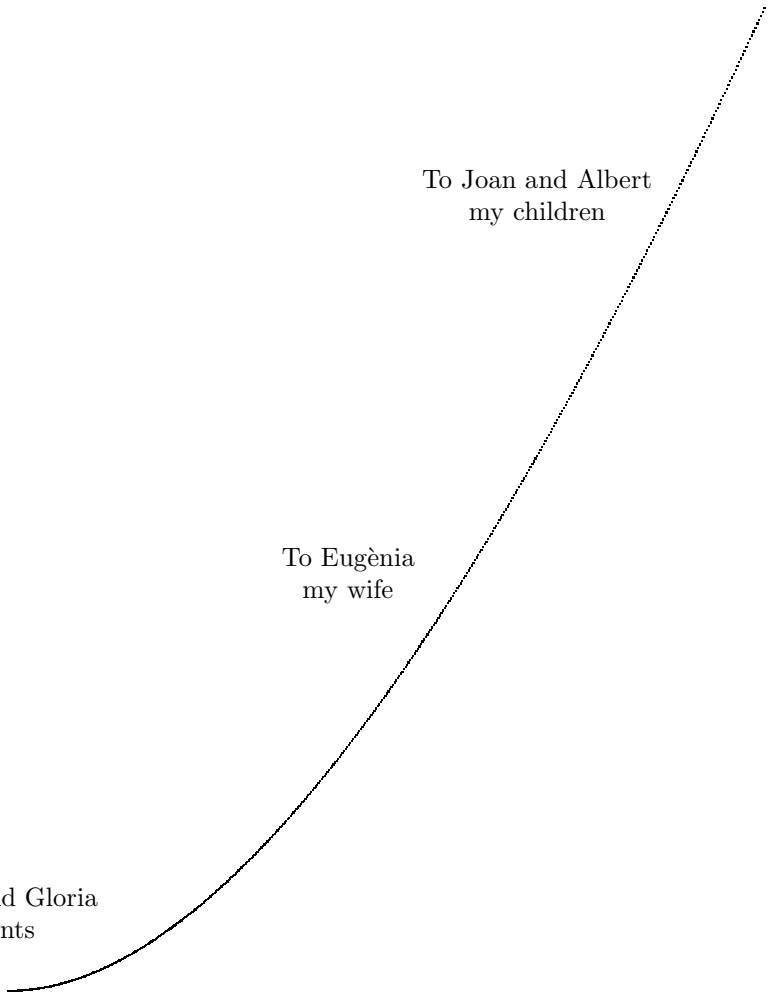
Doctoral Thesis

May 2006

José Ramón Herrero

Advisor: **Prof. Juan J. Navarro**

UNIVERSITAT POLITÈCNICA DE CATALUNYA
Departament D'Arquitectura de Computadors


To Joan and Albert
my children
To Eugènia
my wife
To Ramón and Gloria
my parents

Stillicidi casus lapidem cavat
Lucretius
(c. 99 B.C.-c. 55 B.C.)
De Rerum Natura[1]

[1] *Continual dropping wears away a stone.* Titus Lucretius Carus. On the nature of things

Abstract

Matrix computations lie at the heart of most scientific computational tasks. The solution of linear systems of equations is a very frequent operation in many fields in science, engineering, surveying, physics and others. Other matrix operations occur frequently in many other fields such as pattern recognition and classification, or multimedia applications. Therefore, it is important to perform matrix operations efficiently. The work in this thesis focuses on the efficient execution on commodity processors of matrix operations which arise frequently in different fields.

We study some important operations which appear in the solution of real world problems: some sparse and dense linear algebra codes and a classification algorithm. In particular, we focus our attention on the efficient execution of the following operations: sparse Cholesky factorization; dense matrix multiplication; dense Cholesky factorization; and Nearest Neighbor Classification.

A lot of research has been conducted on the efficient parallelization of numerical algorithms. However, the efficiency of a parallel algorithm depends ultimately on the performance obtained from the computations performed on each node. The work presented in this thesis focuses on the sequential execution on a single processor.

There exists a number of data structures for sparse computations which can be used in order to avoid the storage of and computation on zero elements. We work with a hierarchical data structure known as hypermatrix. A matrix is subdivided recursively an arbitrary number of times. Several pointer matrices are used to store the location of submatrices at each level. The last level consists of data submatrices which are dealt with as dense submatrices. When the block size of this dense submatrices is small, the number of zeros can be greatly reduced. However, the performance obtained from BLAS3 routines drops heavily. Consequently, there is a trade-off in the size of data submatrices used for a sparse Cholesky factorization with the hypermatrix scheme. Our goal is that of reducing the overhead introduced by the unnecessary operation on zeros when a hypermatrix data structure is used to produce a sparse Cholesky factorization. In this work we study several techniques for reducing such overhead in order to obtain high performance.

One of our goals is the creation of codes which work efficiently on different platforms when operating on dense matrices. To obtain high performance, the resources offered by the CPU must be properly utilized. At the same time, the memory hierarchy must be exploited to tolerate increasing memory latencies. To achieve the former, we produce inner kernels which use the CPU very efficiently. To achieve the latter, we investigate nonlinear data layouts. Such data formats can contribute to the effective use of the memory system.

The use of highly optimized inner kernels is of paramount importance for obtaining efficient numerical algorithms. Often, such kernels are created by hand. However, we want to create efficient inner kernels for a variety of processors using a general approach and avoiding hand-made codification in assembly language. In this work, we present an alternative way to produce automatically efficient kernels based on a set of simple codes written in a high level language, which can be parameterized at compilation time. The advantage of our method lies in the ability to generate very efficient inner kernels by means of a good compiler. Working on regular codes for small matrices most of the compilers we used in different platforms were creating very efficient inner kernels for matrix multiplication. Using the resulting kernels we have been able to produce high performance sparse and dense linear algebra codes on a variety of platforms.

In this work we also show that techniques used in linear algebra codes can be useful in other fields. We present the work we have done in the optimization of the Nearest Neighbor classification focusing on the speed of the classification process.

Tuning several codes for different problems and machines can become a heavy and unbearable task. For this reason we have developed an environment for development and automatic benchmarking of codes which is presented in this thesis.

As a practical result of this work, we have been able to create efficient codes for several matrix operations on a variety of platforms. Our codes are highly competitive with other state-of-art codes for some problems.

Acknowledgements

My family and me have undergone really tough moments these years. I feel joyful we have overcome them. I am also glad that such situations have allowed me to learn many things. One of the most important is to be thankful.

I want to express my deepest gratitude to Eugènia, my wife. She has taken care of a really large number of important things to allow me to work an uncountable number of hours; she cheered me up when I was feeling down; she has been besides me at all times.

This work, as it is, would have never been achieved without the support of my parents. They have supported and encouraged me all my life, providing me with the means to study what I wanted. I feel really thankful for the scale of values I have inherited from them. Thanks to Javier, my brother, for his company and faith in me. I also feel indebted to my family in-law. They have supported us when our children were sick or I was working very hard, i.e. always.

I want to thank Juanjo Navarro, my supervisor for granting me the opportunity to work in our department. He has given me good advice while he has let me do it my way. He has been very patient and has trusted me.

It has been important for me to receive the encouragement of Jenny Edwards and Cristina Barrado. They have shown interest in my work, offered their help and insisted that it was time to finish. My recognition also for Oriol Riu. Some of my jobs had large memory requirements and didn't fit into normal batch queues. He kindly scheduled my processes in special queues to run overnight or during the weekends.

I thank all my office-mates and all the other colleagues with which I have been teaching undergraduate courses for being so easy going. I would like to acknowledge the system administrators in LCAC and CEPBA for their good work keeping the systems up and running, and the department's administrative staff which manage to make bureaucracy less painful. Many thanks to some colleagues which offered support, complicity, resources and warm words: Agustín Fernández, David López, Josep L. Larriba, Jordi Torres, Xavier Martorell, Enric Morancho, Mateo Valero, J.C. Cruellas, Leandro Navarro and Miguel Lechón.

Thanks to V. Olivé, M. Antoni, X. Muro, R. Martínez, S. Stroke and C. Naranjo among others for contributing to my self-consciousness and to overcome some of my limitations. There are many other people towards which I feel gratitude. Most of them will never read this but I have them in my heart.

And finally, the *last, but not least* part. Some will agree that matrix computations are important. Few would discuss that health is of extreme importance. Thanks to J. Moreno, X. Krauel and their team at Hospital Sant Joan de Déu in Barcelona for helping our son Albert save his life. Thanks to C. Casajoana and G. Tissedre for keeping my family and me alive and kicking.

Eugènia, Joan and Albert: thanks for coming. Thanks for staying.

Good Use Right

It is strictly prohibited to use, to investigate or to develop, in a direct or indirect way, any of the scientific contributions of the author contained in this work by any army or armed group in the world, for military purposes and for any other use which is against human rights or the environment, unless a written consent of all the persons in the world is obtained.

Funding

This work was supported by the Ministerio de Educación y Ciencia of Spain (TIN2004-07739-C02-01).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the goals and topics covered in this doctoral dissertation relating them to the corresponding sections in the document.

1.1 Motivation

Matrix computations lie at the heart of most scientific computational tasks. The solution of linear systems of equations is a very frequent operation in many fields in science, engineering, surveying, physics and others. Other matrix operations occur frequently in many other fields such as pattern recognition and classification, or multimedia applications. Therefore, it is important to perform matrix operations efficiently. The work in this thesis focuses on the efficient execution of some frequent matrix operations on commodity processors.

1.2 Goals

The main goal of this thesis is that of obtaining efficient implementations of codes which operate on matrices. We are interested in operations which arise frequently in different fields. We will study some important operations which appear in the solution of real world problems: some sparse and dense linear algebra codes and a classification algorithm. In particular, we will focus our attention on the efficient execution of the following operations: sparse Cholesky factorization; dense matrix multiplication; dense Cholesky factorization; and Nearest Neighbor Classification.

A lot of research has been conducted on the efficient parallelization of numerical algorithms. However, the efficiency of a parallel algorithm depends ultimately on the performance obtained from the computations performed on each node. The work presented in this thesis will focus on the sequential execution on a single processor.

There exists a number of data structures for sparse computations which can be used in order to avoid the storage of and computation on zero elements. We work with a hierarchical data structure known as hypermatrix. A matrix is subdivided recursively an arbitrary number of times. Several pointer matrices are used to store the location of submatrices at each level. The last level consists of data submatrices which are dealt with as dense submatrices. Our goal is that

of reducing the overhead introduced by the unnecessary operation on zeros when a hypermatrix data structure is used to produce a sparse Cholesky factorization.

One of our goals is the creation of codes which work efficiently on different platforms when operating on dense matrices. To obtain high performance, the resources offered by the CPU must be properly utilized. At the same time, the memory hierarchy must be exploited to tolerate increasing memory latencies. To achieve the former, we want to produce inner kernels which use the CPU very efficiently. We would like to create them for a variety of processors. However, we want to achieve this using a general approach, avoiding hand-made codification in assembly language. To achieve the latter, we want to investigate nonlinear data layouts. Such data formats can contribute to the effective use of the memory system.

We want to show that techniques used in linear algebra codes can be useful in other fields. To do this, we will optimize another important algorithm: the Nearest Neighbor classification. We will focus on the speed of the classification process.

In summary, we would like to identify the key points for obtaining high performance from matrix computations and be able to create efficient codes for several matrix operations on a variety of platforms.

1.3 Overview

Next, we will briefly introduce the topics covered in this work. For each of them we include the corresponding ACM categories and subject descriptors.

Compiler-optimized inner kernels

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra
G.4 [**Mathematical Software**]: Efficiency

General Terms Algorithms, performance

Keywords Inner kernel, dense matrix multiplication.

The efficiency of the inner kernel is of paramount importance for the overall performance of an algorithm. For this reason the inner kernel is usually coded by hand using assembly language. However, this is a difficult and time consuming task when this work has to be done for many platforms. We are interested in developing high performance inner kernels using a high level language. We address the issue of generating high performance inner kernels in the following way: we produce specialized routines which allow for the efficient execution of basic linear algebra codes on small matrices (matrices which fit in the first cache level). We can automatically tune these routines for different matrix sizes for a given platform. We introduce these routines in a library which we call the Small Matrix Library (SML). Then, we can use such routines as inner kernels for operations on sparse and dense matrices. This topic is explained in chapter 2.

Sparse Cholesky Factorization

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems, sparse, structured, and very large systems (direct methods)*
G.2.2 [**Discrete mathematics**]: Graph Theory—*graph labeling*
G.4 [**Mathematical Software**]: Efficiency

General Terms Algorithms, performance

Keywords Sparse Cholesky, hypermatrix, sparse matrix ordering, elimination tree, amalgamation, matrix multiplication.

One of the most frequently occurring problems in all areas of scientific endeavor is that of solving a system $Ax = b$ of n linear equations in n unknowns. Depending on the characteristics of the problem different methods can be used to solve them. It is a basic tenet of numerical analysis that structure should be exploited whenever solving a problem. Algorithms for general matrix problems can be streamlined in the presence of such properties as symmetry, definiteness and sparsity [66]. A very important operation is the Sparse Cholesky factorization of a symmetric positive definite matrix. This operation is used frequently in applications of Finite Element Methods as well as in Interior Point Methods of linear programming, taking a substantial proportion of their total computing time.

In chapter 3 we will provide some details on the work we have done in order to produce a rather efficient implementation of a sparse Cholesky factorization using a hypermatrix [61], data structure (defined in section 3.2). We have developed several overhead reduction techniques trying to reduce the operations on zeros which can be stored within data submatrices. Section 3.4.1 shows that an important contribution to the performance improvement obtained by our sparse hypermatrix Cholesky comes from the usage of routines from our SML. The use of these routines, specialized in the operation on matrices of a given and small size, allows us to reduce data block size while keeping BLAS3 performance. In fact, section 3.4.2 shows that the best results are obtained when rectangular data submatrices are used. We have tried to reduce the number of non productive operations performed on parts of submatrices which are full of zeros. Sections 3.4.3 and 3.4.4 present the results obtained by using *bit vectors* and *windows* of non-zeros within data submatrices. Both techniques improve the performance of our sparse hypermatrix Cholesky factorization. However, the former becomes unnecessary when the latter is used. We have also investigated two forms of *amalgamation* for use in a sparse hypermatrix Cholesky: in both cases we allow for the possibility of storing zeros and performing computations on them. First, using *Intra-block amalgamation* we extend windows within data submatrices under certain circumstances. We present this technique in section 3.7.1. Second, we have developed algorithms which implement a *hypermatrix oriented supernode amalgamation*. Only some matrices benefit from this technique in a sequential code. However, the resulting hypermatrix is better partitioned for parallel factorization. These algorithms are presented in section 3.7.2. In section 3.5 we show that a 2D recursive data layout and scheduling of the computations is beneficial to the efficient computation of the

Cholesky factorization of large sparse matrices. Final considerations on sparse hypermatrix Cholesky factorization are presented in section 3.8.

We have compared the performance of our sparse hypermatrix Cholesky factorization with that of codes using traditional approaches such as supernodal and multifrontal codes. A package which includes implementations of those two methods is TAUCS. The implementation in TAUCS is known to provide reasonable performance [68]. We have found that our code outperforms TAUCS for many problems arising in Interior Point Methods, providing similar performance on matrices taken from the application of the Finite Element Method.

Operation on dense matrices

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra
G.4 [**Mathematical Software**]: Efficiency
E.1 [**Data Structures**]: Arrays

General Terms Algorithms, performance

Keywords Dense Cholesky, dense matrix multiplication, hypermatrix, square block data layout.

One of our goals is the creation of codes which work efficiently on different platforms when operating on dense matrices. For this reason we must exploit the machine resources as effectively as we can. Our efforts will go in two directions: first, try to use the resources within the CPU efficiently; second, exploit the memory hierarchy adequately. To achieve the former we use the routines in our SML as inner kernels. To carry out the latter we have investigated the usage of two nonlinear array layouts: a hypermatrix scheme and a square block data layout.

As it can be seen in chapter 4, the use of hypermatrices with SML routines as inner kernels is quite competitive on some platforms. Recently, however, we have used a different data structure: a Square Block data layout (*SB*) which can be found in section 4.4. Basically, this is a two dimensional layout of square data submatrices. Using this data structure we have reduced some overhead incurred by the recursive codes which operate on the hypermatrix data structure. The resulting Mflops rate increased to outperform that of ATLAS [167] double precision general matrix multiplication (DGEMM) routine on some platforms, and got very close to their performance on other machines in which ATLAS relies on hand-made inner kernels coded in assembler.

Application to other fields: Nearest Neighbor Classification

Categories and Subject Descriptors

I.5.4 [**Pattern Recognition**]: Applications
I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms Algorithms, measurement, performance

Keywords Nearest Neighbor classification

The Nearest Neighbor (NN) classification procedure is a popular technique in pattern recognition, speech recognition, multitarget tracking, medical diagnosis tools, etc. A major concern in its implementation is the immense computational load required in practical problem environments. Other important issues are the amount of storage required and the data access time. In chapter 5 we address these issues by using techniques widely used in linear algebra codes: use floating-point operations instead of integer arithmetic, apply tiling, loop unrolling or software pipelining. We show that a simple code can be very efficient on commodity processors and can sometimes outperform complex codes which can be more difficult to implement efficiently. This is something which this application has in common with the sparse codes: sometimes it can pay off to do more operations, as long as they are performed faster.

POSTDATE: Performance Oriented SofTware Development And Tuning Environment

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*
D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*
D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*
G.4 [**Mathematical Software**]: Efficiency

General Terms Algorithms, measurement, performance

Tuning several basic linear algebra codes for many matrix sizes and machines became a heavy and unbearable task. For this reason we developed an environment, presented in chapter 6, which consists of three components.

- *maker*: A front-end to *make* to ease the build process.
  As the amount of software developed grew and more platforms were used, a need for a simplification of the build process appeared. We created an environment for the development of programs which can be found in section 6.1.

- *ACME*: A framework to ensure accurate measurements.
  Getting accurate measurements for all our benchmarks is of paramount importance. For this reason we developed a set of routines which ease this process. Details are provided in section 6.2.

- *BMT*: A benchmarking tool.
  A framework to handle the tuning process automatically. Such framework, is introduced in section 6.3.

Figure 1.1 shows the relations among some parts of the work done during the development of this thesis and connects them to the corresponding chapters in the dissertation. SML is a key part for the efficient execution of both dense and sparse codes. We used *maker*, ACME, and BMT to develop most of our codes and benchmark them.

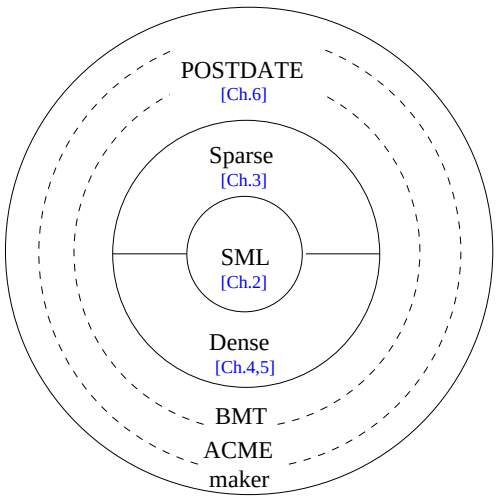


Figure 1.1: Overview of the main parts of this work.

Chapter 2

Compiler-optimized inner kernels

The use of highly optimized inner kernels is of paramount importance for obtaining efficient numerical algorithms. Often, such kernels are created by hand. In this chapter, however, we present an alternative way to produce efficient kernels based on a set of simple codes which can be parameterized at compilation time. Using the resulting kernels we have been able to produce high performance sparse and dense linear algebra codes on a variety of platforms.

2.1 Introduction

Creation of efficient code has traditionally been done manually using assembly language and based on a great knowledge of the target architecture. Such an approach, however cannot be easily undertaken for many target architectures and algorithms. Alternatively, codes specially optimized for a particular target computer can be written in a high level language [105, 134]. This approach avoids the use of the assembly language but keeps the difficulty of manually tuning the code. It still requires a deep knowledge of the target architecture and produces a code that, although portable, will rarely be efficient on a different platform.

Many linear algebra codes can be implemented in terms of matrix multiplication [9, 104]. Thus, it is important to have efficient matrix multiplication routines at hand. The Fortran implementation of Basic Linear Algebra Subroutines (BLAS) [47] is inefficient, and developing efficient codes for a variety of platforms can take a great effort. Consequently, there have been attempts to produce such codes automatically. A new paradigm was created: Automated Empirical Optimization of Software (AEOS). The goal is to use empirical timings to adapt a package automatically to a new computer architecture. PHiPAC [27] was the first such project. Later, the Automatically Tuned Linear Algebra Software (ATLAS) project [167] appeared, which continues to date. Today, ATLAS-tuned libraries represent one of the most widely used libraries. Automatic code generation and empirical search has also been applied in other fields. The specific problem of sparse matrix vector multiplication is addressed in the SPARSITY project [99]. In the signal processing domain, FFTW [59] au-

tomatically tunes an implementation of the discrete Fourier transform (DFT). Code generation in search for portable optimal performance for linear digital signal processing is studied in the SPIRAL project [149].

ATLAS uses many well known optimization techniques developed by both linear algebra and compiler optimization experts. However, and despite its name, a great effort has been applied to produce high performance inner kernels for matrix multiplication using hand-coded routines contributed by some experts. Directory `tune/blas/gemm/CASES` within the ATLAS distribution contains about 90 files which are, in most cases, written in assembler, or use some instructions written in assembler to do data prefetching. Often, one or more of these codes outperform the automatically generated codes. The best code is automatically selected as the inner kernel. The use of such hand-made inner kernels has improved significantly the overall performance of ATLAS subroutines on some platforms. Many processors have specific kernels built for them. However, there exist processors for which no such hand-made codes are available. Then, the performance obtained by ATLAS on the latter platforms is comparatively worse than that obtained on the former.

BLAS routines are usually optimized to deal with matrices of different sizes. One amongst several inner codes can be selected at runtime depending on matrix dimensions. This is very convenient for medium and large matrices of different sizes. When small matrices are provided as input, however, the overhead incurred becomes too large and the performance obtained is poor. There are applications which produce a large number of matrix operations on small matrices. For instance, programs which deal with sparse problems or multimedia codes. In those cases, the use of BLAS can be ineffective to provide high performance.

We are interested in obtaining efficient codes when working on both small and large matrices on a variety of platforms. For these purposes we have created a framework which allows us to produce ad hoc routines which can perform matrix multiplications quite efficiently operating on small matrices. Though our codes are adapted to the underlying architecture, they are written in a high level programming language (Fortran). On each platform, our codes were compiled with the native compilers: Compaq Fortran 77, MIPSpro F77, IBM xlf, and Intel Fortran compilers respectively. Using this approach we have created efficient routines on a variety of platforms: MIPS R10000, Digital Alpha 21164 and 21264, IBM Power4, Intel Pentium4 and Itanium2. Based on these routines, we have produced efficient implementations of both sparse and dense codes for several platforms which are presented in chapters 3 and 4. In the following sections we present our approach and comment on the results. All the results presented in this document refer to double precision floating point data.

2.2 Generation of efficient inner kernels

Our approach relies on the quality of code produced by current compilers. The resulting code is usually less efficient than that written manually by an expert. However, its performance can still be extremely good and sometimes it can yield even better code.

2.2.1 Taking advantage of compiler optimizations

Compiler technology is a mature field. Many optimization techniques have been developed over the years. A very complete survey of compiler optimization techniques can be found in [17]. The knowledge of the target platform introduced in the compiler by its creators together with the use of well known optimization techniques such as software pipelining, loop unrolling, auto-vectorization, etc., can result in efficient codes which can exploit the processor's resources in an effective way. This is specially true when it is applied to highly regular codes such as a matrix multiplication kernel. Since many platforms have outstanding optimizing compilers available nowadays, we want to let the compiler do the creation of optimized object code for our inner kernels.

2.2.2 Smoothing the way to the compiler

The optimizations performed by the compiler can be favored by certain characteristics of the compiled code. For instance, some loop orders can be more beneficial than others. Some access patterns can be more effective in using memory. Knowing the number of iterations of a loop can help the compiler decide on the application of some techniques such as loop unrolling or software pipelining. We have taken this approach for creating a Small Matrix Library (SML). Basically, we:

- Provide the compiler with as much information as possible regarding matrix leading dimensions and loop trip counts;
- Try several variants of code, with different loop orders or unroll factors.

In addition, in some cases the resulting code can be more efficient if:

- Matrices are aligned;
- All matrices are accessed with stride one;
- Store operations are removed from the inner kernel.

2.3 Creation of a Small Matrix Library (SML)

We have created a library called Small Matrix Library (SML) which contains routines meant to work on small matrices of fixed size. For each desired operation, we have written a set of codes in Fortran. Although the most important kernel is the matrix multiplication operation, we have also included in the library two other operations: SYRK (Symmetric Rank K update) and TRSM (Solve Triangular System of equations) since these operations appear within a Cholesky factorization. Nevertheless, we will concentrate on the matrix multiplication since, in general, this is the operation which takes most of the computation time.

For each operation we have written several variants of code with different loop orders (*kji, ijk*, etc.) and unroll factors. We compile each of them using the native compiler and trying several optimization options. For each resulting executable, we automatically execute it and register its performance. These results are kept in a database and finally employed to produce a library using

the best combination of parameters. This process is done automatically with a benchmarking tool introduced in section 6.3. This work was presented in [87]. We also tried feedback driven compilation using the Alpha native compiler but performance either remained the same or even decreased slightly.

By fixing the leading dimensions of matrices and the loop trip counts at compilation time we have managed to obtain very efficient codes for matrix multiplication on small matrices. Since several parameters are fixed at compilation time the resulting object code is useful only for matrix operations conforming to these fixed values. Actual parameters of these routines are limited to the initial addresses of the matrices involved in the operation performed. Thus, there is one routine for each matrix size. Each one has its own name in the library. In this document, however, we refer to any of them as *mxmts_fix*.

One approach related to ours uses C++ templates which are used to create kernels for basic linear algebra operations of fixed size [159]. The authors create a Fixed Algorithm Size Template (FAST) library with sizes fixed at compilation time. This library is then used to create a Basic Linear Algebra Instruction Set (BLAIS) library.

Our approach is also related to Iterative compilation [109] which consists in a repetitive compilation of code using different parameters. Program transformations such as loop tiling and loop unrolling are very effective techniques to exploit locality and expose instruction level parallelism. The authors claim that finding the optimal combination of tile size and unroll factor is difficult and machine dependent. Thus, they propose an optimization approach based on the creation of several versions of a program and decide upon the best by actually executing them and measuring their execution time. Our approach for obtaining high performance code is similar with the difference that, while they apply a set of transformations, we use simple codes and let the compiler do its best. We must note that the use of the highest optimization flag for the compiler was not necessarily the one producing the best performance. Thus, for each variation we tried several compiler optimization flags, keeping the resulting performance in a database. Eventually, we pick the best code variation and compiler flag to create the routine inserted in the library.

We must also note that the best loop order and unroll factor obtained for some matrix dimensions on one processor is not necessarily the best for other matrix dimensions or platforms. Choosing a single code for all cases would result in an important performance loss.

2.3.1 A Poly-Algorithmic Approach

A single algorithm is not always efficient for solving a target problem on any platform [118, 71, 165, 48, 132]. Thus, we follow a poly-algorithmic[1] approach for selecting the most suitable code among various ones for a given problem size and platform. For instance, consider the matrix multiplication commented above. Table 2.1a presents eleven implementations of $C = C - A \times B^T$ matrix multiplication operation where $A(i \times k)$, $B(j \times k)$ and $C(i \times j)$. For simplicity, each algorithm is coded with a number. The notation for the order of the loops (forms) was introduced in [135]. kji means that loop k (direction k in

[1]The term *polyalgorithm* was introduced by Professor John Rice and refers to the choice of one suitable algorithm from a set of candidate algorithms, all designed to solve the same problem, with the aim of obtaining the best possible performance in a given situation [165].

the iteration space) is specified as the outer loop while loop i is found in the inner loop. $4\tilde{i}$ means that a loop with 4 iterations in direction i has been fully unrolled. *BCreg* means values in matrices B and C are kept in local variables in an attempt to improve register reuse. $i(\ldots)$ means tiling is done in the i dimension.

Alg.	form
1	jik_Creg
2	jik
3	kji_Breg
4	$i(jk4\tilde{i}\_BCreg)$
5	$i(jk4\tilde{i}\_Breg)$
6	$i(jk4\tilde{i})$
7	ijk
8	kji
9	jki
10	$jik4\tilde{k}\_Creg$
11	$jik8\tilde{k}\_Creg$

a)

Matrix sizes	*Alpha 21164*	*R10000*
4_4_4	2	8
4_4_8	3	5
4_4_16	3	3
4_4_32	3	3
8_8_8	10	6
8_8_16	11	8
8_8_32	11	5
16_16_16	11	2
16_16_32	11	5
32_32_32	11	6

b)

Table 2.1: Some implementations of the $C = C - A \times B^T$ operation: **a)** List of algorithms **b)** Algorithm giving best performance for each matrix dimensions on an Alpha 21164 and an R10000.

Table 2.1b shows the best algorithm found for the matrix multiplication operation for several matrix sizes on the Alpha 21164 and R10000 processors. Matrix sizes are expressed as i_j_k. There is a large variation in the optimal algorithm chosen. Seven out of eleven algorithms resulted to be the best for some particular combination of sizes and target platform. Choosing a single algorithm for all cases would result in an important performance loss. Subsequently, we have developed up to 42 variations of the matrix multiplication kernel which have been used in the experiments presented in the sequel of this document.

2.4 Inner matrix multiplication kernel for operations on sparse matrices

Exploiting structure is important when working on sparse matrices. There exist data structures which avoid any storage of zeros and computation on them. However, operation on single data elements is very inefficient. Thus, such elements are grouped in order to work more efficiently using BLAS3 operations, at the expense of possibly storing and working on zero values. Thus, even when we are operating on sparse matrices it is interesting to use dense operations which work on submatrices.

As we will see in chapter 3 we split sparse matrices into small blocks. We can avoid storage and operation on submatrices which would be full of zeros. For submatrices which contain nonzero values we perform operations on dense submatrices of small size. To do this, we will use the appropriate routines from our SML in the inner kernel of the codes which deal with sparse matrices of larger sizes.

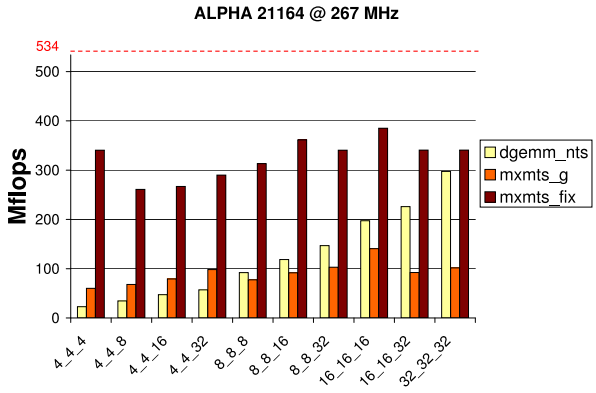


Figure 2.1: Performance of different $A \times B^T$ routines for several matrix sizes on an Alpha 21164.

Figure 2.1 shows the performance of different routines for dense matrix multiplication for several matrix sizes on an Alpha-21164.[2] The matrix multiplication performed in all routines benchmarked uses: the first matrix without transposition (n); the second matrix transposed (t); and subtracts the result from the destination matrix (s). This is the reason why we call the vendor BLAS routine *dgemm_nts*. This operation appears within a Cholesky factorization of a lower triangular matrix L using Fortran column-wise storage. Actually, it is the operation which takes most of the computation time for this operation.

The vendor BLAS routine *dgemm_nts* yields very poor performance for very small matrices getting better results as matrix dimensions grow towards a size that fills the L1 cache (8 Kbytes for the Alpha-21164). This is due to the overhead of passing a large number of parameters, checking for their correctness, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when the operation is performed on large matrices. However, it is notable when small matrices are multiplied. Also, since its code is prepared to deal with large matrices, further overhead can appear in the inner code by the use of techniques such as strip mining or data copying. We will show in section 3.4.1 that this is not adequate for our hypermatrix Cholesky factorization.

A simple matrix multiplication routine *mxmts_g* which avoids any parameter checking and scaling of matrices can outperform the BLAS for very small matrix sizes. Finally, our matrix multiplication code *mxmts_fix* with leading dimensions and loop limits fixed at compilation time gets excellent performance for all block sizes ranging from 4x4 to 32x32. The latter is the maximum value that allows for a good use of the L1 cache on the Alpha unless tiling techniques are used.

Figure 2.2 shows the performance of routines *dgemm_nts* and *mxmts_fix* for several matrix sizes on a 250 MHz MIPS R10000 processor. The first level instruction and data caches have size 32 Kbytes. There is a secondary unified instruction/data cache with size 4 Mbytes. This processor's theoretical peak

[2]Labels in this figure and the next refer to the three dimensions of the iteration space. However, for all the other figures matrices are assumed to be square and of equal size. In all plots the dashed line at the top shows the theoretical peak performance of the processor.

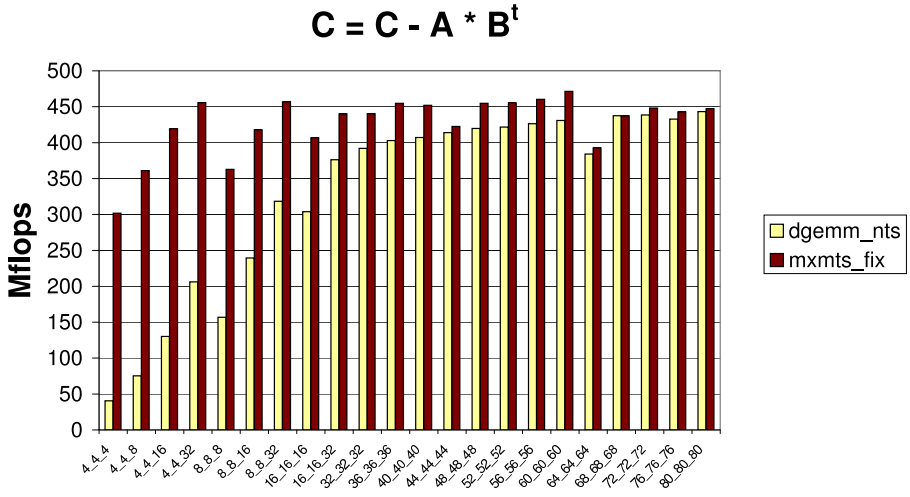


Figure 2.2: Performance of different $A \times B^T$ routines for several matrix sizes on an R10000.

performance is 500 Mflops. Results on the R10000 processor are similar to those of the Alpha with the only difference that the *mxmts_g* performs very well[3]. This is due to the ability of the MIPSpro F77 compiler to produce software pipelined code, while the Alpha compiler hardly ever manages to do so. We conclude that, as long as a good compiler is available, fixing leading dimensions and loop limits is enough to produce high performance codes for very small dense matrix kernels.

2.5 Inner matrix multiplication kernel for operations on dense matrices

We have extended our Small Matrix Library (SML) with routines which work with sizes larger than the ones used for the sparse codes. We choose as inner kernel the one providing best performance.

Figures 2.3 and 2.4 show the peak performance of the $C = C - A \times B^T$ matrix multiplication routines in our SML for several matrix dimensions on two different processors. On an Alpha 21264A we have chosen the routine which works on matrices of size 48×48. On the Intel Itanium2 we have selected size 92×92 while on the R10000 the elected size is 60×60 (see figure 2.2). The following tables show information about the caches present in these three platforms. Table 2.2 shows the number of caches and their sizes. Table 2.3 shows the minimum latency for a floating-point load when it hits in each cache level on several machines.

The Itanium2 has three levels of cache. In the first level it has separate instruction and data caches with 16 Kbytes each. Then, it also has a 256 Kbytes L2 cache and an off-chip L3 cache with possible sizes ranging from 1.5

[3]The corresponding bar has been excluded from the figure to avoid cluttering.

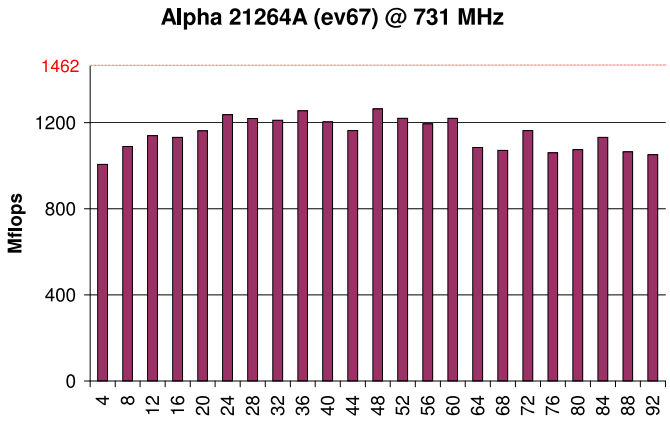


Figure 2.3: Peak performance of SML dense matrix multiplication routines: Alpha 21264A.

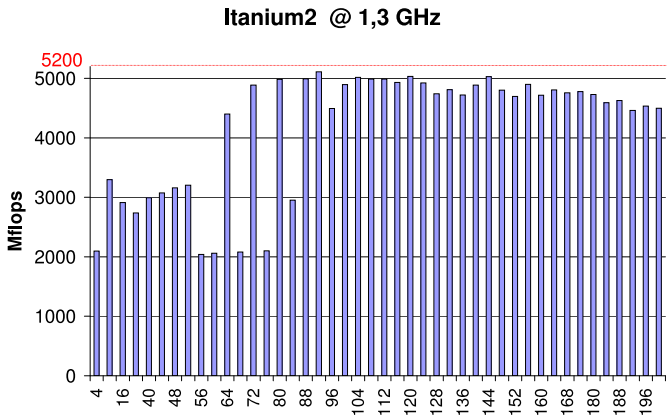


Figure 2.4: Peak performance of SML dense matrix multiplication routines: Intel Itanium2.

up to 9 MB. The configuration used had a 3 MB L3 cache. It is interesting to note that on the Itanium2 the highest performance was obtained for matrix sizes which exceed the capacity of the level 1 (L1) data cache. The reason for this comes from the fact that this machine does not cache floating-point data in L1 cache [100]. In addition, it has low latency when a load hits in its level 2 (L2) cache. Table 2.3 shows the minimum latency for floating-point loads in each cache level for the R10000, Alpha 21264A and the Itanium2 processors. The latency of a floating-point load which hits in the Itanium2 level i cache is similar to that of the Alpha level $i-1$ cache. The Intel Fortran compiler applied the software pipelining technique automatically for tolerating such latency and produced efficient codes. This are the reasons why on this machine the best peak performance for our SML matrix multiplication routines was found for matrices which exceed the L1 data cache size.

On MIPS, ALPHA, and Intel Itanium2 platforms we could obtain very efficient inner kernels for the matrix multiplication. However, on Power4 and Intel

Table 2.2: Cache sizes

Cache Level	R10000	ALPHA 21264A	Itanium2
L1	32 KB	64 KB	16 KB
L2	4 MB	4 MB	256 KB
L3	-	-	3 MB

Table 2.3: Floating-point load latency (minimum) when load hits in cache.

Cache Level	R10000	ALPHA 21264A	Itanium2
L1	3	4	-
L2	8-10	13	**6**
L3	-	-	13

Pentium 4 Xeon the Mflops obtained are far from the theoretical peak of the machine. Figure 2.5 shows the performance obtained on Power4 and Pentium 4 Xeon processors. The best results obtained with our compiler-optimized codes on the Power4 are over 2500 Mflops for matrices of size 36. The sustained performance of the vendor DGEMM gets around 2700 Mflops. Therefore, the code produced for our inner kernel is reasonably good. However, on the Intel Xeon the performance obtained with our approach (3334 Mflops) is not only far from the theoretical peak but also from the sustained performance of the DGEMM routines of GotoBLAS (4000 Mflops). On the other hand it approaches that of ATLAS (3500 Mflops). The Intel Fortran Compiler (version 8.0) is unable to exploit efficiently the SSE2 [160] extension which provides SIMD parallelism on this processor for the $C = C - A \times B^T$ matrix multiplication. In section 2.7 we will see that the situation changes when data is properly aligned and accessed with stride one.

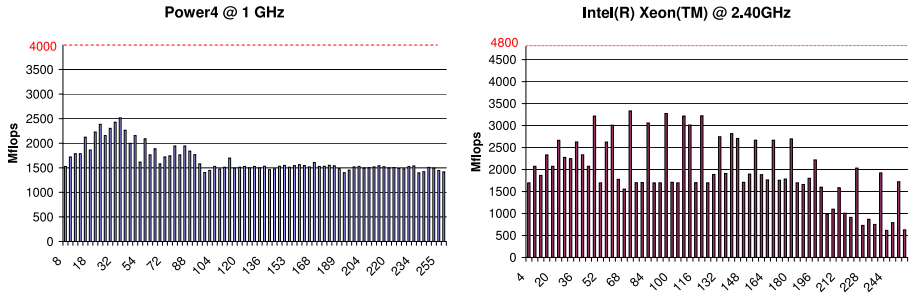


Figure 2.5: Peak performance of SML dense matrix multiplication routines: Power4 and Pentium 4.

2.6 Generalization of the matrix multiplication

We have generalized our matrix multiplication codes to be able to perform the matrix operations $C = \beta C + \alpha op(A) \times op(B)$ where α and β are scalars and op(A) is A or A^T. Actually, we consider $\beta = 1$ since it is more efficient to perform the

multiplication of matrix C by β before calling the matrix multiplication kernel rather than performing this multiplication within it [43]. We allow values of 1 and -1 for α.

We parameterize our kernels with preprocessor symbols which are adequately defined at compilation time to determine the type of operation performed. Thus, given a particular loop order and unroll factor, we can produce up to eight kernels: those corresponding to the combinations of transposition of matrices and values of α.

The case $C = C + \alpha A^T \times B$ is particularly appealing since it allows accessing all three matrices with stride one. In addition, stores to matrix C can be hoisted from the inner loop, resulting in additional performance gains. Actually, this is the kernel used in ATLAS.

2.7 Alignment

Contrary to what happens on other platforms tested, the results obtained for our matrix multiplication routines on an Intel Pentium 4 Xeon were initially very poor. This machine allows for vectorization with the SSE2 instruction set [160] and the Intel Fortran compiler can take advantage of them. However, we were getting Mflops values far from the theoretical peak for this machine. The problem is that data must be properly aligned and accessed with stride one. Otherwise the compiler is not able to produce such an efficient code. By forcing the alignment of matrices to 16, the Intel Fortran Compiler is able to vectorize the code, resulting in a substantial performance increase. The best case is that with $A^T \times B$ which gets a peak performance of 3810 Mflops. Table 2.4 summarizes these results. On this platform, as on the Itanium2, floating-point loads are not cached in the L1 cache. Thus, the block size automatically chosen (104) targets the L2 cache. This work was presented in [96]. Other works which state the importance of alignment for certain processors can be found in [2, 32].

Table 2.4: Peak Mflops of inner kernel on a Pentium 4 Xeon Northwood.

	$A \times B^T$	$A^T \times B$
No align	3334	3220
Align	3457	3810

2.8 Conclusions

It is possible to obtain high performance inner kernels using a high level language and a good optimizing compiler. The inner kernel can target the first level cache. It is interesting to note that it can also target the second level cache: on some machines the highest performance was obtained for matrix sizes which exceed the capacity of the L1 data cache. This happens on processors which do not cache floating-point data in the first level cache, as on the Intel Pentium 4 and Itanium2 processors. We must note, however, that on these machines the latency of a floating-point load from the second level cache is quite low. Thus,

in some cases it can be hidden using techniques for tolerating latency such as software pipelining and prefetching.

The case $C = C + \alpha A^T \times B$ is particularly appealing since it allows accessing all three matrices with stride one. In addition, stores to matrix C can be hoisted from the inner loop, resulting in additional performance gains.

There is a large variation in the optimal algorithm chosen for different matrix sizes and platforms. Choosing a single algorithm for all cases would result in an important performance loss. A poly-algorithmic approach is necessary to obtain high performance inner kernels.

We agree with other authors [165] which identify the need for a standard kernel interface, different from BLAS, so that optimized kernels tuned to specific processors can be made available to library developers for writing high performance codes. These kernel routines operate on small blocks that fit in the L1 cache.

Using routines in our Small Matrix Library can improve the performance of codes which perform a large number of operations on small matrices. Examples of such programs are sparse matrix and multimedia codes (such as 3D graphics engines, signal processing or video encoding/decoding algorithms). These routines can also be used as inner kernels for general dense operations. Based on these routines, we have produced efficient implementations of both sparse and dense codes for several platforms which are presented in chapters 3 and 4.

Chapter 3

Sparse Hypermatrix Cholesky Factorization

In this chapter we present our work on the sparse Cholesky factorization using a hypermatrix data structure. First, we provide some background on the sparse Cholesky factorization and explain the hypermatrix data structure. Next, we present the matrix test suite used. Afterwards, we present the techniques we have developed in pursuit of performance improvements for the sparse hypermatrix Cholesky factorization of a symmetric positive definite matrix into a lower triangular factor L.

3.1 Background

Sparse Cholesky factorization is heavily used in several application domains, including finite-element and linear programming algorithms. It forms a substantial proportion of the overall computation time incurred by those applications. Consequently, there has been great interest in improving its performance [50, 137, 154]. Methods have moved from column-oriented approaches into panel or block-oriented approaches. The former use level 1 BLAS while the latter have level 3 BLAS as computational kernels [154]. Operations are thus performed on blocks (submatrices).

Columns having similar structure are taken as a group. These column groups are called *supernodes* [122]. Some supernodes may be too large to fit in cache and it is advisable to split them into *panels* [137, 154]. In other cases, supernodes can be too small to yield good performance. This is the case of supernodes with just a few columns. Level 1 BLAS routines are used in this case and the performance obtained is therefore poor. This problem can be reduced by *amalgamating* several supernodes into a single larger one [11]. Although, some null elements are then both stored and used for computation, the resulting use of level 3 BLAS routines often leads to some performance improvement.

We address the optimization of the sparse Cholesky factorization of large matrices. For this purpose, we use a *Hypermatrix* [61] block data structure.

3.2 Hypermatrix data structure

Our application uses a data structure based on a hypermatrix (HM) scheme [61, 138], in which a matrix is partitioned recursively into blocks of different sizes. Commercial package such as NASTRAN or PERMAS have used the hypermatrix structure for solving very large systems of equations [13]. They can solve very large systems out-of-core and can work in parallel. This approach is also related to a variety of recursive/nonlinear data layouts which have been explored elsewhere for both regular [33, 58, 165, 170] and irregular [129] applications.

The HM structure consists of *N* levels of submatrices, where N is an arbitrary number. In order to have a simple HM data structure which is easy to traverse we have chosen to have blocks at each level which are multiples of the lower levels. The top *N-1* levels hold pointer matrices which point to the next lower level submatrices. Only the last (bottom) level holds data matrices. Data matrices are stored as dense matrices and operated on as such. Hypermatrices can be seen as a generalization of quadtrees. The latter partition each matrix precisely into four submatrices [169].

Null pointers in pointer matrices indicate that the corresponding submatrix does not have any non-zero elements and is therefore unnecessary. This is useful when matrices are sparse. Figure 3.1 shows a sparse matrix and a simple example of corresponding hypermatrix with 2 levels of pointers.

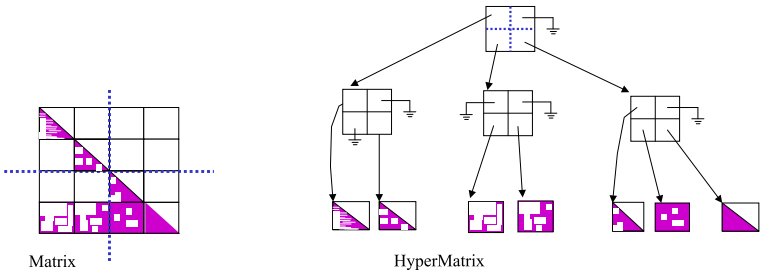


Figure 3.1: A sparse matrix and a corresponding hypermatrix.

The main potential advantage of a HM structure is the ease of use of multilevel blocks to adapt the computation to the underlying memory hierarchy. The operation on dense submatrices can take advantage of the BLAS3 routines. At the same time however, HM suffers from the disadvantage that zeros can be stored within data submatrices and used in computation. This is due to the fact that we do not descend down to the element level. Instead we use data submatrices of arbitrary size, considering them as dense blocks. Consequently, such data submatrices can store some zero elements. This can produce substantial overhead on sparse Cholesky. In all figures shown in this chapter we will present the *Effective Mflops* obtained. They refer to the number of useful floating point operations (*#flops*) performed per second. Although the time includes the operations performed on zeros, this metrics excludes nonproductive operations on zeros performed by the HM Cholesky algorithm when data submatrices contain such zeros.

$$Effective\ Mflops = \frac{\#flops(excluding\ operations\ on\ zeros) \cdot 10^{-6}}{Time\ (including\ operations\ on\ zeros)}$$

Block sizes can be chosen either statically (fixed) or dynamically. In the former case, the matrix partition does not take into account the structure of the sparse matrix. In the latter case, information from the *elimination tree* [121] is used. Initially we will partition the hypermatrix statically. Afterwards, we will use the elimination tree to create a dynamic (variable) partitioning. In the latter case we use an amalgamation algorithm specific for use with hypermatrices.

3.3 Matrix characteristics

We have used several test matrices. Most results presented in this document were obtained using sparse matrices corresponding to linear programming problems. QAP matrices come from Netlib [136] while others come from a variety of linear multicommodity network flow generators: A Patient Distribution System (PDS) [29], with instances taken from [57]; RMFGEN [19, 65]; GRIDGEN [117]; TRIPARTITE [64]. Table 3.1 shows the characteristics of several matrices obtained from such linear programming problems. Matrices were ordered with METIS [107] and renumbered by an elimination tree postorder.

Table 3.1: Matrix characteristics: application of Interior Point Methods.

Matrix	Dimension	NZs	NZs in L[a]	Density	Flops to factor[b]
GRIDGEN1	330430	3162757	130586943	0.002	278891
QAP8	912	14864	193228	0.463	63
QAP12	3192	77784	2091706	0.410	2228
QAP15	6330	192405	8755465	0.436	20454
RMFGEN1	28077	151557	6469394	0.016	6323
TRIPART1	4238	80846	1147857	0.127	511
TRIPART2	19781	400229	5917820	0.030	2926
TRIPART3	38881	973881	17806642	0.023	14058
TRIPART4	56869	2407504	76805463	0.047	187168
pds1	1561	12165	37339	0.030	1
pds10	18612	148038	3384640	0.019	2519
pds20	38726	319041	10739539	0.014	13128
pds30	57193	463732	18216426	0.011	26262
pds40	76771	629851	27672127	0.009	43807
pds50	95936	791087	36321636	0.007	61180
pds60	115312	956906	46377926	0.006	81447
pds70	133326	1100254	54795729	0.006	100023
pds80	149558	1216223	64148298	0.005	125002
pds90	164944	1320298	70140993	0.005	138765

[a] Number of non-zeros in factor L (matrix ordered using METIS).
[b] Number of floating point operations (in Millions) necessary to obtain L from the original matrix (ordered with METIS).

Recently, we have also benchmarked our code using matrices which come from applications of the Finite Element Method. Their characteristics can be found in table 3.2. Matrices bear, rail and methan come from structural engineering problems used in the PERMAS project [13]; cfd1 and cfd2 are pressure

matrices from problems in computational fluid dynamics from Ed Rothberg, Silicon Graphics, Inc.; nasasrb is a structural engineering problems from NASA. These last three matrices can be obtained from [42]. Matrix inline_1 is a stiffness matrix from MSC-Software used in the PARASOL project [144].

Table 3.2: Matrix characteristics: applications of Finite Element Analysis

Matrix	Dimension	NZs	NZs in L[a]	Density	Flops to factor[b]
bear	25906	412447	3278225	0.009	847
rail	11783	799545	3768886	0.054	1594
methan	48162	1234332	16631801	0.014	10493
nasasrb	54870	1366097	10489476	0.007	3496
cfd1	70656	949510	20910296	0.008	13523
cfd2	123440	1605669	37696869	0.005	31218
inline_1	503712	18660027	174608135	0.001	150974

[a]Number of non-zeros in factor L (matrix ordered using METIS).
[b]Number of floating point operations (in Millions) necessary to obtain L from the original matrix (ordered with METIS).

3.4 Reducing overhead

In this section we present several aspects of the work we have done to improve the performance of our sparse Hypermatrix Cholesky factorization. Both of them are based on the fact that a matrix is divided into submatrices and operations are thus performed on blocks (submatrices).

A matrix M is divided into submatrices. We call M_{br_i,bc_j} the data submatrix in block-row br_i and block-column bc_j. Figure 3.2 shows several submatrices within a matrix. The highest cost within the Cholesky factorization process comes from the multiplication of data submatrices. It takes approximately 90% of the total factorization time. In order to ease the explanation we will refer to the three submatrices involved in a product as A, B and C. For block-rows br_1 and br_2 (with $br_1 < br_2$) and block-column bc_j, each of these blocks is $A \equiv M_{br_2,bc_j}$, $B \equiv M_{br_1,bc_j}$ and $C \equiv M_{br_2,br_1}$. Thus, the operation performed is $C = C - A \times B^T$, which means that submatrices A and B are used to produce an update on submatrix C.

As we already introduced in section 3.2 the use of dense data submatrices allows for the use of level 3 Basic Linear Algebra Subroutines (BLAS3) which is the traditional way to obtain performance in a portable way. However, a certain number of zero elements can be stored in data submatrices and used during the factorization operation. This reduces the effective number of calculations performed since operations on such zeros are unnecessary. When the block size used is small, the number of zeros can be greatly reduced. However, the performance obtained from BLAS3 routines drops heavily. Consequently, there is a trade-off in the size of data submatrices used for a sparse Cholesky factorization with the hypermatrix scheme.

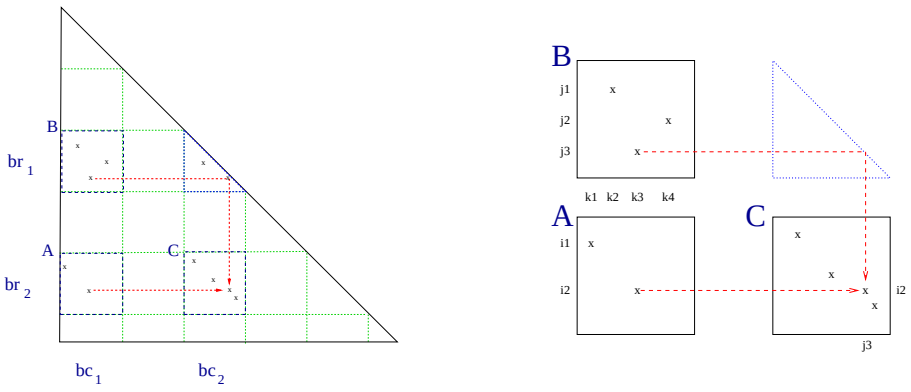


Figure 3.2: Static partition of a matrix: definition of blocks and example of use.

3.4.1 Using Small Matrix Library (SML) routines

In section 2.3 we introduced our library called Small Matrix Library (SML) which works on small matrices of fixed size. By fixing leading dimensions and loop trip counts at compilation time we can produce very efficient codes if a good compiler is available. Although the most important kernel is the matrix multiplication operation, we have also included in the library the SYRK (Symmetric Rank K update) and TRSM (Solve Triangular System of equations) since these operations appear within a Cholesky factorization.

The matrix multiplication performed in all routines benchmarked in this section uses the first matrix without transposition, the second matrix transposed, and subtracts the result from the destination matrix. This is the reason why we call the BLAS routine *dgemm_nts*. This operation appears within a Cholesky factorization of a matrix into a lower triangular matrix L.

As we mentioned in section 2.4 the vendor BLAS routine *dgemm_nts* yields very poor performance for very small matrices getting better results as matrix dimensions grow towards a size that fills the L1 cache. This is due to the overhead of passing a large number of parameters, checking for their correctness, and scaling the matrices (*alpha* and *beta* parameters in *dgemm*). This overhead is negligible when the operation is performed on large matrices but too large for operation on small matrices. Also, since the code is optimized for large matrices, further overhead can appear in the inner code by the use of techniques such as strip mining or data copying. Next, we show that this is not adequate for our hypermatrix Cholesky factorization.

Figure 3.3 shows results of the HM Cholesky factorization on an R10000[1] for matrix QAP15 from the Netlib set of sparse matrices corresponding to linear programming problems [136]; and problem pds40 from a Patient Distribution System (40 days) [57]. Ten submatrix sizes are shown: 4x4, 4x8, 4x16, ... 32x32. *Effective Mflops* are presented. They refer to the number of useful floating point operations performed per second.

When *dgemm_nts* is used, the best performance is usually obtained with data submatrices of size 16×16 or 16×32. Since the amount of zeros used can be large, the effective performance is quite low. Using *mxmts_fix* however,

[1]Results on the Alpha are similar.

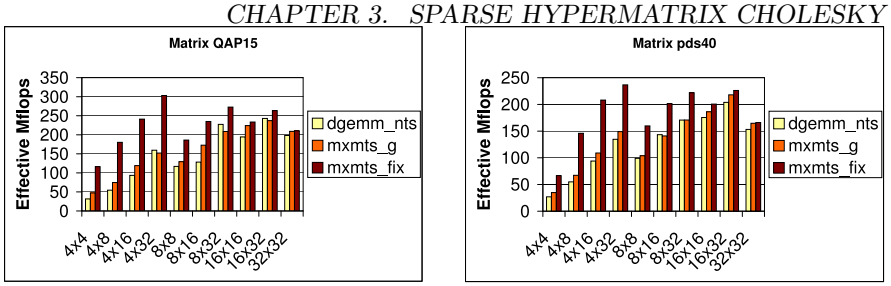


Figure 3.3: Impact of the matrix multiplication routine on sparse hypermatrix Cholesky. Mflops obtained using different $A \times B^T$ codes in the factorization of matrices QAP15 and pds40 on an R10000.

smaller submatrix sizes usually produce better results than larger submatrix sizes. Particularly effective in this application is the use of rectangular matrices due to the fill-in produced by the Cholesky factorization. For instance, using 4×16 or 4×32 submatrix sizes, the routine used yields very good performance. Since the number of operations on zeros is considerably lower, the effective Mflops obtained are much higher than those of any other combination of size and routine.

The use of a fixed dimension routines in the SML library speeded up our Cholesky factorization an average of 12% for our matrix test suite.

Information about the creation of the SML was given in [87]. The application of SML to sparse hypermatrix Cholesky was presented in [89].

3.4.2 Rectangular data submatrices

As seen from the results shown in the preceding section, the use of rectangular data matrices can favor performance. For instance, on the R10000, we store sparse matrices with subblocks of size 4×32 and use routine *mxmts_4_4_4_4_4_32*. The characteristics of a sparse Cholesky factorization explain this. Let us assume that the lower triangular matrix (L) is stored and used. Often an off-diagonal nonzero produces updates on other positions in the same row to its right. It is well known that fill-in can be introduced: positions which were originally zero become different from zero after the factorization. Since this updates are produced in a row-wise fashion, the rectangular shape is better suited for storing data submatrices.

3.4.3 Bit Vectors

We want to be able to avoid unnecessary matrix multiplications between matrices with elements in disjoint columns. What we need to know is whether a column within a data submatrix has any non-zero elements or not. We associate a set of bits to each data submatrix. We refer to such a set of bits as *bit vector*. Figure 3.4.3 shows a data submatrix with a bit vector associated to it.

Each bit in the vector is used to point to the existence of any non-zero in the corresponding column. For instance, consider matrix B in figure 3.5a. Let us consider column indices start at 1 (Fortran indexing). There are non-zero elements only in columns $k_2 = 3$, $k_3 = 4$ and $k_4 = 7$. Thus, only bits 3, 4 and 7 in BV_B will be different from 0. A bit-wise AND between bit vectors corresponding to matrices A and B can be used to decide whether the matrix

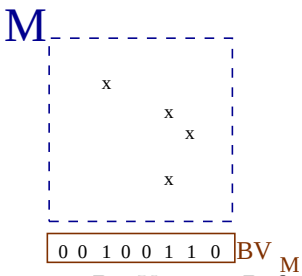


Figure 3.4: Bit Vectors: Definition

multiplication between those matrices is necessary or not. If a single bit of the bit-wise AND results to be 1 then we need to perform the operation. If all bits are zero, then we can skip it. This test can be done in a couple of CPU cycles with an AND operation followed by a comparison to zero. The creation of the bit vectors can be done initially, when the hypermatrix structure is prepared using the symbolic factorization information. The overhead for their creation is negligible.

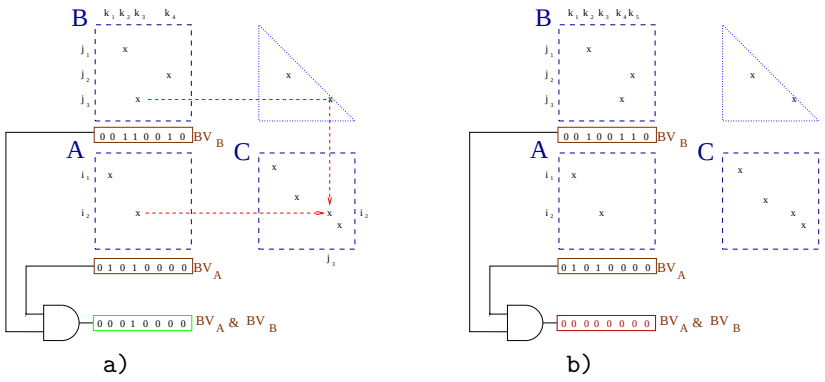


Figure 3.5: Using Bit Vectors. **a)** $BV_A \& BV_B \neq 0$: operation must be performed. **b)** $BV_A \& BV_B = 0$: operation can be avoided.

In [91] we showed that bit vectors can be effective to improve performance as long as windows are not used. Results can be found in section 3.6.

3.4.4 Windows within data submatrices

In order to reduce the storage and computation of zero values, we define *windows* of non-zeros within data submatrices in a way similar to that described in [55]: by keeping information about the actual space within a data submatrix which stores non-zeros (dense window) Figure 3.6a shows a window of non-zero elements within a larger block. The window of non-zero elements is defined by its top-left and bottom right corners. All zeros outside those limits are not used in the computations. Null elements within the window are still stored and computed. Storage of columns to the left of the window's leftmost column is avoided since all their elements are null. Similarly, we do not store columns to the right of the window's rightmost column. However, we do store zeros over

the window's upper row and/or underneath its bottom row whenever these window's boundaries are different from the data submatrix boundaries, i.e. whole data submatrix columns are stored from the leftmost to the rightmost columns in a window. We do this to have the same leading dimension for all data submatrices used in the hypermatrix. Thus, we can use our specialized SML routines which work on matrices with fixed leading dimensions. Actually, we extended our SML library with routines which have the leading dimensions of matrices fixed, while the loop limits may be given as parameters. Some of the routines have all loop limits fixed, while others have only one, or two of them fixed. Other routines have all the loop limits given as parameters. The appropriate routine is chosen at execution time depending on the windows involved in the operation. Thus, although zeros can be stored above or underneath a window, they are not used in computation. Zeros can still exist within the window but, in general, the overhead is greatly reduced.

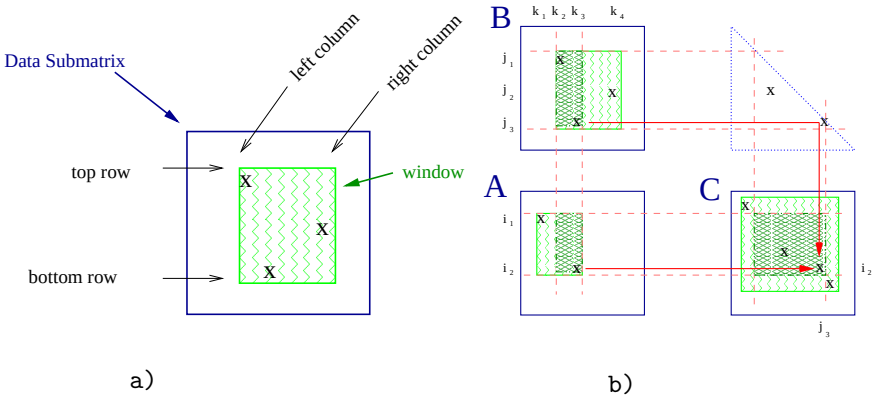


Figure 3.6: Windows within dense submatrices. **a)** A data submatrix and a window within it. **b)** Windows can reduce the number of operations.

The use of windows of non-zero elements within blocks allows for a larger default block size. When blocks are quite full operations performed on them can be rather efficient. However, in those cases where only a few non-zero elements are present in a block, or the intersection of windows results in a small area, only a subset of the total block is computed (dark areas within figure 3.6b).

When the column-wise intersection of windows in matrices A and B is null, we can avoid the multiplication of these two matrices (figure 3.7a). There are cases where the window definition we have used is not enough to avoid unnecessary operations. Consider figure 3.7b: there is a column-wise intersection of windows in A and B. Thus, we would perform a product using the dark area within the three matrices involved.[2] Results will be presented in section 3.6.

3.5 1D vs 2D data layouts and computations

In [90], we have compared 1 and 2 dimensional recursive layouts of data and computations on data blocks. The 2D recursive layout systematically produced

[2]However, if we look at the elements within those matrices we can see that the product $A \times B^T$ will produce a null update on C. In this case, the usage of bit vectors would be useful and could avoid this operation.

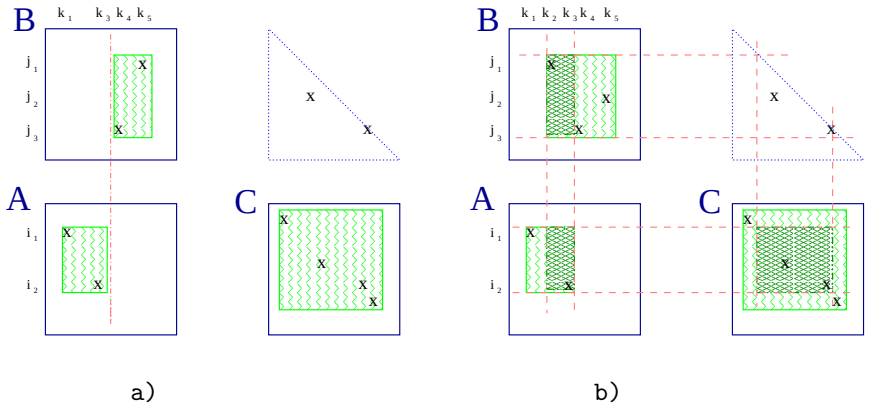


Figure 3.7: Windows: column-wise intersection. **a)** Disjoint windows can avoid matrix products. **b)** Windows can be ineffective to detect false intersections.

faster factorizations for the matrix test suite. Section 3.6.3 shows the results obtained with several variants of our HM Cholesky code. In all cases the code follows a right looking scheduling with a fixed partitioning of the hypermatrix (the elimination tree is consequently not used at all). We use a 4×32 data matrix size as stated in section 3.4.2.

3.6 First results and analysis

In this section we present and analyze the results we have obtained on our matrix test suite. Matrices come from two different areas: Interior Point Methods (IPM) of linear programming and Finite Element Analysis (FEA). We show results for five different codes. The processing done prior to the factorization is the same in all of them. The original sparsity pattern is used. Data values, however, are generated to obtain a positive definite matrix suitable for Cholesky factorization. Then, the sparse matrix is reordered using METIS [107] and renumbered by an elimination tree postorder [121].

In the sparse HM Cholesky code, after the elimination tree postorder we perform a symbolic factorization. Afterwards, we build the HM data structure. Finally, the resulting hypermatrix is factored. In this section we use the best variant of our sparse hypermatrix Cholesky. Later, in section 3.6.3, we will analyze each variant of our code and show which is the one which usually produces best performance.

Execution took place on a 250 Mhz MIPS R10000 Processor. The first level instruction and data caches have size 32 Kbytes. There is a secondary unified instruction/data cache with size 4 Mbytes. This processor's theoretical peak performance is 500 Mflops.

3.6.1 Problems solved using Interior Point Methods

We have used several sparse matrices corresponding to linear programming problems. Table 3.1 shows the characteristics of these matrices obtained from such linear programming problems.

Supernodal Cholesky (Ng-Peyton)

The left half of table 3.3 presents results obtained by a supernodal (SN) block Cholesky factorization [137]. It takes as input parameters the cache size and unroll factor desired. This algorithm performs a 1D partitioning of the matrix. A supernode can be split into *panels* so that each panel fits in cache. This code has been widely used in several packages such as LIPSOL [174], PCx [39], IPM [31] or SparseM [111]. Although the test matrices we have used are in most cases very sparse, the number of elements per column is in some cases large enough so that a few columns fill the first level cache. Thus, a one-dimensional partition of the input matrix produces poor results. As the problem size gets larger, performance degrades heavily. We noticed that we could improve its results by specifying cache sizes larger than the actual first level cache. However, performance degrades in all cases for large matrices. Due to this trend, some parameter combinations were not tried and correspond to blanks in the table.

HM ± windows ± BVs

The right half of table 3.3 shows results obtained by several variants of our sparse hypermatrix Cholesky code. We have used SML [89] routines to improve our sparse matrix application based on hypermatrices. A fixed partitioning of the matrix has been done to be able to test the impact of each overhead reduction technique used. We present results obtained with and without bit vectors for two data submatrix sizes: 8×8 and 4×32. For the latter we also introduce the usage of windows.

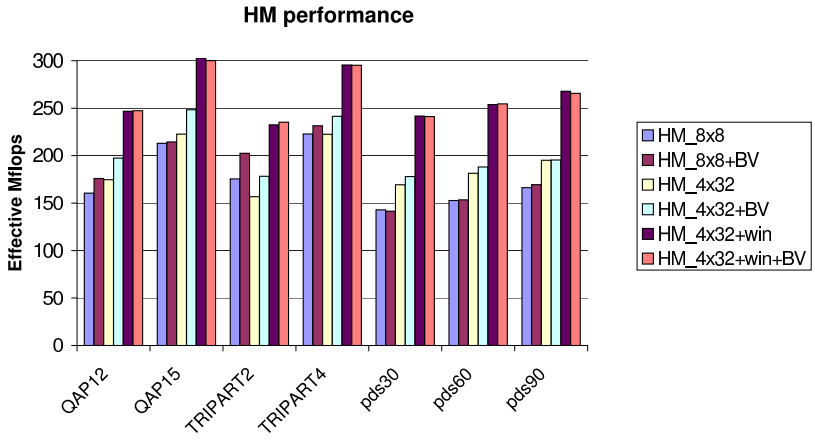


Figure 3.8: Performance of hypermatrix Cholesky with bit vectors (BV) and windows (win).

Figure 3.8 summarizes these results. The usage of windows clearly improves the performance of our sparse hypermatrix Cholesky algorithm. We observe that the usage of bit vectors can improve performance slightly when windows are not used. When windows are used, however, bit vectors are not effective at all.

Table 3.3: Supernodal vs Hypermatrix Cholesky: Effective Mflops

	Supernodal Cholesky (Ng-Peyton)								Hypermatrix Cholesky					
Upper levels									32 x 512					
Block size									8 x 8		4 x 32			
Windows									No		No		Yes	
Bit Vectors									No	Yes	No	Yes	No	Yes
Cache	32K		512K		1M		2M							
Unrolling	4	8	4	8	4	8	4	8						
GRIDGEN1						23.8		24.4	—	—	—	—	201.2	199.5
QAP8	186.6	194.2	186.7	194.9		175.1		177.3	139.0	142.5	146.6	151.5	179.6	180.2
QAP12	118.8	102.2	181.0	223.0		215.7		166.3	160.5	176.0	174.7	197.5	246.8	247.3
QAP15	49.0	54.8	152.4	186.0		165.6		149.1	213.1	214.4	222.7	248.4	303.1	300.2
RMFGEN1	55.9	61.9	169.8	189.0		256.2		154.9	221.0	220.8	202.8	210.7	298.4	300.9
TRIPART1	118.7	176.4	175.6	177.1		160.5		164.5	151.2	170.7	151.0	152.8	203.6	207.1
TRIPART2	205.2	182.5	208.4	213.1		216.9		171.8	175.5	202.5	156.8	178.3	232.5	235.3
TRIPART3		116.9		142.7		188.2		151.3	199.0	213.1	181.7	185.3	256.6	261.1
TRIPART4		46.4		119.3	121.1	133.8		118.7	222.8	231.5	222.5	241.4	295.5	295.2
pds1	87.5	89.6		75.7		73.8			19.0	20.2	13.7	14.3	20.2	20.2
pds10	121.5	125.5		183.5		132.2			102.4	106.5	111.6	121.3	193.3	192.3
pds20	106.8	82.7		130.3		104.2			126.1	127.1	139.3	149.9	229.7	227.6
pds30	104.0	78.2		133.5		135.5			142.9	141.5	169.3	178.0	241.7	241.1
pds40	99.9	97.3		126.4		159.8			144.5	144.6	169.8	176.8	247.9	242.1
pds50	84.8	92.0		121.1		121.4			140.2	147.5	181.3	191.4	252.4	252.2
pds60	85.0	66.0		131.8		112.0			152.8	153.4	181.4	188.1	253.9	254.5
pds70		91.4		127.4		111.3			154.6	154.8	186.0	194.4	253.0	252.4
pds80		62.3		132.5		107.2			154.1	164.4	196.6	198.0	260.1	259.5
pds90				108.5		105.1			166.3	169.4	195.2	195.4	267.9	265.7

Supernodal Cholesky (Ng-Peyton) vs HM

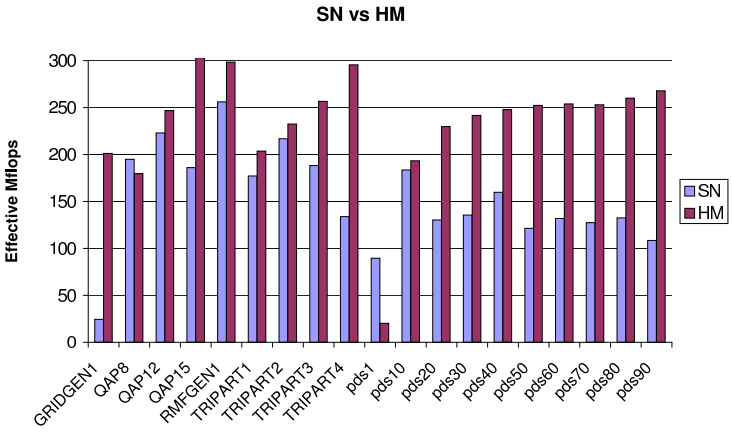


Figure 3.9: SN vs HM performance.

Figure 3.9 compares the best result obtained with each algorithm for the whole set of test matrices. We have included matrix pds1 to show that for small matrices the hypermatrix approach is usually very inefficient. This is due to the large overhead introduced by blocks which have plenty of zeros. For large matrices however, blocks are quite dense and the overhead is much lower. Performance of HM Cholesky is then much better than that of the supernodal algorithm. This is due to the better usage of the memory hierarchy: locality is properly exploited with the two dimensional partitioning of the matrix which is done in a recursive way using the HM structure.

Finally, figure 3.10 shows performance of each algorithm on several matrix families. Note that, contrary to the supernodal algorithm behavior, the hypermatrix Cholesky factorization improves its performance as the problem size gets larger.

We conclude that a two dimensional partitioning of the matrix is necessary for large sparse matrices. The overhead introduced by storing zeros within dense data blocks can be reduced by keeping information about a dense subset (window) within each data submatrix. Although some overhead still remains, the performance of our sparse hypermatrix Cholesky is up to an order of magnitude better than that of a supernodal block Cholesky which tries to use the cache memory properly by splitting supernodes into panels. Using windows and SML routines our HM Cholesky often gets over half of the processor's peak performance for medium and large size matrices factored in-core.

Comparison with other packages

Figure 3.11 shows the results obtained by five different sparse Cholesky factorization codes on the set of matrices introduced above. Matrix families are separated by dashed lines. The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [137] already discussed above.

The second bar shows the performance obtained by the sequential version of a 2D block-oriented approach [155] as found in the SPLASH-2 [173] suite.

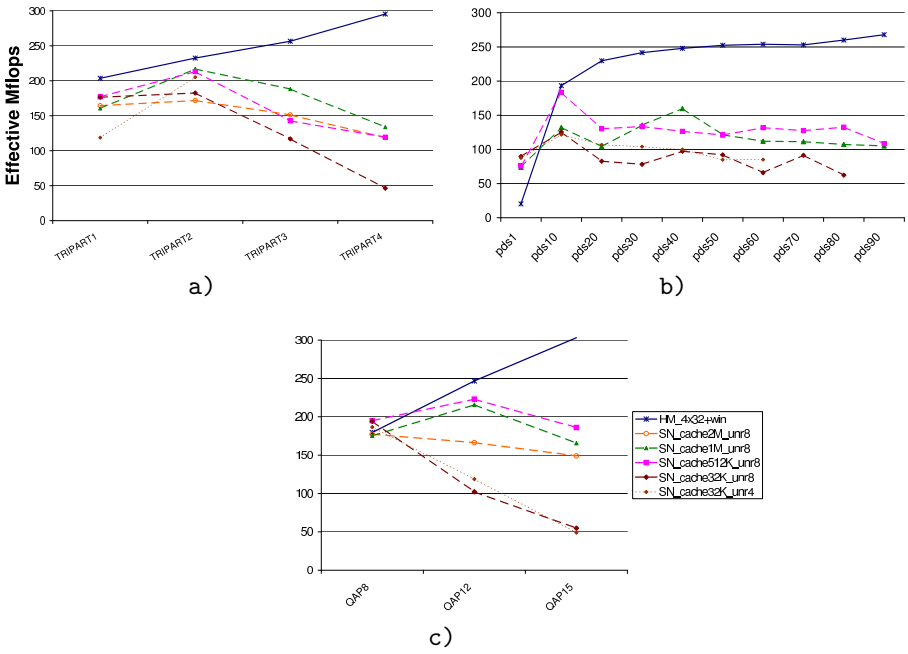


Figure 3.10: SN vs HM Cholesky for 3 matrix families: **a)** Tripart; **b)** PDS; **c)** QAP.

According to [155] submatrices are kept in a two-dimensional data layout. However, this code fails to produce efficient factorizations for large matrices.

The third and fourth bars correspond to sequential versions of the supernodal left-looking (SN-LL) and supernodal multifrontal (SN-MF) codes in the TAUCS package (version 2.2) [102]. In these codes the matrix is represented as a set of supernodes. The dense blocks within the supernodes are stored in a recursive data layout matching the dense block operations. The performance obtained by these codes is quite uniform.

Finally, the fifth bar shows the performance obtained by our right looking sparse hypermatrix Cholesky code (HM). We have used windows within data submatrices and SML [89] routines to improve our sparse matrix application based on hypermatrices. A three-level fixed partitioning of the matrix has been used. We present results obtained for data submatrix sizes 4×32 and upper hypermatrix levels with sizes 32×32 and 512×512. Basically, the sparse hypermatrix Cholesky factorization improves its efficiency as the problem dimension gets larger.

On this set of matrices, the sparse hypermatrix Cholesky code is clearly the best amongst all five codes. Its performance is considerably better than the others for large matrices.

3.6.2 Problems solved using Finite Element Analysis

Many codes in mechanical engineering and computational fluid dynamics use direct solvers. In this section we present results obtained on some matrices

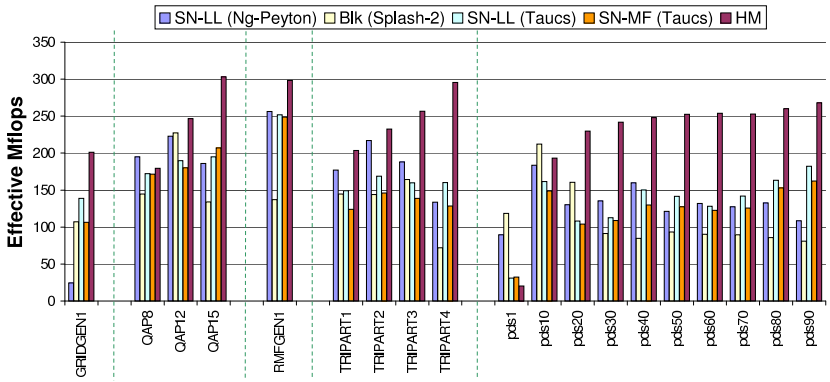


Figure 3.11: Performance of several sparse Cholesky factorization codes: IPM.

which belong to these areas. Their characteristics can be found in table 3.2.

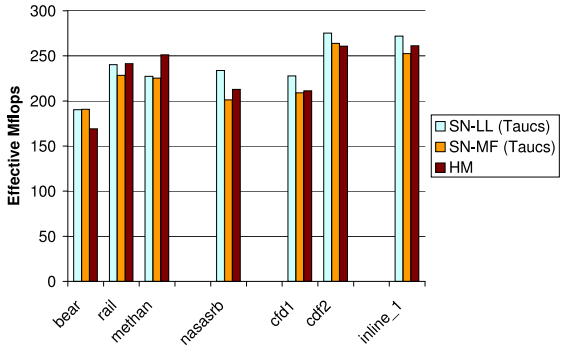


Figure 3.12: Performance of several sparse Cholesky factorization codes: FEA.

Figure 3.12 compares the performance of our sparse HM Cholesky with that obtained by the supernodal left looking and the multifrontal codes available in the TAUCS package. We can see that, on these types of matrices, our code is not as efficient as with the IPM matrices. However, it is still competitive, with performance similar to TAUCS codes.

3.6.3 Analysis of sparse hypermatrix Cholesky

Next, we analyze the performance of variants of our sparse HM Cholesky implementation. In all cases the code follows a right looking scheduling with a fixed partitioning of the hypermatrix (the elimination tree is consequently not used at all). We use a 4×32 data matrix size as stated in section 3.4.2.

Usage of windows within data submatrices

Figure 3.13 shows the results obtained with several variants of our sparse HM Cholesky code on matrices taken from IPM. The first and second bars allow for the evaluation of the usage of windows within data submatrices (in a 2D layout

of such submatrices). The usage of windows clearly improves the performance of our sparse hypermatrix Cholesky algorithm.

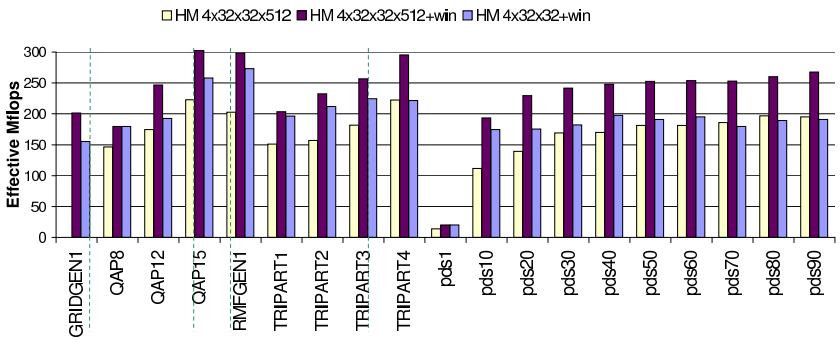


Figure 3.13: Sparse HM Cholesky: performance for several input matrices in IPM.

The reason is the large reduction in operations on zeros when windows are used within data submatrices. Figure 3.14 shows the increase in number of floating point operations in sparse HM Cholesky with blocks of size 4×32 w.r.t. the minimum (with no operations on zero elements). The number of unnecessary operations on zeros elements is very large for all matrices. When windows are used, however, there is a large reduction in the number of operations on zeros.

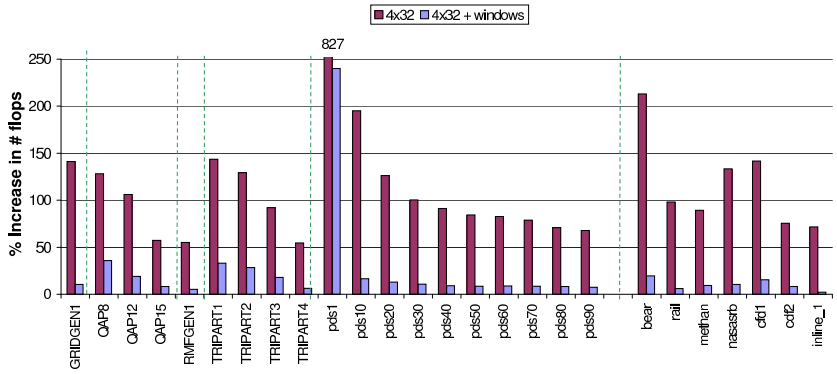


Figure 3.14: Increase in number of floating point operations in sparse HM Cholesky w.r.t. the minimum: windows reduce the number of operations on zeros.

2D layout and scheduling

The second and third bars in figure 3.13 show the results of 2D and 1D data layouts and scheduling of the data submatrices (windows are used in both cases). The former uses upper levels in the HM with sizes 32×32 and 512×512: each

of them allows for efficient use of the corresponding level of cache. For the latter, we define only an upper level with sizes 32×32. We observe that a 2D data layout and scheduling of the computations is beneficial to the efficient computation of the Cholesky factorization of large sparse matrices.

Overhead: operations on zeros

We have included matrix pds1 in figure 3.11 to show that for small matrices the hypermatrix approach is usually very inefficient. This is due to the large overhead introduced by blocks which have many zeros. Figure 3.15 shows the percentage of increase in number of flops in sparse HM Cholesky (using windows within data submatrices of size 4×32) w.r.t. the minimum (using a Compact Sparse Row storage). For large matrices however, blocks are quite dense and the overhead is much lower. Then, sparse HM Cholesky can obtain over half of the processor's peak performance.

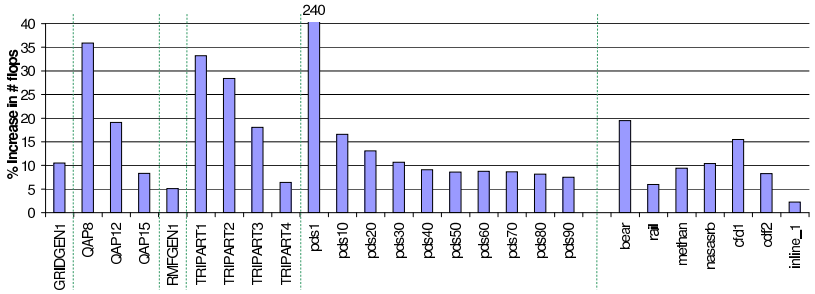


Figure 3.15: Sparse HM Cholesky using windows in data submatrices of size 4x32: Increase in number of operations.

Matrix Multiplication: efficiency of different routines

We focus on matrix multiplication because it is, by far, the most expensive operation within the Cholesky factorization. We use four matrix multiplication operations. FULL refers to the routine where all elements in data submatrices are used, i.e. no windows are used. WIN_1DC names the routine where windows are used for columns while WIN_1DR indicates the equivalent applied to rows. Finally, WIN_2D denotes the case where windows are used for both columns and rows. Figure 3.16 shows the percentage of calls to each matrix multiplication routine for each matrix taken from IPM.

Figure 3.17 shows the percentage of multiplication operations performed by each of the four $A \times B^T$ routines we use. We can observe that the number of floating point operations is not proportional to the number of calls to each matrix multiplication subroutine. For instance, one call to routine FULL for data submatrix size 4×32 produces $4 \times 4 \times 32$ (multiply-subtract) operations, while one call to routine WIN_2D computes less operations (which can be as low as one).

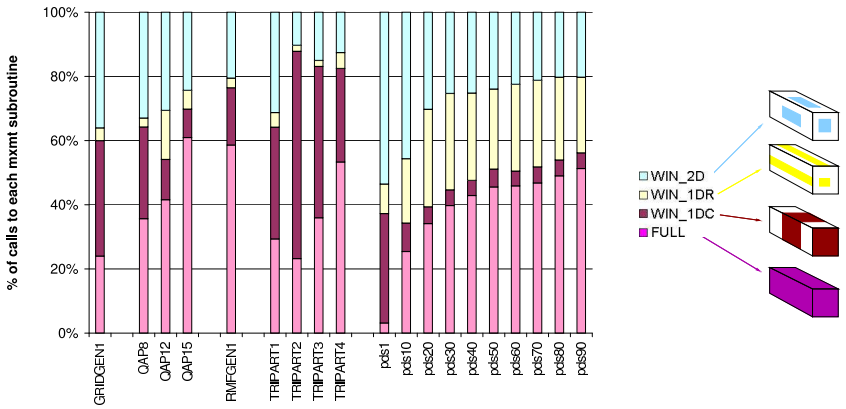


Figure 3.16: Sparse HM Cholesky: Percentage of calls to each $A \times B^T$ subroutine type.

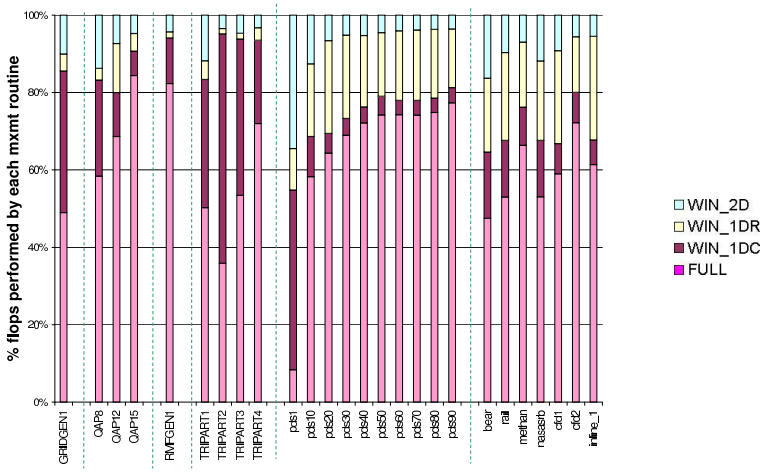


Figure 3.17: Sparse HM Cholesky: flops per $A \times B^T$ subroutine type.

In addition, the time spent in each routine is also different. Let's consider matrices QAP8 and QAP12. For each of them, figure 3.18 shows three bars. Given our four $A \times B^T$ routines, these bars give their different proportions in number of calls, number of flops, and time.

The explanation for the differences in computation time come from the efficiency of each code. FULL refers to the routine where all matrix dimensions are fixed. This is the most efficient of the four routines and can perform matrix multiplications faster. WIN_2D denotes the case where windows are used for both columns and rows. Thus, for the latter, no dimensions are fixed at compilation time and it becomes the least efficient of all four routines. WIN_1DC computes matrix multiplications more efficiently than WIN_1DR because of the constant leading dimension used for all three matrices. This is the reason why the performance obtained for matrices in the TRIPARTITE set is better than that obtained for matrices with similar size belonging to the PDS family. The

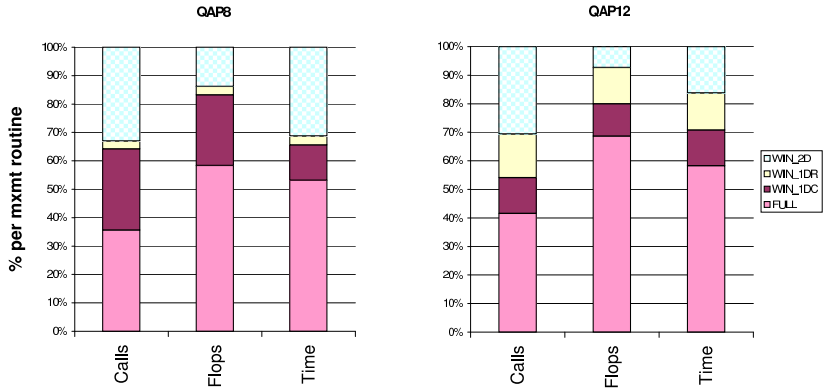


Figure 3.18: Calls, flops and time per $A \times B^T$ subroutine type: QAP8 and QAP12.

performance obtained for the matrix with the largest number of floating point operations in our test suite, namely matrix GRIDGEN1, is less than half of the theoretical peak for the machine used. This is basically due to the dispersion of data in this matrix which leads to the usage of the FULL $A \times B^T$ routine less than 50% of the time. In addition, a large number of calls is done to the least efficient routine WIN_2D. This routine is used to compute about 10% of the operations.

3.7 Amalgamation

Amalgamation [11] has been used for a long while in sparse codes. It consists in joining supernodes with different structures allowing for the presence of zeros. This is used to produce larger blocks and thus improve the performance by a better use of the machine resources via BLAS3 routines. Next, we present the ways in which we introduce amalgamation in our sparse hypermatrix Cholesky code. First we introduce *Intra-block* amalgamation. Second, we present *Hypermatrix oriented supernode amalgamation.*

3.7.1 Intra-block amalgamation

Approximately 90% of the sparse Cholesky factorization time comes from matrix multiplications. Thus, a large effort must be devoted to perform such operations efficiently. We have 4 codes specialized in the multiplication of two matrices. Figure 3.19 shows graphically the data used by each routine. The most efficient is the one on the left. Their efficiency diminishes as we move to the right. The names and characteristics of each routine, from left to right, are: *FULL* operates on the entire matrices, *WIN_1DC* uses windows in columns; *WIN_1DR* uses windows in rows; finally, *WIN_2D* uses windows in both dimensions and is the slower code amongst all 4 codes.

In this section we present an aspect of the work we have done to improve the performance of our sparse Hypermatrix Cholesky factorization.

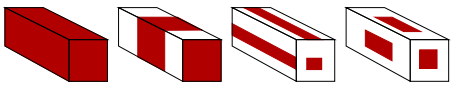


Figure 3.19: Four matrix multiplication routines deal with different input matrices.

The usage of windows reduces the number of unnecessary operations on zeros. However, routines which work on submatrices with windows have a considerably lower performance than that of the routine for full data submatrices, where all leading dimensions and loop trip counts are fixed at compilation time (see the analysis in [90]). Performing a slightly higher number of operations with a faster routine could some times pay off. For this reason we have decided to add the possibility to extend windows with rows or columns full of zeros. This work was presented in [94].

Figure 3.20 shows a rectangular data submatrix with a window defined within it. That window saves operations in both columns and rows. Each time this matrix needs to be multiplied by another matrix the *WIN_2D* code will be used. Since this is the slower code amongst our 4 matrix multiplication routines, this can produce a reduction in performance.

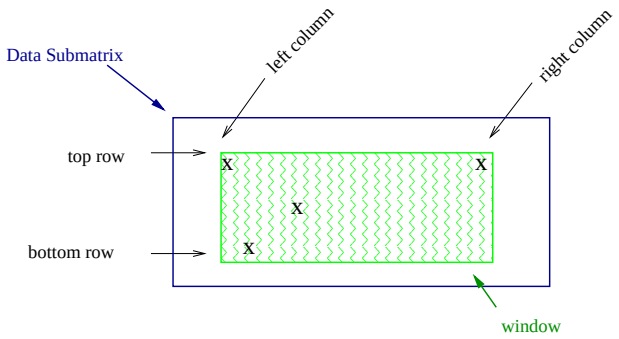


Figure 3.20: Original data submatrix before intra-block amalgamation.

Figure 3.21 shows how we can extend the window row-wise. We are aware that the resulting window will have rows full of zeros either at the top or at the bottom. This extension introduces extra overhead due to the extra number of operations on such zeros. However, this intra-block amalgamation can reduce the number of calls to routine *WIN_2D*. Instead, some new calls to *WIN_1DC* can be done. Since the latter routine is more efficient than the former, this can result in a performance improvement as long as the number of unnecessary operations on zeros is not too large. We only perform such amalgamation if the dimension of the window is close to the dimension of the data submatrix. We define a threshold for rows and another one for columns.

Figure 3.22 shows how we can extend the window column-wise. In this case, the resulting window will have columns full of zeros in at least one of its sides. Again, this can reduce the number of times in which routine *WIN_2D* is used. In this case, such calls could be replaced by calls to *WIN_1DR*.

Finally, figure 3.23 shows how we can extend the window both row and column-wise. In this case, the resulting window matches the whole data subma-

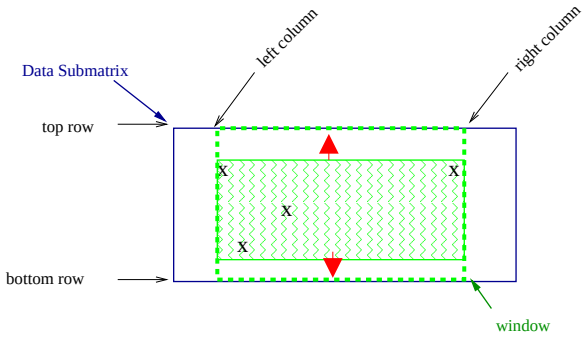


Figure 3.21: Data submatrix after row-wise intra-block amalgamation.

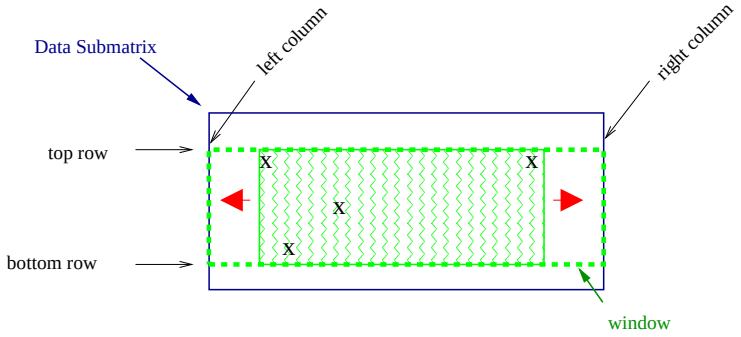


Figure 3.22: Data submatrix after column-wise intra-block amalgamation.

trix. Again, this action can reduce the number of times routine *WIN_2D* is used. In this case, such calls could be replaced by calls to any of the 3 other matrix multiplication routines (either *FULL*, *WIN_1DC* or *WIN_1DR*) depending on the other matrix involved in the multiplication.

Results

Given a 4×32 data block, we have introduced intra-block amalgamation in rows and columns. We have used values from 0 to 3 for row-wise amalgamation and 0 to 9 for column-wise amalgamation. A row-wise amalgamation with value 3 means that no routines dealing with windows in rows would be used. Next, we show details on the performance obtained using several values of row and column-wise amalgamation for some sparse matrices: QAP8 (fig. 3.24) , QAP12 (fig. 3.25), TRIPART1 (fig. 3.26), TRIPART2 (fig. 3.27), pds10 (fig. 3.28) and pds20 (fig. 3.29). Effective Mflops are presented. They refer to the number of useful floating point operations performed per second. This metrics excludes useless operations on zeros performed by the HM Cholesky algorithm when data submatrices contain zeros.

In all cases, row-wise amalgamation with values 1 and 2 obtain the best performance. There are several values for column-wise amalgamation with similar performance. We have chosen a value of 5 for the latter since this was often the best one or nearly the best.

Figure 3.30 shows the performance obtained with our sparse HM Cholesky

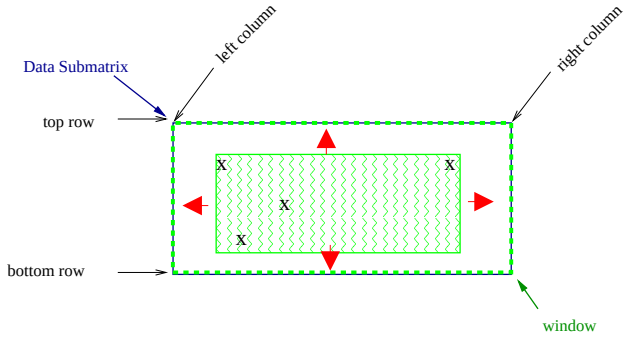


Figure 3.23: Data submatrix after applying both row and column-wise intra-block amalgamation.

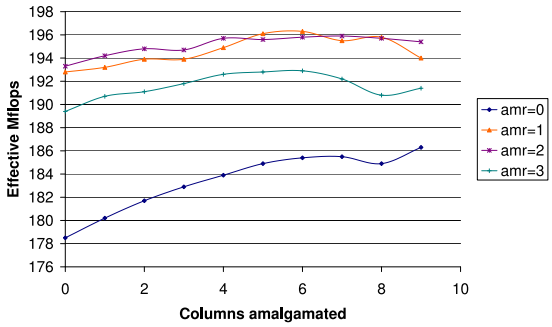


Figure 3.24: Intra-block amalgamation: matrix QAP8.

code with and without intra-block amalgamation. In both cases SML routines and windows have been used. The block sizes are also the same in both cases: 4×32 as the data submatrix size; 32×32 for the next pointer level, and 512×512 as the upper pointer level.

Finally, we present results obtained by five different sparse Cholesky factorization codes. Figure 3.31 shows the results obtained with each of them for the set of matrices introduced above. Matrix families are separated by dashed lines.

The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [137]. The second bar shows the performance obtained by the sequential version of a 2D block-oriented approach [155] as found in the SPLASH-2 [173] suite. Although submatrices are kept in a two-dimensional data layout this code fails to produce efficient factorizations for large matrices. The third and fourth bars correspond to sequential versions of the supernodal left-looking (SN-LL) and supernodal multifrontal (SN-MF) codes in the TAUCS package (version 2.2) [102]. In these codes the matrix is represented as a set of supernodes. The dense blocks within the supernodes are stored in a recursive data layout matching the dense block operations. The performance obtained by these two codes is quite uniform.

Finally, the fifth bar shows the performance obtained by our right looking sparse hypermatrix Cholesky code (HM). We have used windows [90] within data submatrices and SML [89] routines to improve our sparse matrix appli-

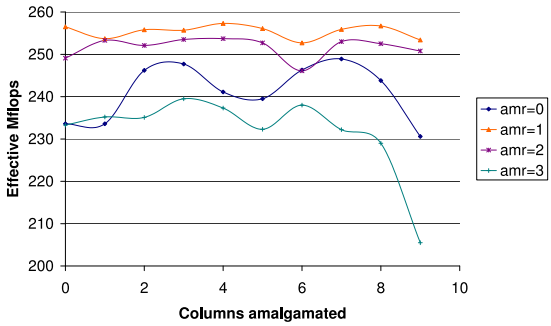


Figure 3.25: Intra-block amalgamation: matrix QAP12.

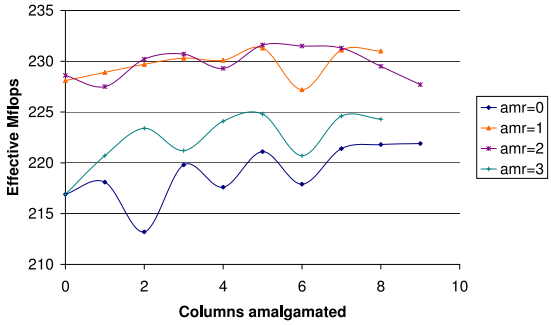


Figure 3.26: Intra-block amalgamation: matrix TRIPART1.

cation based on hypermatrices. Values 1 for row-wise and 5 for column-wise amalgamation have been used [94]. A fixed partitioning of the matrix has been used. We present results obtained for data submatrix sizes 4×32 and upper hypermatrix levels with sizes 32×32 and 512×512.

Conclusions

A performance improvement was always achieved for row-wise amalgamation with values 1 and 2. A value around 5 was usually the best for column-wise amalgamation on the matrices tested. Using intra-block amalgamation by rows with a value of 1, and by columns with a value of 5, produces a performance improvement between 3% and 12.9% with an average of 5.3% on our sparse matrix test suite on an R10000 processor.

When we introduce intra-block amalgamation we increase the overhead associated with the unnecessary operations on zeros. Thus, the next step towards performance improvements could come from a new data storage for data submatrices. In the future, we plan to add the possibility to store data submatrices as supernodes. In this way we could join several non-consecutive rows in a consecutive fashion in the same way as a code based on supernodes. We believe this could reduce the overhead since the storage and operation on zeros could be avoided while the efficient execution with BLAS3 would still remain.

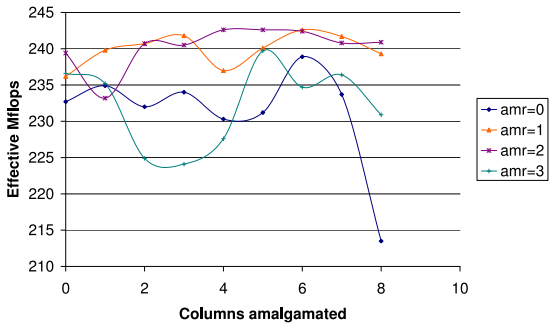


Figure 3.27: Intra-block amalgamation: matrix TRIPART2.

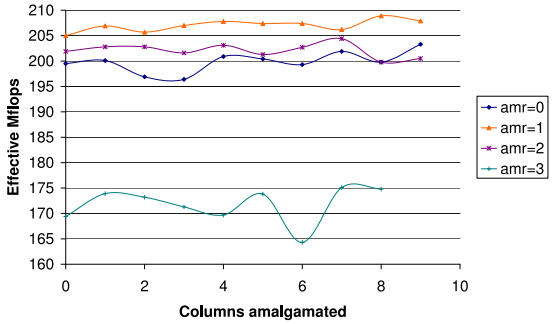


Figure 3.28: Intra-block amalgamation: matrix pds10.

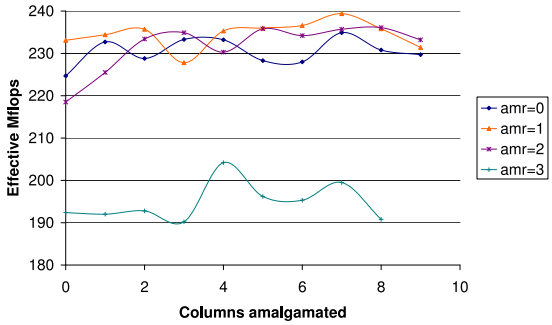


Figure 3.29: Intra-block amalgamation: matrix pds20.

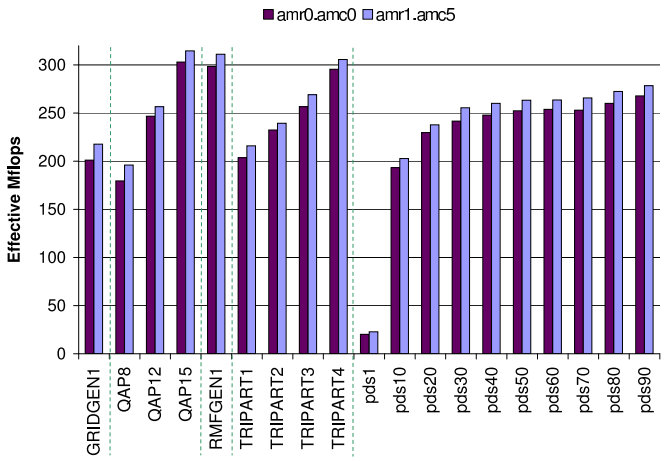


Figure 3.30: Performance of sparse HM Cholesky without and with intra-block amalgamation.

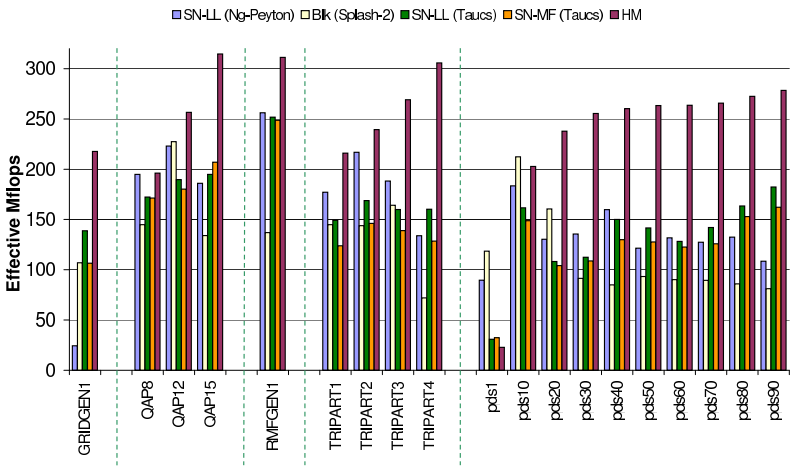


Figure 3.31: Performance of several sparse Cholesky factorization codes.

3.7.2 Hypermatrix oriented supernode amalgamation

We have been using a fixed partitioning of the hypermatrix. Now, we want to use a variable partitioning of the hypermatrix based on the information of the frontal tree. Such partitioning can avoid unnecessary dependences and can expose more parallelism than a fixed-sized partitioning. However, this can produce very small blocks when supernodes have a reduced number of columns. Consequently, performance can degrade unless we amalgamate supernodes. In [12] the authors reported that a variable size blocking was introduced in PERMAS to save storage and to speed the parallel execution. However, that paper does not explain how the variable sized blocking is achieved. We have developed some supernode amalgamation algorithms addressed to the hypermatrix scheme.

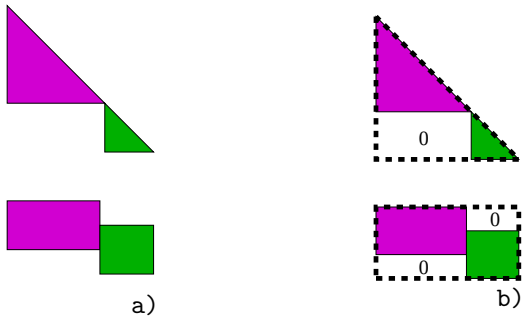


Figure 3.32: **a)** Two supernodes. **b)** Supernode amalgamation into a single supernode which contains zeros.

When two supernodes are amalgamated, zeros are introduced in the resulting supernode (figure 3.32). Usually amalgamation algorithms are parameterized with either the absolute number of zeros allowed, or the relative increase in the number of zeros after the amalgamation. Having too many zeros within supernodes outweighs the possible gains (efficient operations on larger blocks using BLAS3 routines). When we use a hypermatrix scheme, however, we can avoid the storage and computation of blocks of zero elements by keeping null pointers (figure 3.1). In addition, we can reduce it further by using windows within data submatrices. (figure 3.6). Thus, we can amalgamate supernodes to obtain new ones with larger number of columns (nodes) while the number of zeros included in the hypermatrix representation of such supernodes is not necessarily high. For this reason we need a special amalgamation algorithm for use with hypermatrices.

We define three amalgamation operations which can be used to merge supernodes in the supernodal (or frontal) tree:

- *merge_one* (fig. 3.33[3]): merge single children.
  Used to merge the frontal tree allowing only chains of nodes to merge.

[3]In these simple frontal trees each circle represents a column (node) or group of columns (supernode). The node at the top (parent) can only be processed when all the nodes below (children) have been processed. The size of the node refers to the amount of columns within it. The shaded nodes are amalgamated into their parent, forming a larger node.

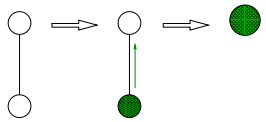


Figure 3.33: Merge one child node into its parent.

- *merge_all* (fig. 3.34): merge all children into parent node.
  Allows a parent to absorb all children.

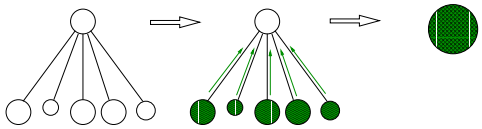


Figure 3.34: Merge all child nodes into their parent.

- *merge_any* (fig. 3.35) merge any children.
  Allows some children fronts to merge with their parent.

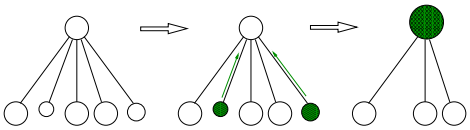


Figure 3.35: Merge some child nodes into their parent.

In all cases a post-order traversal of the frontal tree is used. The criterion used to decide whether nodes are assimilated into the parent is the following: we force amalgamation if the number of columns in the child (merge_one), or the sum of the columns in the children (some of them in merge_any and all of them in merge_all) is smaller than a given value. We call this value the *amalgamation threshold*.

We combine these three amalgamation operations to form four different algorithms.

- Algorithm 1: *merge_all(); merge_any();*
- Algorithm 2: *merge_one(); merge_all(); merge_any();*
- Algorithm 3: *merge_all(); merge_any(); merge_one();*
- Algorithm 4: *merge_one(); merge_all(); merge_any(); merge_one();*

In addition, we have implemented a variant which applies amalgamation only to nodes which are leaves of the frontal tree:

- Algorithm 5: *merge_leaves_one(); merge_leaves_all(); merge_leaves_any();*

The approach we use is the following: first, we allow for the amalgamation of whole subtrees using merge_all(). In this case the sum of the number of columns in all children must be lower or equal to the amalgamation threshold. Second, we traverse the tree again using merge_any() which allows some nodes to be

merged with the parent: at each step we amalgamate the child with the lowest number of columns and continue up to the point when the sum of the columns of all the children already amalgamated added to those of the next candidate node exceed the threshold.

However, we have variations of this approach to allow for amalgamation of chains of nodes: we allow to merge a single node either at the beginning of the process or at the end using merge_one(). Calling it in the beginning, we allow single nodes which are not leaves of the tree to merge into their parents. It can be observed below, when we present the results, that this hardly ever has any effect. In the case we call the merge_one() routine at the end of the process, we use a different amalgamation threshold: the total number of columns. We do this to force all single children to be merged. This has a positive effect on the performance obtained in the numerical factorization.

Results

Figure 3.36 presents the performance obtained in the factorization of matrices pds10 and TRIPART1 for each amalgamation algorithm and threshold value. Results for algorithms 1 and 2 are almost equal. The same happens with algorithms 3 and 4. This means that using a first step which merges a single child with its parent is irrelevant. However, doing this step at the end of the amalgamation process can pay off. Algorithms 3 and 4 are systematically the best. Also, the best values for the amalgamation threshold are similar for matrices of several sizes. We will study this in the next section. Finally, algorithm 5 is clearly the worst one in terms of performance. Merging only leaf nodes produces many small fronts for small and medium amalgamation thresholds. Data submatrices are then too small and the performance obtained is poor. Similar results were obtained on the other matrices in our test suite. We will use amalgamation algorithm 4 for the rest of our experiments.

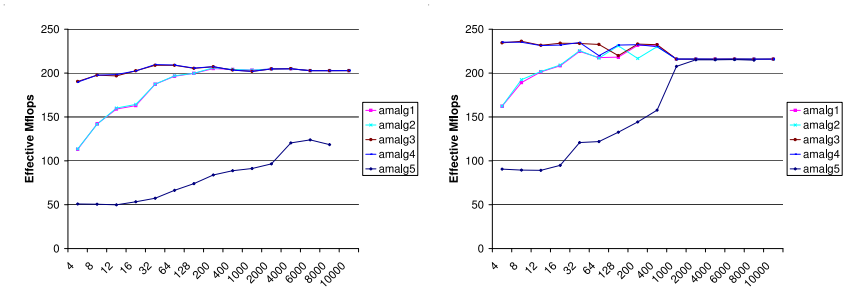


Figure 3.36: Performance of five amalgamation algorithms on matrices pds10 (left) and TRIPART1 (right).

Analysis: amalgamation threshold

Figure 3.37 shows the effective Mflops obtained for each matrix using amalgamation algorithm 4. We reckon effective Mflops counting only operations on nonzero elements since operations performed on zeros are non productive. The values of the amalgamation threshold range from 4 to a very large number for

which the amalgamated frontal tree has only one front. In such case, the resulting hypermatrix is equivalent to one with a static (fixed) partitioning. In some cases, several values for the threshold produce a single front in the tree. That is the reason why several matrices have some bars with exactly the same height (same effective Mflop rate). On the other hand, we can observe that, in general, very small threshold values obtain the worst performance.

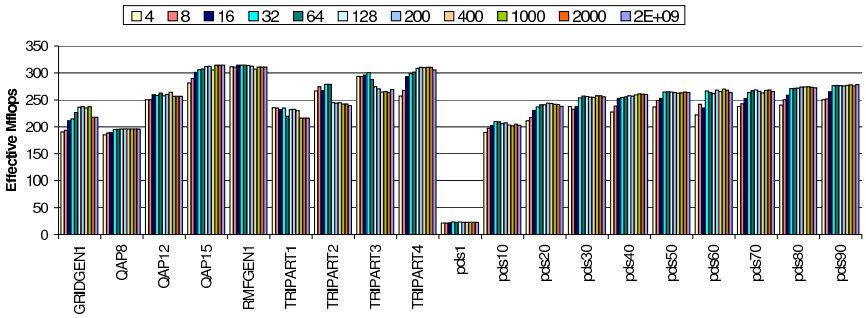


Figure 3.37: Performance obtained with each amalgamation threshold using algorithm 4.

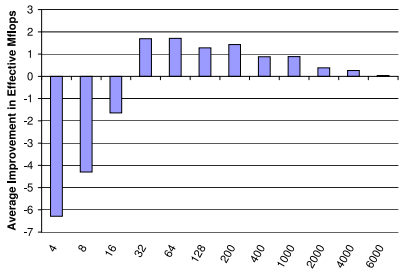


Figure 3.38: Average improvement per amalgamation threshold compared to the static partitioning.

Figure 3.38 shows the average improvement obtained using our amalgamation algorithm 4 on our test matrix suite for several values of the amalgamation threshold. Large values produce hypermatrix structures similar to those obtained using a static partitioning. Consequently, the resulting performance is similar to that of the static partitioning. On the other hand, the use of small values often results in a degradation of performance. The reason for this comes from the reduction in the size of some data submatrices. This occurs when the supernodes being amalgamated contain very few columns. We only force amalgamation of nodes with less columns than the threshold value. The performance degradation occurs when the threshold used is smaller than the number of columns used in the creation of the matrix multiplication routines with fixed dimensions (32 columns for our target platform). This means that, for any operation on such submatrices we can only use routines *WIN_1DC*, which uses windows along the columns, and *WIN_2D*, which uses windows in both dimensions and is the slowest amongst all 4 codes. In addition, each block is smaller,

but there are more blocks. Thus, more subroutine calls are necessary with the consequent overhead. Let us illustrate this point with some details on the hypermatrix Cholesky factorization of matrix GRIDGEN1.

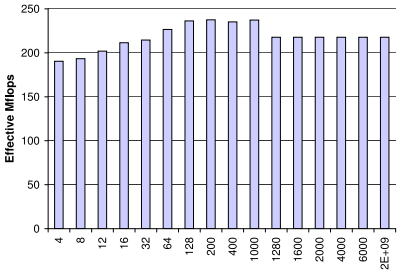


Figure 3.39: Effect on performance of amalgamation algorithm 4 with different threshold values in factorization of matrix GRIDGEN1.

Figure 3.39 shows the effective Mflops obtained in the factorization after the application of our amalgamation algorithm 4 for different threshold values on matrix GRIDGEN1. The best Mflop rate is obtained for threshold values in the range 128 to 1000. The left part of figure 3.40 shows the increase in the total number of floating point operations of the hypermatrix Cholesky algorithm w.r.t. the minimum, i.e. without any operations on zeros. The lowest increase in the number of operations is obtained when the threshold value equals 4. However, the resulting performance in Mflops is the worst amongst all the values tested. The right part of figure 3.40 shows the total number of calls to any of the 4 different matrix multiplication routines performed for each threshold value. We can see that the number of calls for the leftmost bar, which corresponds to a threshold value of 4, is the highest. In spite of performing the lowest number of floating point operations amongst all values of amalgamation threshold, it issues more calls to the matrix multiplication routines. Furthermore, the percentage of calls to the slower routine is higher than those done using the other threshold values.

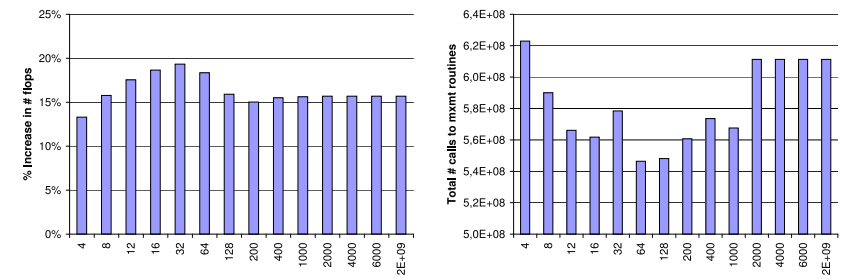


Figure 3.40: Increase in total number of floating point operations (left) and total number of calls to $A \times B^t$ routines (right) on matrix GRIDGEN1 with amalgamation algorithm 4.

Figure 3.41 shows the percentage of calls to each of the four matrix multiplication routines and figure 3.42 presents the percentage of floating point operations performed by each of them. From top to bottom, the four parts of each bar

correspond to calls to routine *WIN_2D*, *WIN_1DR*, *WIN_1DC* and *FULL*. For amalgamation threshold 4 we see that about 37% of the calls are made to the least efficient routine *WIN_2D* (figure 3.41) which computes only about 10% of the operations (figure 3.42). The best Mflop rate for the hypermatrix Cholesky factorization is obtained from those cases with the higher percentage of floating point operations computed using the *FULL* routine. Amongst them, the ones with lower number of operations on zeros provide the best results. Actually, this was the reason why some intra-block amalgamation was useful to improve performance.

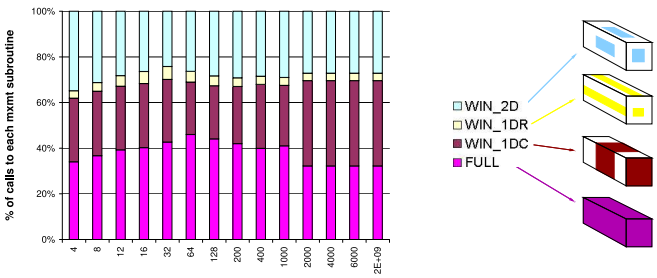


Figure 3.41: Percentage of calls to each AxB^t subroutine type: GRIDGEN1.

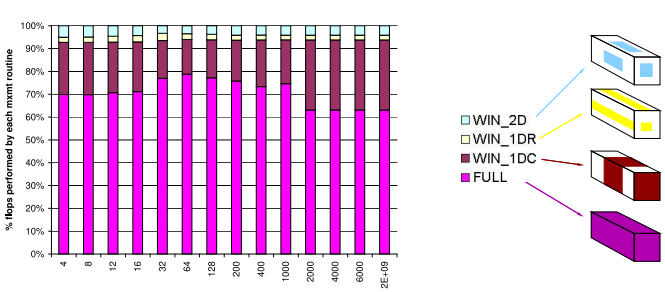


Figure 3.42: Floating point operations per AxB^t subroutine type: GRIDGEN1.

Similar results have been observed for the other matrices in the test suite (figure 3.37). Values of amalgamation threshold under 32 usually yield performance loss. Recall that we fix the number of columns to 32 in the routines which operate on small matrices with fixed dimension. In most cases values of 32 and 64 are the optimum. However, the largest matrices can benefit from larger values: e.g. from 128 to 400 (GRIDGEN1) or 128 to 2000 (TRIPART4).

Figure 3.43 shows the percentage of floating point operations performed by each matrix multiplication routine for a fixed partitioning of the hypermatrix (left) and a dynamic partitioning using the elimination tree and amalgamation algorithm 4 with a threshold value of 32 (right).

Figure 3.44 compares the performance obtained with both codes. The higher gain in Mflops (figure 3.44) is achieved in those cases where the percentage of flops performed by our faster matrix multiplication routine (*FULL*) increases substantially and the operation on zero elements is reduced: 16,5% (TRIPART2), 11,7% (TRIPART3), 9% (TRIPART1 and GRIDGEN1).

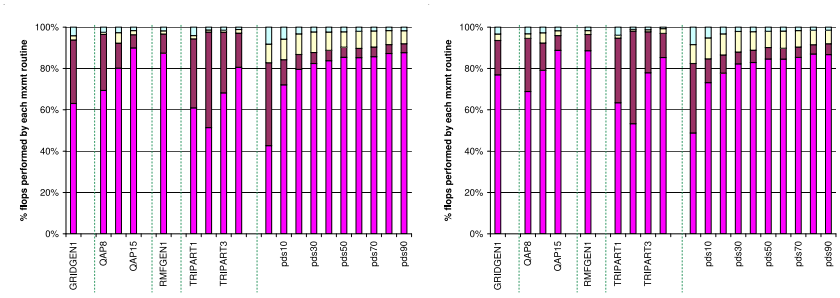


Figure 3.43: Percentage of flops performed by each AxB^t routine: static (left) vs dynamic (right).

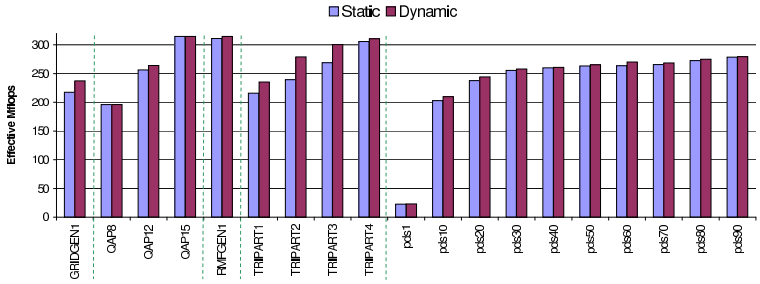


Figure 3.44: Performance of hypermatrix Cholesky: static vs dynamic partitioning.

Finally, we present results obtained by five different sparse Cholesky factorization codes. In all cases matrices have been reordered using METIS and a postorder of the resulting elimination tree. Figure 3.45 shows the results obtained with each of them for the set of matrices introduced above. Matrix families are separated by dashed lines.

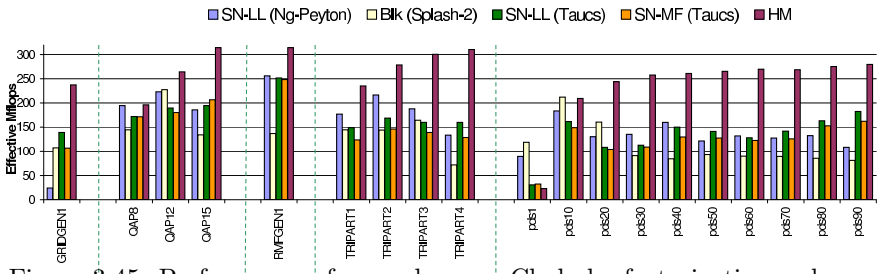


Figure 3.45: Performance of several sparse Cholesky factorization codes.

The first bar corresponds to a supernodal left-looking block Cholesky factorization (SN-LL (Ng-Peyton)) [137]. The second bar shows the performance obtained by the sequential version of a 2D block-oriented approach [155] as found in the SPLASH-2 [173] suite. Although submatrices are kept in a two-dimensional data layout this code fails to produce efficient factorizations for large matrices. The third and fourth bars correspond to sequential versions

of the supernodal left-looking (SN-LL) and supernodal multifrontal (SN-MF) codes in the TAUCS package (version 2.2) [102]. In these two codes the matrix is represented as a set of supernodes. The dense blocks within the supernodes are stored in a recursive data layout matching the dense block operations. In both cases the vendor BLAS library is used. The performance obtained by these two codes is quite uniform. While the first two codes can be considered out-of-date, TAUCS codes are known to provide reasonable performance [68].

Finally, the fifth bar shows the performance obtained by our right looking sparse hypermatrix Cholesky code (HM). We have used windows [90] within data submatrices and SML [89] routines to improve our sparse matrix application based on hypermatrices. Values 1 for row-wise and 5 for column-wise amalgamation have been used [94]. A dynamic partitioning of the matrix has been used. We present results obtained for data submatrix sizes 4×32 and upper hypermatrix levels with sizes 32×32 and 512×512. These block sizes were chosen to have data reused in each level of the memory hierarchy of the machine used.

Number of fronts

The supernodal (or frontal) tree describes the structure of the matrix. This provides one source of parallelism: different branches can be factored simultaneously. Once those branches are factored, their parent can be treated. Figure 3.46 shows the number of fronts obtained with each amalgamation algorithm on matrices pds10 and TRIPART1. We must note that very high number of fronts are obtained with small front sizes for all algorithms. Algorithm 5 is the one providing the largest number of fronts. However, we saw in section 3.7.2 that the performance of algorithm 5 is clearly the worst.

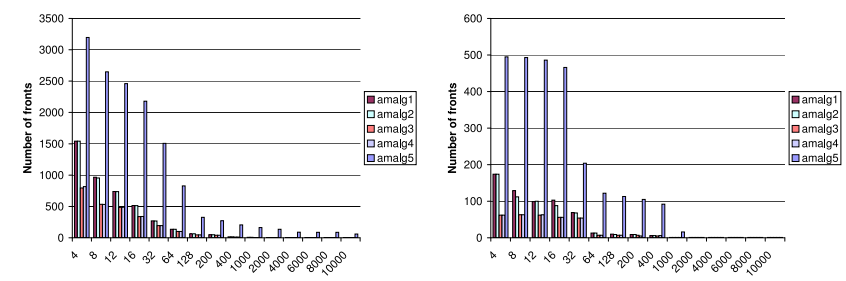


Figure 3.46: Number of fronts obtained with each amalgamation algorithm: matrix pds10 (left) and TRIPART1 (right).

Let's consider again amalgamation algorithm 4. Figure 3.47 shows the number of fronts per amalgamation threshold for each matrix using amalgamation algorithm 4. As the amalgamation threshold grows, the number of fronts tends to 1. However, for small values the number of fronts becomes large. In section 3.7.2 we saw that small values of amalgamation threshold yield reduced performance. The performance of a parallel code depends ultimately on the performance of the code executed on each node. Thus, it would not be advisable to use very small values of amalgamation threshold. The best performance is achieved with values around 64. In figure 3.47 we can see that the number of

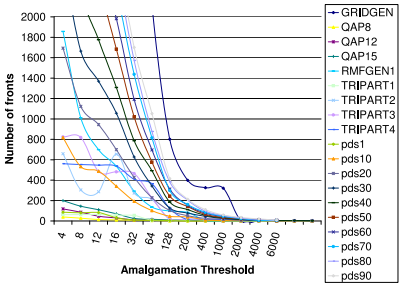


Figure 3.47: Number of fronts per amalgamation threshold with algorithm 4.

fronts for this amalgamation threshold is still high for medium and large matrices. Thus, we can still exploit this source of parallelism. We plan to implement a parallel version of the hypermatrix Cholesky factorization in the near future. Based on the information of the frontal tree we can create a variable sized partitioning of the hypermatrix which avoids unnecessary dependences and can expose more parallelism than a fixed-sized partitioning.

Conclusions

Given the sparse nature of the problems, we need to adapt the blocks as much as possible to the input data. In this section we have introduced a supernode amalgamation algorithm which takes into account the characteristics of a hypermatrix data structure. The resulting frontal tree is then used to create a variable-sized partitioning of the hypermatrix. The resulting sparse hypermatrix Cholesky factorization is slightly faster than the one which uses a fixed-sized partitioning. It also reduces data dependencies which limit exploitation of parallelism from the frontal tree. We plan to implement a parallel hypermatrix Cholesky in the future.

3.8 Other considerations on sparse hypermatrix Cholesky factorization

3.8.1 Porting efficiency to a new platform

In this section we summarize the port of our application from a machine based on a MIPS R10000 process to a platform with an Intel Itanium2 processor. We address the optimization of the sparse Cholesky factorization based on a hypermatrix structure following several steps.

First we create our Small Matrix Library (SML) automatically as explained in section 2.3 . We have used 4×32 as data submatrix dimensions since these were the ones providing best performance on the R10000. Later in this document we will present the results obtained with other matrix dimensions. As we mentioned in section 3.4.4 we use windows within data submatrices since they have proved effective in reducing both the storage of and operation on zero elements.

Afterwards, we experiment with different intra-block amalgamation values. As an example, figure 3.48 shows the performance obtained using several val-

ues of intra-block amalgamation on the hypermatrix Cholesky factorization of matrix pds20 on an Itanium2. Each curve corresponds to one value of amalgamation along the rows. The curve at the top (amr=3) corresponds to the largest amalgamation threshold along the rows: three, for submatrices consisting of four rows. The one at the bottom corresponds to the case where this type of amalgamation is disabled (amr=0).

On the Itanium2 and using our matrix test suite, the worst performance is obtained when no amalgamation is done along the rows (amr=0). As we allow increasing values of the intra-block amalgamation along the rows the overall performance increases. The best performance for this matrix dimensions is obtained when amalgamation along the rows is three. This means that, for data submatrices of size 4×32, we will use no windows along the rows. This suggests that a larger number of rows could provide improved performance. We will analyze this issue in section 3.8.4. As we move right on the curves, we observe the performance obtained with increasing values of amalgamation threshold along the columns. The difference is often low, but the best results are obtained with values ranging from six to ten. This work will appear in [97].

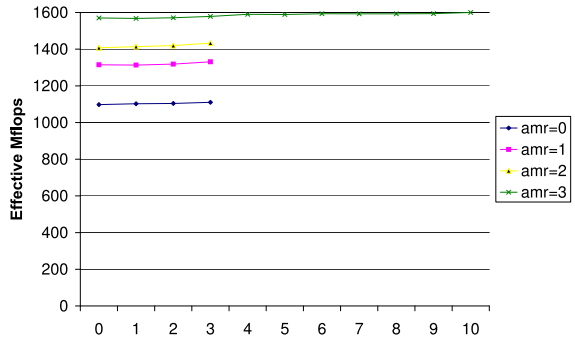


Figure 3.48: Sparse HM Cholesky on an Intel Itanium2: Performance obtained with different values of intra-block amalgamation on submatrices of size 4×32 on matrix pds20.

Notice that the threshold values providing best performance on the R10000 were different: one on the rows and five on the columns. The reason for this is the different relative performance of the matrix multiplication routines. The ability of one compiler to generate efficient code for routines which take windows into account can be different from its aptitudes when dealing with routines which do not use windows at all. And these capabilities can be different from those of the compiler found on another platform. As a consequence we get different optimal values for the intra-block amalgamation thresholds on each platform.

3.8.2 Sparse matrix reordering

A sparse matrix can be reordered to reduce the amount of fill-in produced during the factorization. Also it can be reordered aiming to improve parallelism. In all the tests presented so far we have been using METIS as the reorder algo-

rithm. This algorithm is considered a good algorithm when a parallel Cholesky factorization has to be done. Also, when matrices are relatively large, graph partitioning algorithms such as METIS usually work much better than MMD, the traditional Minimum Degree ordering algorithm [68].

Ordering sparse matrices for hypermatrix Cholesky

Although our current implementation is sequential, we have tried to improve the sparse hypermatrix Cholesky for the matrices produced by METIS. In this way, the improvements we get are potentially useful when we go parallel.

However, we have also studied several other algorithms. On small matrices in our matrix test suite, when the Multiple Minimum Degree (MMD) [123] algorithm was used the hypermatrix Cholesky factorization took considerably less time. However, as we use larger matrices (RMFGEN1, pds50, pds60, ...) the time taken to factor the resulting matrices became several orders of magnitude larger than that of METIS. We have also tried older methods [62] such as the Reverse Cuthill-McKee (RCM) and the Refined Quotient Tree (RQT). RCM tries to keep values in a band as close as possible to the diagonal. RQT tries to obtain a tree partitioning of a graph. These methods produce matrices with denser blocks. However the amount of fill-in is so large that the factorization time gets very large even for medium sized matrices.

METIS implements a Multilevel Nested Dissection algorithm. This sort of algorithms keep a global view of the graph and partition it recursively using the Nested Dissection approach [63] splitting the graph in smaller disconnected graphs. When these subgraphs are considered small, a local ordering algorithm is used. METIS uses the MMD algorithm for the local phase. By default, METIS changes to the local ordering strategy when the number of nodes is less than 200.

We have modified the code in METIS so that we can:

- Change the algorithm for the local reordering phase:
  We can now choose amongst MMD, RCM and RQT
- Change the threshold for switching to the local reordering phase:
  We can use a command line flag in our application to choose the switch value.

Our preliminary experiments with local ordering algorithms other than MMD result in performance loss in many cases for the default switch value. When the switch value is smaller (for instance 20) the RQT seems to be a good choice for some matrices. In some cases the performance gain over the default values in METIS was about 20%. Unfortunately, there is a large variation in the best switch value from one matrix to another. For this reason. we will need to develop some new heuristic which adapts the switch value to the current part of the graph being studied. This will be done after the thesis is presented. Thus, our contribution for the time being is limited to give hints for possible changes in a graph partitioning algorithm which could potentially return some performance improvements.

Ordering for Linear Programming problems

Working with matrices which arise in linear programming problems we may use sparse matrix ordering algorithms specially targeted for these problems. The METIS [106] sparse matrix ordering package offers some options which the user can specify to change the default ordering parameters. Following the suggestions found in its manual we have experimented with values which can potentially provide improved orderings for sparse matrices coming from linear programming problems. There are eight possible parameters. We skip the details of these parameters for brevity. However, for the sake of completeness, we include the values we have used: 1, 3, 1, 1, 0, 3, 60, and 5.

Using this configuration usually produces better sparse matrix orderings than the default configuration for the type of problems we are dealing with. This ordering results in faster sparse Cholesky numerical factorization. However, this comes at the expense of a larger ordering process which incurs in a larger ordering time. We have measured both the ordering and numerical factorization time obtained using the two configurations of METIS discussed above. We must take into account that Interior Point Methods (IPM), i.e. the methods which use the sparse Cholesky factorization on linear programming problems have an iterative nature. In each iteration a sparse Cholesky factorization is performed on matrices with different data but the very same structure. Thus, the ordering process can be performed only once, while the numerical factorization is repeated many times on different data. Until now, we have been using METIS default configuration. To evaluate the potential for the new ordering parameters we have measured the number of iterations necessary to amortize the cost of the improved matrix reordering. Figure 3.49 presents the number of iterations after which the specific ordering starts to be advantageous. We can see that in many cases the benefits are almost immediate. Consequently, in the rest of this work we will present results using the modified ordering process specific for matrices arising in linear programming problems. This work will appear in [97].

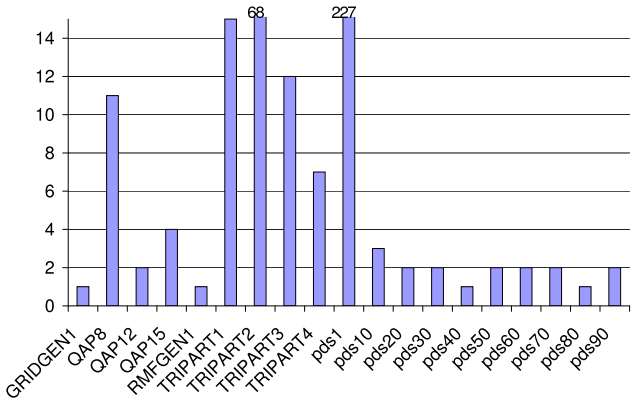


Figure 3.49: Number of iterations necessary to amortize cost of improved ordering.

3.8.3 Data submatrix storage: compression

In section 3.8.1 we showed that on an Intel Itanium2 and using data submatrices of size 4×32 the optimal threshold for amalgamation in the rows was three. This suggests that using data submatrices with a larger number of rows should be tried. However, the larger the blocks, the more likely it is that they contain zeros. As we have discussed in previous sections, the presence of zero values within data submatrices causes some drawbacks. Obviously, the computation on such null elements is completely unproductive. However, we allow them as long as operating on extra elements allows us to do such operations faster. On the other hand, a different aspect is the increase in memory space requirements with respect to any storage scheme which keeps only the nonzero values. Next, we present the way in which we can avoid some of this additional storage.

As we mentioned in section 3.4.4, we use windows to reduce the effect of zeros in the computations. However, we still keep the zeros outside of the window. Figure 3.50 shows two data submatrices stored contiguously. Even when each submatrix has a window we store the whole data submatrix as a dense matrix.

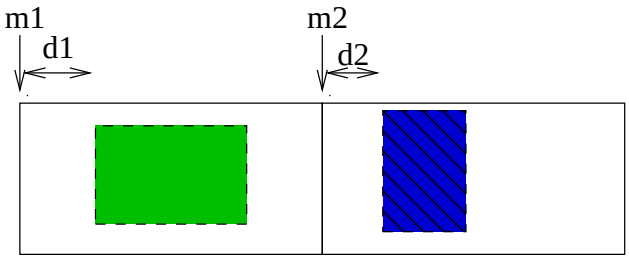


Figure 3.50: Data submatrices before compression.

However, we could avoid storing zeros outside of the window, i.e. just keep the window as a reduced dense matrix. This approach would reduce storage but has a drawback: by the time we need to perform the operations we need to either uncompress the data submatrix or reckon the adequate indices for a given operation. This could have a performance penalty for the numerical factorization. To avoid such overhead we store data submatrices as shown in figure 3.51.

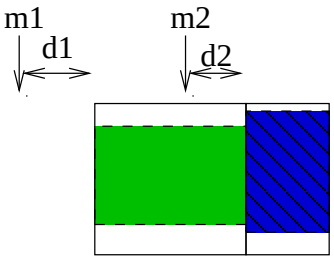


Figure 3.51: Data submatrices after compression.

We do not store zeros in the columns to the left and right of the window. However, we do keep zeros above and/or underneath such window. We do this

for two reasons: first, to be able to use our routines in the SML which have all leading dimensions fixed at compilation time (we use Fortran, which implies column-wise storage of data submatrices); second, to avoid extra calculations of the row indices. In order to avoid any extra calculations of column indices, we keep pointers to an address which would be the initial address of the data submatrix if we were keeping zero columns on the left part of the data submatrix. Thus, if the distance from the initial address of a submatrix and its window is d_x we keep a pointer to the initial address of the window with d_x subtracted from it. These pointers are kept in the last level of pointers in the hypermatrix. Thus, when the numerical factorization takes place, we can take advantage of performing dense operations on data submatrices, i.e. use our efficient routines working on small matrices and avoid complex calculation of indices. Note that in the presence of windows, we will never access the zero columns to the left or right of a window, regardless of having them stored or not.

Figure 3.52 presents the savings in memory space obtained by this method compared to storing the whole data submatrices of size 4×32. We can observe that the reduction in memory space is substantial for all matrices. This work will appear in [97].

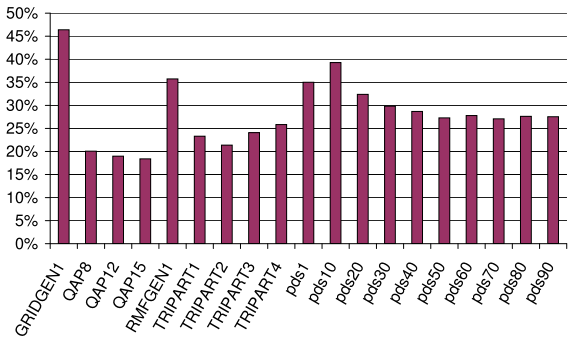


Figure 3.52: HM structure: reduction in space after submatrix compression.

3.8.4 Larger data submatrices: performance

The reduction in memory space allows us to experiment with larger matrix sizes (except on the largest matrix in our test suite: GRIDGEN1). Figure 3.53 presents the variation in execution time when the number of rows per data submatrix was increased to 8, 16 and 32. In almost all cases the execution time increased. Only matrices of the TRIPARTITE family benefited from the use of larger submatrices.

We must note that the performance obtained with matrices of size 8×32 is worse that that obtained with submatrices of size 16×32. The reason for this is the relative performance of the routines which work on each matrix size. The one with larger impact on the overall performance of the sparse hypermatrix Cholesky factorization is the one with fixed matrix dimensions and loop trip counts. The corresponding routine for each matrix size obtains the

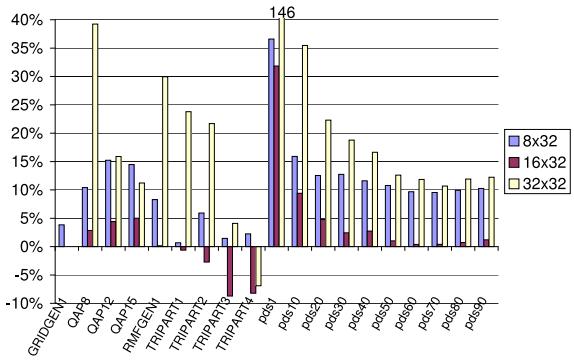


Figure 3.53: Sparse HM Cholesky: variation in execution time for each submatrix size relative to size 4×32.

peak performance shown in table 3.4. We can observe that the efficiency of the routine working on matrices with four rows is similar to the one which works on matrices with eight rows. However, the overhead, in terms of additional zeros, is much larger for the latter. This explains their relative performance. However, the improved performance of the matrix multiplication routine when matrices have 16 rows can pay off. Similarly to the comparison between codes with eight rows and four, using matrices with 32 rows produces a performance drop with respect to the usage of data submatrices with 16 rows. This work will appear in [97].

4×32	8×32	16×32	32×32
4005	4080	4488	4401

Table 3.4: Performance of the $C = C - A \times B^T$ matrix multiplication routine for each submatrix size.

3.8.5 Sparse HM Cholesky vs WSSMP: Performance

In [68] an exhaustive evaluation of several state-of-art sparse Cholesky factorization packages was done. According to that work, the numerical sparse Cholesky factorization in the Watson Sparse Matrix Package (WSMP) [74] was very often the fastest. For this reason, we wanted to compare the performance obtained by our code with WSSMP (the name of the sparse Cholesky routine in WSMP based on the multifrontal algorithm [124]). The BLAS library used by WSSMP was ATLAS version 3.7.11.

In addition, this package has its own sparse matrix reordering code [73] based on a Nested Dissection algorithm. This code is supposed to handle matrices from the linear programming field effectively [72]. The user can activate this special ordering with one of the routine parameters: namely, setting `iparm(20)=1`. We will refer to such case as WSSMP LP in the figures.

We have run both codes on an Itanium2 machine measuring the time taken

by each of them. Note that the number of operations performed in each case is different since the ordering algorithms used were different: for our sparse hypermatrix Cholesky (HM) we used METIS with the configuration suitable for linear programming problems (METIS LP in the figures). Data submatrices of size 4×32 were used in all cases except for the TRIPART family on which 16×32 submatrices were used. A fixed partitioning of the hypermatrix was used except for the GRIDGEN1 matrix and the TRIPART family. In these cases we used amalgamation algorithm 3. In the former case the amalgamation threshold was 160. On the TRIPART family the threshold used was 32. Figure 3.54 show the performance of these two codes relative to the best one.

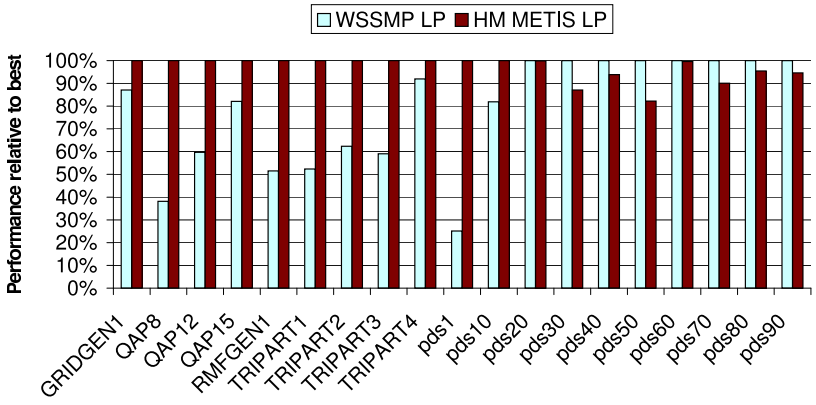


Figure 3.54: Sparse HM Cholesky vs WSSMP LP.

We can observe that in many cases HM METIS LP outperforms WSSMP LP, getting close to the latter in those cases where the latter was the best.

In order to see the influence of the different orderings we have run WSSMP with the default ordering algorithm (WSSMP in the next figure) and also using METIS LP. Figure 3.55 show the performance of these two codes relative to the best one.

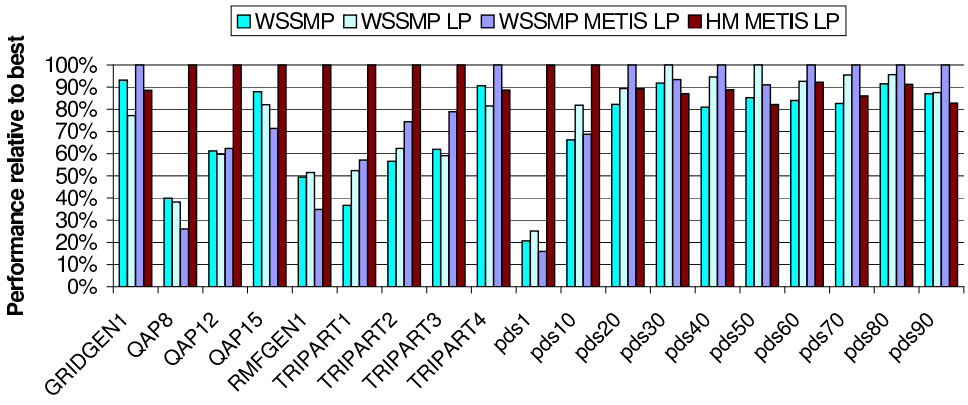


Figure 3.55: Sparse HM Cholesky vs WSSMP using several ordering algorithms.

We can observe that WSSMP outperforms WSSMP LP on some matri-

ces (GRIDGEN1, QAP family, TRIPART3 and TRIPART4). When we use WSSMP METIS LP the performance increases for several matrices (GRIDGEN1, TRIPART family, and some matrices of the pds family). We can observe that no single ordering heuristic obtains the best performance across the whole matrix test suite. We can also observe that our sparse hypermatrix Cholesky obtains the best performance for several matrices and over 80% of the best WSSMP case for the rest.

3.8.6 Future work

Next, we present some ideas for further improvements on sparse hypermatrix Cholesky factorization.

Column versus row storage

We have been using a column-wise storage for data submatrices since this is the standard Fortran order. Now, we want to see whether a row-wise storage could benefit our algorithms. The possible advantage for the sparse Cholesky factorization comes from the fact that about 90% of the time is spent in a matrix multiplication such as $C = C - A * B^T$. Having the data submatrices stored row-wise could result in streaming of all 3 matrices (accessing matrices with stride 1). This would improve spatial locality and data could be brought from upper levels of memory faster. On a Cholesky factorization, assuming Fortran column major storage, this is equivalent to performing operations on an upper triangular matrix (U) rather than the lower triangle (L). We plan to implement this in the near future.

Supernodes within hypermatrix data submatrices

The results obtained with the intra-block amalgamation in section 3.7.1 suggest that replacing dense data submatrices with supernodes could speed up our code. Although the code is regular and fast, using plain dense matrices we produce many unnecessary operations on zeros. We believe that we could improve the effective Mflop rate of our algorithm if we packed the full rows within those submatrices using a supernodal scheme. We purpose to modify the part of the code which corresponds to the data submatrices to be able to deal with blocks stored as supernodes. These changes however are not trivial. Thus, we will implement it in the future.

Parallel sparse hypermatrix Cholesky

We have been trying to get an efficient sequential implementation of the sparse hypermatrix Cholesky factorization. Once we have obtained it, we plan to develop a parallel version which uses the information provided by the elimination tree. Different branches in the tree can be computed in parallel quite easily. However, we will need to introduce synchronization for the final step, i.e. updating a diagonal block with the contributions produced by the off-diagonal blocks to its left. Further parallelism will be obtained from operations within the large blocks which result at the top of the elimination tree. We plan to do this in the future.

3.8.7 Conclusions

The sparse hypermatrix Cholesky factorization usually improves its performance as the problem size gets larger. There are two reasons for this. The overhead due to unnecessary operations on zeros is usually reduced. The other is that since blocks tend to be larger, more operations are performed by efficient $A \times B^T$ routines. The best performance is obtained from those cases which have few operations on zeros to cause overhead, and which make extensive use of the most efficient matrix multiplication routines.

A two dimensional partitioning of the matrix is necessary for large sparse matrices. Compared to codes which do a 1D partitioning of the matrix, our code can result in a better usage of the memory hierarchy: locality is properly exploited with the two dimensional partitioning of the matrix which is done in a recursive way using the HM structure.

The overhead introduced by storing zeros within dense data blocks in the hypermatrix scheme can be reduced by keeping information about a dense subset (window) within each data submatrix. Although some overhead still remains, using windows and SML routines our sparse HM Cholesky often obtains over half of the processor's peak performance for medium and large sparse matrices factored sequentially in-core. When windows are used bit vectors are unnecessary.

In spite of the simple fixed-sized blocking used, the sparse hypermatrix Cholesky factorization which uses windows and SML routines is highly competitive. On some problems, a variable partitioning of the matrix results in performance improvements.

The efficient execution of a program requires the configuration of the software to adapt it to the problem being solved and the machine used for finding a solution. We have shown the way in which we can tune our sparse hypermatrix Cholesky factorization code for high performance on a new platform. We have seen that the optimal parameters can be different for each problem type and platform. Thus, we need to adapt the code in search for performance.

Using a combination of techniques (windows within dense data submatrices, intra-block and hypermatrix oriented supernode amalgamation) can be quite effective. Our code outperforms some state-of-art codes when working on some matrices taken from Linear Programming problems.

We are planning to experiment with a row-wise storage of data submatrices. We have found many matrices for which routine WIN_1DR was used to do more floating point operations than routine WIN_1DC. We think that using a row-wise storage we could possibly improve the performance of the WIN_1DR routine, and consequently, improve the factorization speed. In addition, we would have data submatrices accessed with stride which can potentially speed up the code. We can achieve this by operating on an upper triangular matrix (U) rather than the lower triangle (L). In addition, we would like to switch to a (faster) dense factorization for the last stages of the factorization.

We would also like to improve our code by reducing the operations on zeros and avoiding the operation on very small matrices. We will try to extend our code so that it can work with supernodes in addition to dense matrices in the last level of the hypermatrix (data submatrix level).

Chapter 4

Operation on dense matrices: Nonlinear array layouts

In this chapter we present the study done on nonlinear data layouts. First, we review related work. Then we comment the work we have done using two different data structures. We have used the hypermatrix data structure for dense matrix multiplication and Cholesky factorization. As we will see, the results are competitive but show that there is room for improvement. Thus, we have studied another data structure for operation on dense matrices: a square block data layout.

4.1 Introduction

As processor speeds continue to increase relative to memory latencies, locality optimizations get to be a significant performance issue for algorithms operating on large matrices. Data has to be reused in cache as effectively as possible: locality has to be exploited. In low associativity caches conflicts should be avoided or reduced. A considerable amount of research has been conducted towards achieving an effective use of memory hierarchies. In this section we provide an overview of such work.

Conventional techniques

Tiling, also known as *blocking*, is a loop transformation which combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together to exploit data locality [101, 172, 30]. An effective usage of the cache also requires avoiding self-interference conflict misses within each tile and reducing cross-interferences. Tile shapes and sizes can be adjusted to eliminate both capacity and self-interference misses and reduce cross-reference misses. Several papers have addressed these issues. The most representative ones are: [52] selects the largest number of non-conflicting columns; [115] select the largest non-conflicting square; [37] select rectangular non-conflicting tile sizes.

An alternative method for avoiding conflict and TLB misses is to *precopy* tiles to a buffer and modify code to use data directly from the buffer [115]. It is possible to copy sections of different arrays to buffers. If buffers are adjacent, then cross-interference misses are avoided. However, precopies are performed at run time and can penalize performance[37]. Some authors have investigated selective copying [163] and made efforts to optimize the routines used to do the precopies[67].

Padding is another method which attempts to reduce conflict and TLB misses by modifying the program's data layout. It is a data alignment technique that involves the insertion of dummy elements into a data structure for improving cache performance. In [16, 152] the authors examine two compile-time data-layout transformations for eliminating conflict misses, concentrating on misses occurring on every loop iteration. Inter-variable padding adjusts variable base addresses, while intra-variable padding modifies array dimension sizes. In other occasions authors combine padding and tiling [151, 143]. Application of this kind of techniques in complex scientific programs has been studied in [82, 153].

Alternative matrix representations

A matrix representation is a method used by a computer language to store matrices of more than one dimension in memory. Fortran and C use different schemes. Fortran uses "Column Major", in which all the elements for a given column are stored contiguously in memory. C uses "Row Major", which stores all the elements for a given row contiguously in memory. These two schemes are considered canonical storage.

The default column/row-major order used by programming languages such as Fortran and C limits locality to a single dimension. Alternative storage formats have been proposed to address the issue of locality. A submatrix storage was proposed in [128] with the purpose of minimizing the page faulting occurring in a paged memory system. The authors partition matrices into square submatrices, keeping one submatrix per page, obtaining orders of magnitude improvement in the number of page faults for several common matrix operations.

More recently, with the advent of parallel computing platforms there has been interest in new array representations. Extensive work has been done on storage optimization of arrays [125, 75, 110, 108]. Lately, Hierarchical Tiled Arrays (HTAs) [4] have been proposed as a generalization of the recursively blocked arrays arising in some linear algebra algorithms. Their purpose is to facilitate parallel programming and programming for locality. Parallel numerical algorithms have been implemented using HTAs [25, 26].

The use of data layouts with recursive patterns is known in parallel computing for improving both load balance and locality and has been applied in several application domains [22, 23, 147, 1]. These data layouts have been described in terms of quadtrees[1] [54, 157] or in terms of space-filling curves (SFC) [156]. The quadtree is an aid to conceptualizing the layout, but none of its internal nodes need to be physically represented in memory when an SFC is used. An SFC is

[1]The Quadtree is a tree data structure in which each internal node has up to four children. Quadtrees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants. The quadtree data structure was created by R. Finkel and J.L. Bentley.

a way of mapping the multi-dimensional space into the 1-D space. It acts as a thread that passes through every cell element in the D-dimensional space so that every cell is visited exactly once. There are numerous kinds of space-filling curves. The difference between such curves is in their way of mapping to the 1-D space. Some of them have been applied more extensively due to locality properties which make them attractive [130]. Peano-Hilbert [146, 98] and Morton [131] (or Z [141]) order have often been used in computer science as locality preserving hashing functions.[2] A mapping is called a locality-preserving mapping in the sense that, if two points are near to each other in the D-dimensional space, then they will be near to each other in the 1-D space.

Serial dense codes using nonlinear array layouts

In the last ten years there have been several studies on the application of nonlinear array layouts in uniprocessor environments. Recursivity has been introduced into linear algebra codes. Block recursive codes for dense linear algebra computations appear to be well-suited for execution on machines with deep memory hierarchies because they are effectively blocked for all levels of the hierarchy [77, 164]. Unfortunately, block recursive algorithms do not interact well with the TLB [3]. This has led to the irruption of new storage formats [76, 8, 7, 80, 5, 6] which have been used both in serial and parallel implementations.

Some studies have focused on the use of quadtrees or SFCs for serial dense codes. In [58], a recursive matrix multiplication with quadtrees using a "two-miss" algorithm was presented[3]. They carried the recursion down to the level of single array elements which causes a dramatic loss of performance. Later [171], the same authors improved performance by stopping recursion at 8×8 blocks.

In [34] the authors have experimented with five different SFCs (U, X, Z, Gray and Hilbert) on the matrix multiplication algorithm. The performance reported was similar for all five. Morton (Z) order has relative simplicity in calculating block addresses compared to the other orderings and is often the order of choice. In spite of this relative simplicity compared to other layouts, calculation of addresses in Morton layout is expensive. There are several indexing techniques which differ in their structure, but which all induce Morton order: Morton, level-order, and Ahnentafel indexing. These indexing schemes require bit manipulation unless a lookup table is precomputed [33]. Bit masks can be used when dimensions are powers of two [14]. However, this requires padding.

In [33] the authors use two nonlinear formats: Morton-order and a block data layout which they refer to as 4D. In 4D data is stored by submatrices which are in turn stored by rows or columns. According to their results, the performance of these two nonlinear data layouts is similar and both outperform that of codes based on column-major layouts by factors of 1.1–2.5 depending on the application.

Recursive patterns have also been applied to linear algebra codes to obtain cache oblivious algorithms. Such algorithms are designed to inherently benefit from any underlying hierarchy of caches, but do not need to know about the

[2] In computer science, a locality preserving hashing is a hash function f that maps a point in a multidimensional coordinate space to a scalar value, such that if we have three points A, B and C such that $distance(A, B) < distance(B, C)$, then $|f(A) - f(B)| < |f(B) - f(C)|$.

[3] The *two-miss* algorithm ensures that one operand is reused between adjacent calls.

exact structure of the cache. Examples of this can be found for Morton [170, 84] and Peano order [18].

A variant of the Z-Morton layout [131, 34, 170] is the Recursive Block Row (RBR) format used in [76, 51]. The latter avoids the restriction of having block sizes which are a power of two. A recursive block ordering is determined by dividing the largest dimension of the rectangular matrix. When there is a tie, the row dimension is divided. A variant of this format called Recursive Block Column (RBC) divides the matrix in a similar way, but dividing the column dimension when there is a tie. RBC corresponds to a variation of the reflected-N-Morton space filling order [156].

Another advantage of nonlinear layouts can be found in implementations of Strassen's algorithm. This algorithm has reduced arithmetic complexity w.r.t the standard matrix multiplication algorithm. However, it has poor algorithmic data locality. In [165], the authors report excellent results for their implementation of the Winograd variant of Strassen's algorithm based on their hierarchical storage format.

Both the quadtree representation used in [58] and the codes presented in [34] require padding to handle arbitrary sized matrices. The hierarchical storage format presented in [165] overcomes this requirement. Their hierarchical matrix storage incorporates Morton order for arbitrary-sized matrices without any additional memory or computation on zero padding.

Tiling can also be applied to nonlinear data layouts. In [145] the authors show that improved cache and TLB performance can be achieved when tiling is applied to both Block Data Layout and Morton layout. In their experiments matrix multiplication with an iterative code using BDL was often faster than a recursive code using Morton layout. As we will comment below, our results agree with this: our iterative tiled algorithm working on BDL outperforms the recursive code operating on hypermatrices. Authors have also investigated on tile size selection for nonlinear array layouts [165, 145, 15] and have come to similar conclusions to the case of canonical storage: blocks should target the level 1 cache.

Different authors refer to a given data layout using different names. A data layout where matrices are stored as submatrices which are in turn stored by columns has been named as Submatrix storage in [128], BC in [76, 51], SB in [80, 78, 81, 79], 4D in [33], and BDL in [145], TDL in [96]. In this document we will refer to such data layout as SB. This name reflects the square nature of the submatrices and is one used most extensively in the literature.

In the next sections we present our work using two nonlinear array layouts.

4.2 A bottom-up approach

We have studied two data structures for dense matrix computations: a Hypermatrix data structure [61] and a Square Block Format [80]. In both cases we drive the creation of the structure from the bottom: the inner kernel fixes the size of the data submatrices. Then the rest of the data structure is produced in conformance. We do this because the performance of the inner kernel has a dramatic influence in the overall performance of the algorithm. Thus, our first

priority is using the best inner kernel at hand. Afterwards, we can adapt the rest of the data structure (in case hypermatrices are used) and/or the computations.

4.2.1 Inner kernel based on our Small Matrix Library (SML)

As we mentioned in section 2.5 we have extended our SML with routines which work with sizes larger than the ones used for the sparse codes. We use the matrix multiplication routine within our SML as the inner kernel of our general matrix multiplication codes.

4.3 Hypermatrix storage

4.3.1 Exploiting the memory hierarchy

Number of levels and dimension of each level

We use the hypermatrix data structure to adapt our codes to the underlying memory hierarchy. Our code can be parameterized with the number of pointer levels and block sizes mapped by each level. For the dense codes we follow the next approach: we choose the data submatrix block size according to the results obtained while creating our SML matrix multiplication routine. The one providing the best performance is taken. As seen in section 2.5 we do this even when the matrix size is too large to fit in the L1 cache. Then, for the upper levels we choose multiples of the lower levels close to the value $\sqrt{C/2}$, where C is the cache size in double words. Such values are known to reduce cache conflicts due to accesses to the other matrices [115]. We found that, for the machines studied, we only needed two levels of pointers for dense in-core operations. For larger matrices, which required to go out-of-core, or in machines with more levels of caches more pointer levels could be used.

Figure 4.1 shows the performance of our hypermatrix multiplication routine on an Itanium2 processor for several matrix dimensions. We have tried several dimensions for the second level of pointers. Values 368 and 460, close to the value $\sqrt{C/2}$, were the best. Using large blocks resulted in performance loss. We tried using a third level with size 736 when the second one has size 368. It didn't produce any benefit since the SML routine was already using efficiently the L2 cache, and the second level of pointers was enough to use the L3 cache adequately. On the Alpha 21264A the size used for data submatrices was 48×48. Then, a second level of pointers in the hypermatrix maps blocks of dimension 480. On the R10000 a data submatrix block size of 60×60 allows for a matrix to be almost permanently in L1 cache. Some conflicts might arise but they should be scarce. A second higher level of 8×8 pointers maps blocks of 480×480 data. This is adequate both for the TLB and second level cache.

Orthogonal blocks

In [135] a class of Multilevel Orthogonal Block forms was presented. In that class each level is orthogonal to the previous: they are constructed so that the directions of the blocks of adjacent levels are different. These algorithms exploit the data locality in linear algebra operations when executed in machines

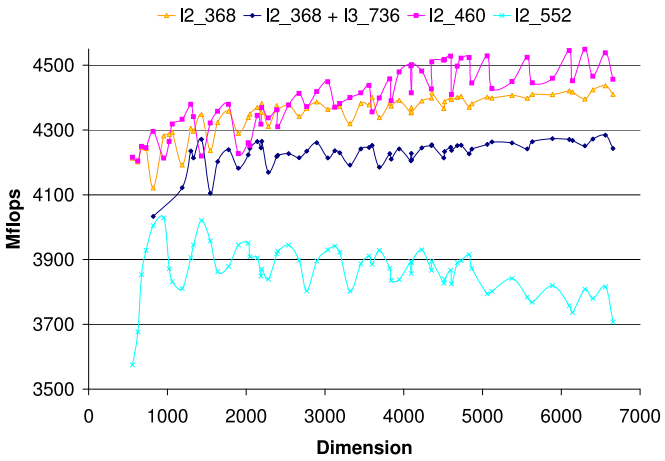


Figure 4.1: Performance of dense matrix multiplication for several hypermatrix configurations on an Intel Itanium 2 processor.

with several levels in the memory hierarchy. Figure 4.2 shows graphically the directions followed by two possible Multilevel Orthogonal Block (MOB) forms.

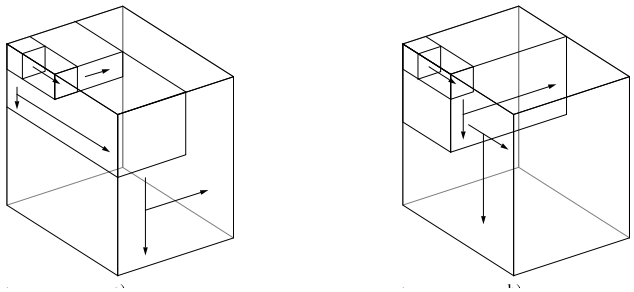


Figure 4.2: Two examples of Multilevel Orthogonal Block forms

We have implemented Multilevel Orthogonal Blocks [135] for the different levels in the hypermatrix structure. Actually, we can generate all combinations of loop orders with a code generator. Figures 4.3 and 4.4 show the performance obtained on a matrix multiplication performed on matrices of size 4507 on Itanium2 and Alpha 21264A processors for all combinations of loop orders for two levels of pointers in a hypermatrix. All bars to the right of the dashed line correspond to orthogonal forms. Although we eventually use only two pointer levels, for hypermatrix multiplication there is an improvement in the performance obtained when the upper level is orthogonal to the lower. In this way the upper level cache is properly used. The performance improvement is modest, but results were always better than those corresponding to non-orthogonal block forms.

Results

We have compared our dense Cholesky factorization and hypermatrix multiplication with vendor and ATLAS [167] DPOTRF and DGEMM routines. On

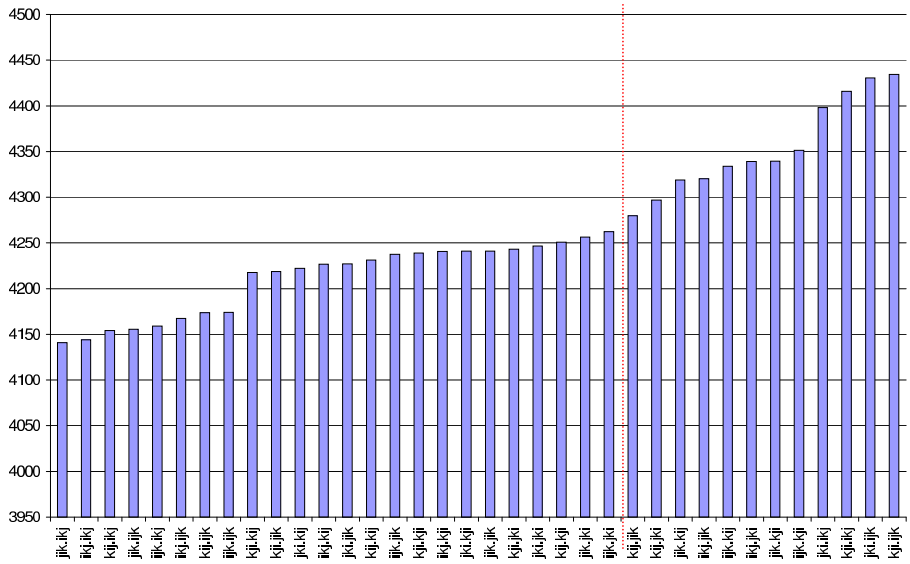


Figure 4.3: Performance of HM dense matrix multiplication for several loop orders on an Intel Itanium 2.

the R10000 our code outperformed both the vendor and ATLAS DPOTRF and DGEMM routines. The graph on the left part of figure 4.5 shows the performance obtained on this platform for a dense Cholesky factorization. The graph compares the results obtained by our code (labeled as HM) with those obtained by routine DPOTRF in the vendor library. Both when upper (U) or lower (L) matrices were input to this routine its performance was worse than that of our code. We also tried the matrix multiplication operation $C = C - A * B^T$ since this is the one which takes about 90% of Cholesky factorization. The results can be seen in the right part of figure 4.5. Our code outperformed the DGEMM matrix multiplication routine in both the vendor and ATLAS libraries. We must note however, that ATLAS was not able to finish its installation process on this platform. Thus, we used a precompiled version of this library which corresponds to an old release of ATLAS.

On an Alpha 21264A ATLAS installation phase lets the user choose whether to install a hand made code specially designed for this platform (GotoBLAS). In both cases, on this system, ATLAS outperforms our matrix multiplication code. One reason for this can be observed in figure 2.3. The peak performance of the matrix multiplication routine in our SML was far from the theoretical peak performance on this machine. However, we obtain the same performance as DPOTRF for large matrices (figure 4.6). On the Itanium2 our performance got close to ATLAS' both for DGEMM and DPOTRF. It was similar to ATLAS for large matrices (figure 4.7). We must note that, in spite of its name, the ATLAS project is often based on matrix multiplication kernels written in assembly code by hand.

This work was presented in [92].

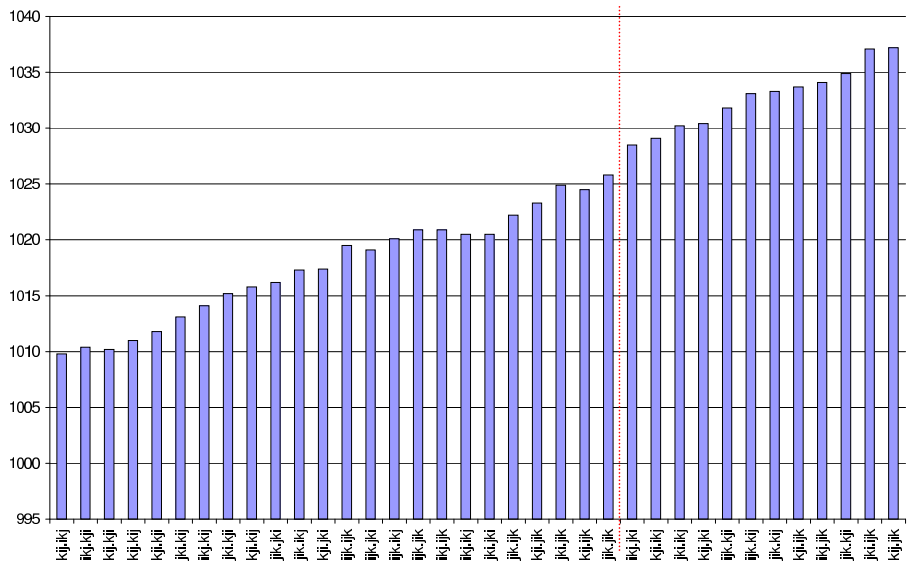


Figure 4.4: Performance of HM dense matrix multiplication for several loop orders on an Alpha 21264A processor.

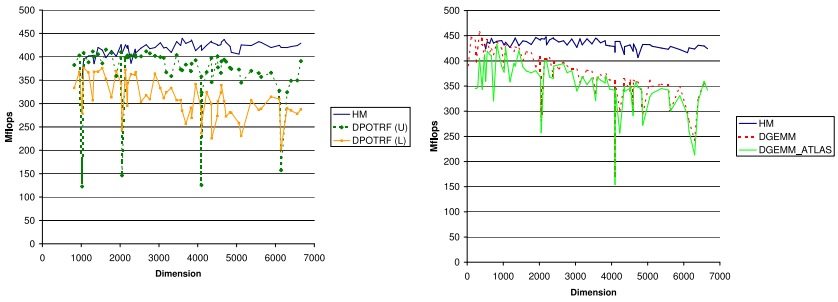


Figure 4.5: Performance of dense Cholesky factorization and matrix multiplication using hypermatrices on a MIPS R10000 processor.

Conclusions

A hypermatrix data structure can be used to adapt the code to the underlying memory hierarchy. For the machines studied and working on dense matrices in-core, two levels of pointers were enough. Going out-of-core or working on machines with more levels of cache memory could benefit from the extension of this scheme to a larger number of levels in the hypermatrix. The use of Multilevel Orthogonal Block forms was always beneficial on the platforms used.

4.3.2 Parallel dense HM multiplication using OpenMP

Introduction

We have used a Hypermatrix data structure [61, 138] in sequential linear algebra codes. We could obtain efficient implementations in both sparse and dense codes. Now, we are interested in the parallelization of our dense codes. This

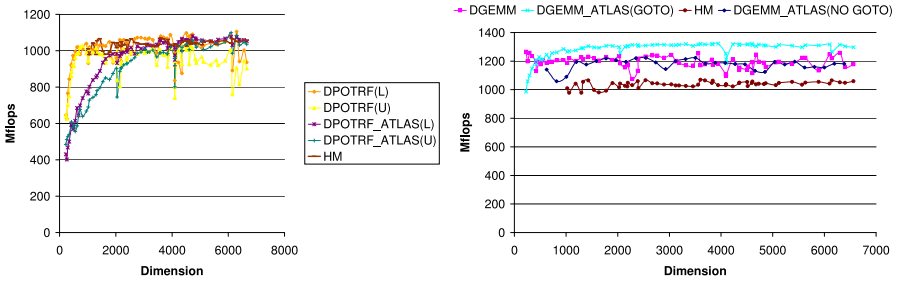


Figure 4.6: Performance of dense Cholesky factorization and matrix multiplication using hypermatrices on an Alpha 21264A processor.

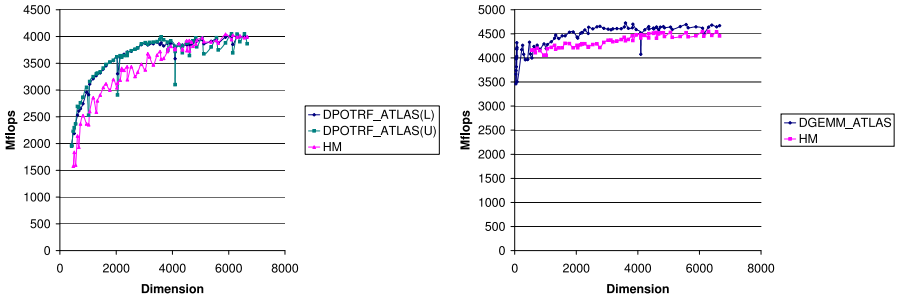


Figure 4.7: Performance of dense Cholesky factorization and matrix multiplication using hypermatrices on an Intel Itanium 2 processor.

data structure, however, presents difficulties when work has to be distributed amongst several processors. Namely, the difficulty to balance the load evenly. In this section we present the work we have done to produce a hypermatrix multiplication based on OpenMP directives. This work was presented in [95]. We wanted to know whether some of the features available in OpenMP could surmount the intrinsic difficulties of parallel codes based on the Hypermatrix data structure. We have chosen matrix multiplication because it is highly parallelizable. Also, it is a very important operation since it appears as a basic kernel in many scientific applications. For this reason it has been studied extensively [167, 36, 34].

OpenMP

OpenMP [139] provides a set of directives and environment variables to express and control parallelism in the execution of a program. The user can choose the scheduling algorithm. When a *static* scheduling algorithm is used, the distribution of iterations to threads is done before the execution of any of them. When a *dynamic* scheduling algorithm is used, the next piece of work for a thread is assigned when it is needed. It is taken from the remaining operations due. There is a default value for the number of iterations assigned to each processor which can be changed by the user explicitly. We refer to the *chunk* size. The default for the static scheduling is to split the work in as many parts as the number of threads defined. The default for the dynamic scheduling is to take one iteration each time.

Several nested loops can be parallelized. OpenMP permits this fact with a feature known as *nested parallelism*. When nested parallelism is activated, parallel constructs can be used within other parallel constructs.

In this work we have used OpenMP for the parallelization of a matrix multiplication code based on the hypermatrix data structure.

Hypermatrix data structure

Now, we are interested in the efficient execution on multiprocessor machines. The hypermatrix data structure, however, presents some difficulties when parallel code is developed. Namely, the partitioning of the matrix is done when the data structure is set. Each pointer in the upper pointer matrix level maps a part of the matrix. If the dimension of such matrix is not a multiple of the number of processors used then the load is not distributed evenly amongst them.

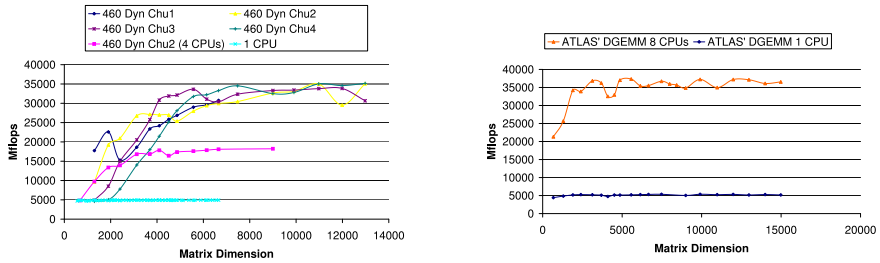
We have started with the study of the hypermatrix multiplication operation, which is very regular and has a high potential for parallelism. We have added OpenMP directives to a few loops and experimented with several features available with OpenMP in the Intel Fortran Compiler: scheduling algorithms, chunk sizes and nested parallelism. We anticipate than none of these features was completely successful for the efficient parallelization of our code.

Parallel dense hypermatrix multiplication using OpenMP

The target machine was a 8-way SMP with Intel Itanium2 processors running at 1.5 GHz. The theoretical peak of this machine is 48 Gflops. The Itanium2 has three levels of cache. In the first level it has separate instruction and data caches with 16 Kbytes each. Then, it also has a 256 Kbytes L2 cache and an off-chip L3 cache with possible sizes ranging from 1.5 up to 9 MB.

We have experimented with four and eight processors. In this section we will discuss the results obtained. Our preliminary study on four CPUs provides a speed-up of 3.7 for medium to large matrices. The best combination was that where the two outermost loops were parallelized using nested parallelism and a dynamic scheduling algorithm was used where the chunk size equaled 2. Figure 4.8a shows the performance of our hypermatrix multiplication code on four processors for the $C = C - A * B^T$ operation for both the sequential and parallel versions of our code. We have used an upper block size of size 460×460, i.e. each upper level pointer maps a block of such size. We then tried the same approach on eight processors. The same figure 4.8a shows the performance obtained on eight processors with a dynamic scheduling strategy, with the two outermost loops parallelized using nested parallelism. Several chunk sizes were used.

We observe that for small matrix dimensions small chunk sizes provide better results. However, as the matrix gets bigger, larger chunk values can be more effective. This is due to the reduction in the overhead which occurs when one thread searches for new work. Giving a thread more work at once reduces the number of times this needs to be done. Also, the memory hierarchy can be better used since contiguous blocks corresponding to consecutive iterations can be reused in the cache. It is important to note that a certain chunk value is effective only when it keeps a good load balancing. Since the loops we have parallelized are the outer loops, they correspond to the upper level pointer matrix. The



a) b)

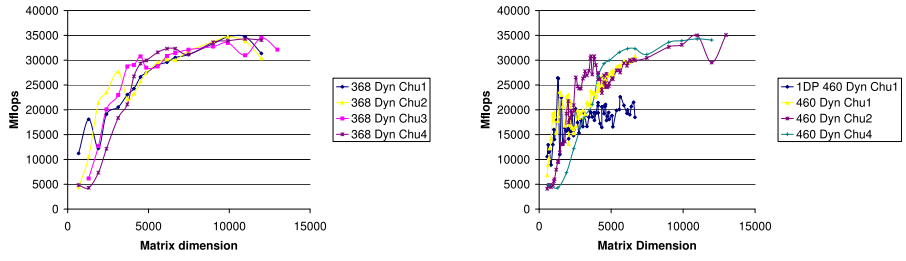
Figure 4.8: **a)** Two parallel loops with dynamic scheduling and several chunk sizes on 8, 4 and 1 processors. **b)** Performance of ATLAS' DGEMM.

chunk size times the number of processors should divide the upper level matrix dimension evenly. Otherwise, load imbalance occurs and the performance drops.

We wanted to compare our results to those of ATLAS [167]. Figure 4.8b shows the performance of the sequential and parallel (on eight processors) versions of ATLAS matrix multiplication routine DGEMM. Their code, starting with the sequential version, outperforms ours. We must note, however, that on this machine ATLAS uses a hand-tuned kernel. The interesting point here comes from the fact that their code achieves high speed-ups sooner than our code. For some large matrix dimensions the speed-up we obtain is similar to theirs (around 7.0). However, for smaller matrices our speed-up is considerably lower. This is due to the load imbalance mentioned above. We have revisited our code and tried several variants aiming to improve its performance, specially when working on smaller matrices.

Reducing the block size

By default we have been using an upper block size of 460×460, i.e. each upper level pointer maps a block of such size. However, we have also reduced the size of the block to 368×368. Figure 4.9a shows the performance obtained with dynamic scheduling and nested parallelism for this block size. Results are similar to those obtained with our default block size of 460×460.



a) b)

Figure 4.9: **a)** Two parallel loops with dynamic scheduling: smaller blocks. **b)** Three loops parallelized (one in the level of pointers to data).

For the upper levels we have also tried other multiples of the lower levels close to the value $\sqrt{C/2}$, where C is the cache size [115]. Figure 4.11b shows the performance obtained with several sizes. To simplify the comparison, the maximum value obtained for all chunk sizes for a given block size are presented.

Results are similar for all of them. We must note that the smaller block size 276×276 provides the worst performance for larger matrices. This size does not use the memory hierarchy so effectively. Also, there is more overhead in the parallelization.

Parallel loop in level of pointers to data

We have tried another code which parallelizes a third loop in addition to the outermost two loops. This loop is the outermost loop in the level of pointers to data. Figure 4.9b compares its results to those shown in figure 4.8a. This code only gets better performance for a few small matrices. For larger matrices, this code does not offer any advantages. The granularity of this third loop is too small and the overhead of the parallelization outweighs any possible advantages.

Static scheduling

Figure 4.10a shows the results obtained with a static scheduling. Results with and without nested parallelism are shown. When only the outermost loop is parallelized we get a saw shape curve. The peaks correspond to sizes which get a perfect partitioning of the hypermatrix, i.e. with a number of pointers in the upper matrix which is a multiple of the number of processors. The use of nested parallelism introduces some overhead. However, it improves the performance for matrix sizes which are not multiples of the number of CPUs. The performance obtained in both cases is in general worse than that presented in figure 4.8.

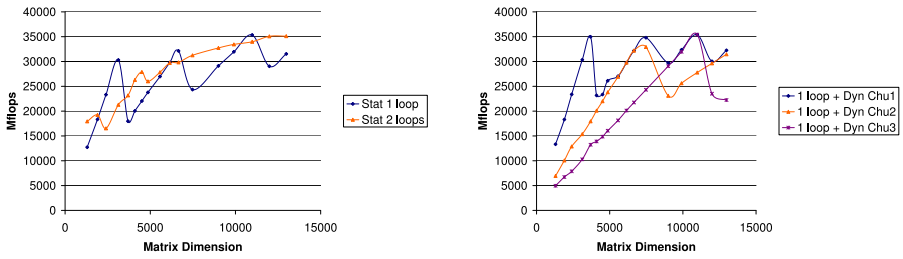


Figure 4.10: **a)** One and two parallel loops with static scheduling. **b)** Parallel outer loop with dynamic scheduling and several chunk sizes.

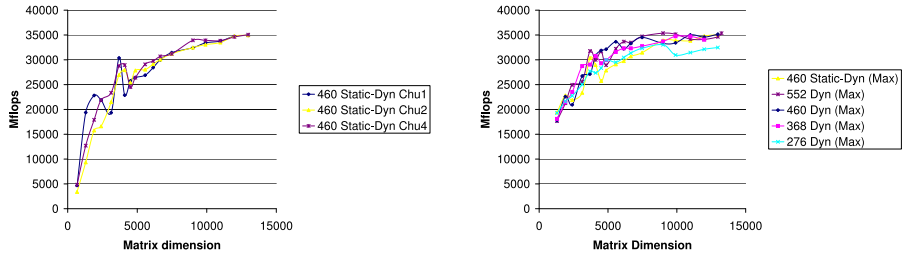
Dynamic scheduling with only 1 Parallel loop

Figure 4.10b shows the results obtained when only the outermost loop is parallelized. A dynamic scheduling is used in this case. Again, we get a saw shaped curve. It is quite obvious that larger chunk sizes suffer from load imbalance more often.

Combining Static and Dynamic scheduling algorithms

We have scheduled the outermost loop using a static scheduling and the second outermost loop using a dynamic scheduling. Figure 4.11a shows the performance obtained. The performance obtained is similar to the one obtained when

both loops are scheduled using a dynamic scheduling algorithm as shown in figure 4.11b.



a) b)

Figure 4.11: **a)** Static scheduling of outermost loop and dynamic scheduling of next inner loop. **b)** Maximum values obtained for each block size and scheduling algorithm.

Perfect matrix partitioning

Figure 4.12 shows the results obtained when the matrix has been partitioned in a number of parts which is multiple of the number of threads. All partitions have the same size: each upper level pointer maps blocks of 460×460. Thus, a dimension of 3680 produces a hypermatrix with eight pointers in the upper level. The number of pointers in the upper level corresponding to the other three matrix dimensions in the figure are 16, 24 and 32 respectively. All of them are examples where the load can be easily balanced amongst the processors.

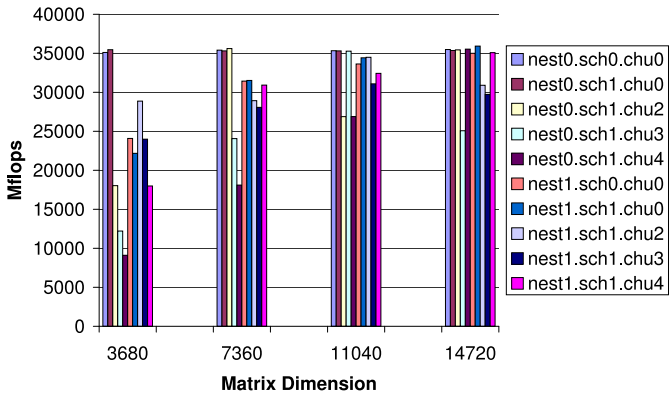


Figure 4.12: Experiments with different OpenMP features when the hypermatrix is partitioned for perfect load balancing.

These results allow us to study the overhead of each strategy. We use *nest* to identify the use of nested parallelism. A value of 0 means nested parallelism is not allowed while a value of 1 means the opposite. Label *sch* corresponds to the scheduling algorithm used. A value of 0 denotes static scheduling. A value of 1 is used to indicate dynamic scheduling. We use *chu* to specify the chunk size. A value of 0 is used to signify the default value for a given scheduling strategy.

On these perfectly partitioned hypermatrices we can observe that, when the matrices are small, the only way to get good speed-ups is via simple strategies: parallelizing only the outermost loop with either static or dynamic scheduling. As the matrices get large there are more strategies which provide good speed-ups. However, in any case it is important to use a chunk size which allows for a good load balancing. The result of dividing the dimension of the upper level pointer matrix by the number of threads must be a multiple of the chunk size.

In a few occasions, a chunk size larger than the default for the dynamic scheduling strategy (which defaults to 1) can improve slightly the performance of our matrix multiplication. This is due to the reduction of the overhead which occurs when one thread takes several iterations at once instead of taking one iteration each time. Also, better use of the memory hierarchy results when one thread reckons several contiguous blocks corresponding to consecutive iterations.

Nested parallelism is not really effective in such situations. It cannot provide any advantages. Instead, it introduces an additional overhead with the creation of the inner parallel construct.

Conclusions

We conclude that the best way to parallelize our application is by means of an adequate partitioning of the matrix. If this is possible, a simple scheduling strategy where just the outermost loop is parallelized turns out to be the best solution. Both static and dynamic scheduling algorithms work well and perform in a similar manner.

When data cannot be partitioned adequately we can take advantage of nested parallelism. Despite its overhead, it offers the advantage of being able to open new parallel sections which can employ otherwise idle processors. The resulting performance curves are smoother than the saw shaped curves which result from those cases where only the outer loop was parallelized.

We have conducted experiments with eight processors and found some load imbalance in those cases where the dimension of the matrix in the upper pointer level is low and is not multiple of the number of processors used. Thus, smaller matrices suffer from load imbalance as the number of processors grow. This can limit the effectivity of parallel codes based on the hypermatrix scheme.

Consequently, we plan to replace the hypermatrix data structure in our algorithms which deal with dense matrices. In the next section we use a plain storage of the data submatrices which can be accessed with a simple indexing scheme. In this way we can still use our routines which deal with small submatrices and, at the same time, we will be able to split the work amongst all processors more effectively.

4.3.3 Data submatrix storage: Column versus row storage and alignment

We have been using a column-wise storage for data submatrices since this is the standard Fortran order. Now, we want to see whether a row-wise storage could benefit our algorithms. On a Cholesky factorization about 90% of the time is spent in a matrix multiplication such as $C = C - A * B^T$. Having the data submatrices stored row-wise could result in streaming of all 3 matrices

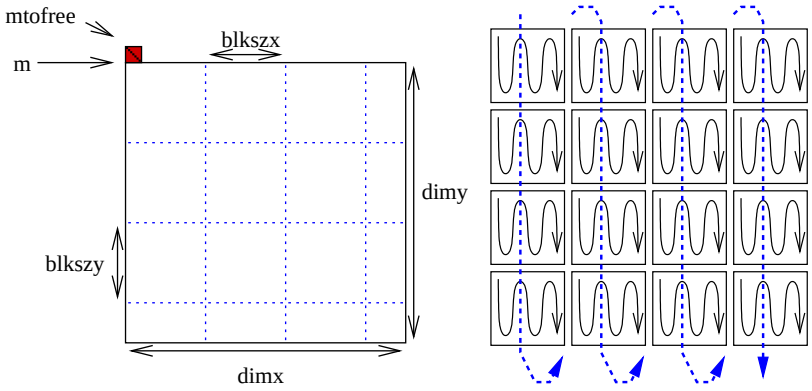


Figure 4.13: Square Block Format: matrices aligned and stored by submatrices.

(accessing matrices with stride 1) for this operation. This would improve spatial locality and data could be brought from upper levels of memory faster.

Note that $C = C - A * B^T$ using row-wise storage is the same as $C = C - A^T * B$ using column-wise storage. For this reason we have experimented with the hypermatrix multiplication with $C = C - A^T * B$. We found that a performance improvement between 5 and 9% was obtained compared to the hypermatrix multiplication where matrix A is not transposed while B is transposed.

4.4 Square Block Format (SB)

The overhead of a dense code based on hypermatrices due to the recursivity and indexing, together with the difficulties to produce efficient parallel codes based on this data structure, has led us to experiment with a different data structure. We use a simple Square Block Format (*SB*) [80]. It corresponds to a 2D data layout of submatrices stored in column-major order (see figure 4.13). The shaded area represents padding introduced to force data alignment.

Using this data structure we were able to improve the performance of our matrix multiplication code, obtaining very competitive results. This work was published in [96]. Our code implements tiling. We use a code generator to create different loop orders. Next, we present the results obtained with the best loop order found.

Results

We present results for matrix multiplication on three platforms. The matrix multiplication used is $C = C - A^T \times B$. Each of the following figures shows the results of DGEMM in ATLAS, Goto or the vendor BLAS, and SB using our SML. Goto BLAS [67] are known to obtain excellent performance. They are coded in assembler and targeted to each particular platform. The dashed line at the top of each plot shows the theoretical peak performance of the processor. Some plots show the performance obtained with the dense codes based on the hypermatrix (HM) scheme. We observe that SB outperforms HM.

For the Intel machines (figure 4.14) we have included the Mflops obtained with a version of the ATLAS library where the hand-made codes were not

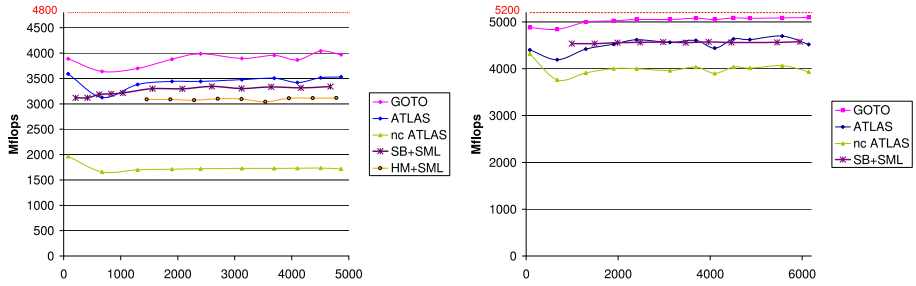


Figure 4.14: Performance of dense matrix multiplication on an Intel Pentium 4 Xeon (left) and an Intel Itanium 2 processor (right).

enabled at ATLAS installation time. We refer to this code in the graphs as 'nc ATLAS'. We can observe that in both cases ATLAS performance drops heavily. SB with SML kernels obtain performance close to that of ATLAS on the Pentium 4 Xeon, similar to ATLAS on the Itanium2, and better than ATLAS on the Power4. For the latter we show the Mflops obtained by the vendor DGEMM routine which outperform both ATLAS and SB (figure 4.15). We can see that even highly optimized routines provided by the vendor can fail under certain circumstances. For instance, some large leading dimensions can be particularly harmful and produce lots of TLB misses if data is not precopied. At the same time, data precopying must be performed selectively due to the overhead incurred at execution time [163]. These problems can be avoided using nonlinear array layouts.

Results for SB assume matrices already stored in block major format. Although new matrix storage formats have been proposed [80, 7, 34, 165] the matrix will probably need to be transformed from column major order into block major order. We have measured the time necessary to create the three matrices used in a matrix multiplication. Taking that into account, the performance of SB drops by about 10% for small matrices, and as low as 1% for the largest matrices tested. The reason for this is that the cost of this transformation is $O(N^2)$ while for the multiplication the cost is $O(N^3)$.

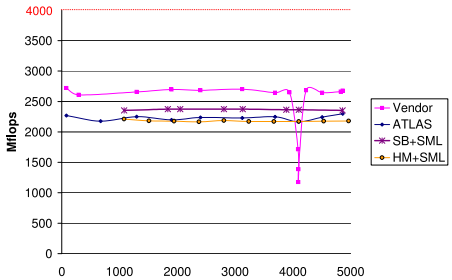


Figure 4.15: Performance of dense matrix multiplication on a Power4 processor.

4.5 Final considerations

Column versus row storage and alignment

We have seen in previous sections that the performance obtained with the operation $C = C - A^T * B$ is superior to that of $C = C - A * B^T$. For this reason we plan to implement a Cholesky factorization which uses the former operation. This can be achieved by performing the Cholesky factorization on an upper triangular matrix U instead of a lower triangular matrix L.

Inner kernel based on hand optimized codes

Both the hypermatrix and SB approaches could also benefit from codes written in assembly language by hand. In order to improve our performance on some systems, we would like to use an ad-hoc matrix multiplication kernel and compare the results. This is a complex task and will be left as future work. Many researchers have based their codes on the use of optimized inner kernels which match the algorithm and the architecture [2, 67, 78, 51, 168]. We believe that BLAS implementations should provide direct entry points to the inner kernels. In this way, other codes could benefit from very efficient inner kernels created by an expert by hand. We agree with [165] that a standard kernel interface different from the BLAS should be added.

Recursive+HM vs Iterative+SB: Conclusions

The results obtained with an iterative code working on a simple Square Block Format outperform the recursive code which uses a hypermatrix. Our results agree with those presented in [145]. We would like to implement a dense Cholesky factorization using an iterative approach and a Square Blocked Lower (or Upper) Packed Format [80]. We plan to do it straightaway.

Chapter 5

Application to other fields: Nearest Neighbor Classification

5.1 Introduction

The Nearest Neighbor (NN) classification procedure is a popular technique in pattern recognition, speech recognition, multitarget tracking, medical diagnosis tools, etc. A major concern in its implementation is the immense computational load required in practical problem environments. Other important issues are the amount of storage required and the data access time.

In this chapter, we address these issues by using techniques widely used in linear algebra codes: use floating-point operations instead of integer arithmetic, apply tiling, loop unrolling or software pipelining. We show that a simple code can be very efficient on commodity processors and can sometimes outperform complex codes which can be more difficult to implement efficiently. This work was presented in [93].

5.1.1 Computer resources

Computer architecture has evolved very quickly in the last decades with important improvements in many areas. We will center our attention in two aspects which are essential to the execution of programs: processor and memory. Current microprocessors have very fast clocks and multiple functional units within the processor. Potentially, some processors can execute billions of operations per second. However, even general purpose processors are usually optimized for scientific computations which require arithmetic with real numbers. This means that, on many processors, a multiplication of floating point numbers will be done much faster than the product of two integers. When multiple functional units are present, several arithmetic operations can be done at the same time. In addition, when those functional units are pipelined, a new arithmetic instruction can be started each cycle, with several operations proceeding through the pipeline. On the other hand, integer arithmetic is usually slower. Also,

evaluating conditionals or taking branches can potentially stall the processor. Therefore, processors can perform certain types of operations faster than others.

Computer memories are getting larger and larger. However, although their access time is getting reduced, it is not progressing as rapidly as the processor speed. This means that, from the processor point of view, the memory is getting slower. For this reason it is important to use the memory hierarchy which is composed of one or more cache memories in addition to the main memory. The smaller they are, the faster they can be accessed. We need to reuse data in the faster levels of the memory hierarchy in order to execute applications quickly. Thus, we need to write "cache conscious" programs.

In this chapter we will show how we can obtain an efficient Nearest Neighbor classification implementation by taking advantage of the machine resources. This is achieved in two complementary steps. First, we use floating-point operations instead of integer operations and avoid conditionals. In this way we use the efficient part of the processor and classify data much faster. Second, we use blocked algorithms to reuse data in the cache memory. This is important in real situations, where data sets are large.

5.1.2 Nearest Neighbor Classification

The classification problem consists in assigning a class from ℓ classes $C_1, C_2, \ldots, C_\ell$ to each of the D_{size} unclassified vectors $\vec{X^j} = [x_1^j, x_2^j, \ldots, x_{V_{size}}^j]$ with length V_{size}, for $j = 1, \ldots, D_{size}$. The NN classification uses a set of vectors $\vec{P^k} = [p_1^k, p_2^k, \ldots, p_{V_{size}}^k]$, for $k = 1, \ldots, P_{size}$, called a set of prototypes, whose class is known. Then, an unclassified vector $\vec{X^j}$ is classified in the same class as $\vec{P^s}$ if $\vec{P^s}$ is the prototype with minimum distance to $\vec{X^j}$, that is

$$d(\vec{X^j}, \vec{P^s}) = \min_{k=1,..,P_{size}} d(\vec{X^j}, \vec{P^k})$$

where, in our examples, the distance function is defined as the square of the Euclidean distance:

$$d(\vec{X^j}, \vec{P^k}) = \sum_{i=1}^{V_{size}} (x_i^j - p_i^k)^2$$

Hence, the distance between the vector $\vec{X^j}$, which is to be classified, and all the vectors $\vec{P^k}$, $k = 1, \ldots, P_{size}$, in the prototype set must be computed. The time needed to classify a set of D_{size} vectors is, consequently, proportional to $(V_{size} \times D_{size} \times P_{size})$.

In the algorithms we use, which are shown below, the set of unclassified vectors is kept in matrix $D(V_{size}, D_{size})$ where $x_i^j = D(i, j)$. The set of prototypes is kept in matrix $P(V_{size}, P_{size})$, where $p_i^k = P(i, k)$. Vector $ClassP(P_{size})$ indicates the class the prototypes belong to, i.e. $ClassP(k) = r$ if $\vec{P^k}$ belongs to class C_r. The result of the classification is stored in vector $ClassD(D_{size})$, where $ClassD(j) = r$ if $\vec{X^j}$ is classified as belonging to class C_r.

Figure 5.1a shows the common brute force algorithm, which we label as *jki* form due to the loop ordering. In this algorithm, all of the V_{size} components of an unclassified vector $\vec{X^j}$, stored in a column of matrix D, are compared to the correspondent components of each vector in the prototype set, stored as

columns of matrix P. Often, the *jki* code has been modified to exit the loop that computes the distance when the current distance exceeds the running minimum found, reducing the number of computations required for classification, while keeping the same accuracy as the brute force algorithm. Figure 5.1b shows the modified *jki* loop, henceforward called *jki_exit* algorithm.

```
MAX = 2 ** 30
DO J = 1, Dsize
 mindis = MAX
 DO K = 1, Psize
  distance = 0
  DO I = 1, Vsize
   sub = D(I,J) - P(I,K)
   distance = distance + sub*sub
  ENDDO
  IF (distance.LT.mindis) THEN
   mindis = distance
   mincla = ClassP(K)
  ENDIF
 ENDDO
 ClassD(J) = mincla
ENDDO
        a)
```

```
MAX = 2 ** 30
DO J = 1, Dsize
 mindis = MAX
 DO 2 K = 1, Psize
 distance = 0
  DO 1 I = 1, Vsize
   sub = D(I,J) - P(I,K)
   distance = distance + sub*sub
   IF (distance.GT.mindis) GO TO 2
1   CONTINUE
   mindis = distance
   mincla = ClassP(K)
2  CONTINUE
  ClassD(J) = mincla
 ENDDO
        b)
```

Figure 5.1: Codes for the **a)** *jki* and **b)** *jki_exit* forms

5.1.3 Data and Computation Diagram

Figure 5.2 shows what we call a Data and Computation Diagram (DCD) for our algorithm. DCDs have been proposed in [133] as a very powerful visual tool in understanding and designing block algorithms. In this diagram, the rectangular parallelepiped represents the iteration space, with the operations in the inside and the data in the faces or in planes parallel to these faces. Thus, each of the 3 orthogonal directions of the Euclidean space is associated with one of the three loops in the code in figure 5.1a. The arrows indicate the order in which the data are accessed and the operations performed in the jki form. To clarify data positions in this DCD, the elements p_{11} and d_{11} are represented in dark. From the code and the DCD it should be apparent that matrix D can be reused in direction k (all iterations of loop k use the same element of D), while matrix P can be reused in direction j. This figure also shows vectors *classP* and *classD*, used to store each vector's classification.

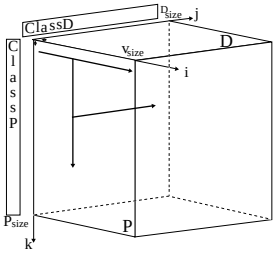


Figure 5.2: Data and Computation Diagram for the jki form of NN classification

5.1.4 Related Work

A major concern in the implementation of the NN technique is the immense computational load associated with it and the large amount of computer memory required when large prototype and data sets exist. These problems have been addressed at length and many alternatives proposed: special-purpose hardware, such as systolic arrays [60] and several approaches have shown to be computationally advantageous over the brute force method:

- Modified metrics as alternative distance measures to the Euclidean distance used in classical NN classifiers [21, 41, 70, 85, 113, 116, 120, 150].
- Selection of a design subset of prototypes from a given set of prototype vectors [41, 46, 83, 116, 150] and generation of prototype reference vectors [45].
- Use of fuzzy logic and Self Organizing Maps [35, 112].

A paper [20] compared RISC-based systems to special purpose architectures for Image Processing and Pattern Recognition (IPPR). They concluded that although a lot of progress had been achieved in RISC technology, low advantages could be obtained for IPPR due to the difficulty of producing efficient code for such machines. In this work, however, we study ways of improving the efficiency of Nearest Neighbor classification on general purpose RISC-based High Performance Workstations since their price can make them cost-effective. Our approach aims to maximize speed maintaining the accuracy of the brute force method by means of an efficient codification of the algorithm using floating-point arithmetic, which increases speed, and a block algorithm, which reduces the number of misses in the cache memory. Such techniques have often been used in numerical applications [10, 133] but never, to our knowledge, to NN classification.

5.1.5 Processor Overview

Our tests have been carried out on two high performance workstations, which incorporate superscalar processors: an HP PA-7150 [142] and a DEC Alpha AXP-21064 processor [38] respectively. Both implement Integer/Floating-Point two-way superscalar operation, i.e. one integer and one floating-point instruction can be issued each cycle. Loads and stores of floating-point registers are treated as integer operations. The CPU can read two consecutive data words (a total of 8 bytes) every cycle from the external data cache. Although floating-point operations can take several CPU cycles, the functional units are pipelined and new operations can be started each cycle.

In order to dispatch operations to the functional units at a high rate, the PA-7150 floating-point instruction set includes instructions which perform a floating-point multiplication operation together with an independent addition or subtraction operation in a single instruction, allowing the floating point unit to dispatch two floating-point operations in a single cycle.

The PA-7150 cache has 256 Kbytes, with a line size of 32 bytes. The number of elements in a line (L) is therefore 32 for byte, 8 for simple and 4 for double precision floating-point data types. According to our experiments, a cache miss produces a penalty of 35 cycles. Consequently, the number of Cycles Per Miss

(CPM) is 35. The AXP-21064 incorporates separate 8 Kbyte on-chip instruction and data caches, and a 1 Mbyte off-chip unified cache. All of them have line size equal to 32 bytes. A first level cache hit has a 3 cycle latency while a miss which hits in the second level cache is available in 11 cycles for the first word, and 18 for the following one.

5.1.6 Performance Metrics

In order to compare different codes that solve the same problem, CPU_{time} is a clear candidate to be used as a metric. However, when problem size is changed from execution to execution it is advisable to use a metric normalized to the size of the problem. For this reason, we introduce Normalized Cycles (NC), which for our classification problem is computed by:

$$NC = \frac{CPU\_time\_in\_cycles}{V_{size} \cdot P_{size} \cdot D_{size}} \qquad (5.1)$$

We model the NC with the following expression:

$$NC = NC(cpu) + NC(mem) \qquad (5.2)$$

where $NC(cpu)$ is the component obtained considering no misses in the memory hierarchy (caches, TLBs, page faults) and $NC(mem)$ represents the penalty cycles due to the misses in the memory system. In the analytical models we develop in this chapter, we do not consider the misses produced by instruction fetches since a separate instruction cache exists and the programs we evaluate are sufficiently small so that no instruction misses occur. We include only misses produced by load accesses to matrices since they constitute almost all the data accesses. Experimental results of the NC for different codes are reported in sections 5.2 and 5.3. All our programs are written in Fortran.

5.2 Algorithm Analysis

In this section we present the results obtained from the execution of several codes using distinct data representations. For certain applications floating-point arithmetic is required. In other cases, however, vector elements can be coded as bytes. We have implemented the NN codes using three different data types for the vector elements: byte, simple and double precision floating-point numbers, which require 1, 4 and 8 bytes of storage space respectively.

For any of the data types used, results depend on problem size. For small problems, the sizes of both the prototype and data to be classified have been defined to be small enough to fit into the cache simultaneously. Data are brought into the cache the first time they are referenced. Subsequent references will hit in cache, since all data are kept in it. Executing a code many times and dividing the execution time by the number of times it is performed hides the misses from the first execution. Therefore, $NC(mem) \approx 0$ for a small problem executed many times and NC is approximately $NC(cpu)$:

$$NC_{SmallProblem} \approx NC(cpu) \qquad (5.3)$$

When the problem size is big enough so that all the data do not fit in the cache at the same time, cache misses arise and data are flushed from the cache between uses. Locality is not well exploited resulting in a poor cache utilization. The $NC(mem)$ component in large problems can be easily estimated by:

$$NC(mem) \approx NC_{LargeProblem} - NC_{SmallProblem} \qquad (5.4)$$

since $NC(cpu)$ are the same for both large and small problems.

To analyze the consequences data size has on performance, two different problem sizes have been tested. Considering the PA-7150's 256 Kbytes data cache, as a small problem we used a database of 200 vectors — 100 for the prototype set and 100 used as data for classification — where each vector has 80 elements. Experiments on a large problem were carried out on a database of 20852 vectors — 10426 for the prototype set and 10426 used as data for classification — each also having 80 components.

In both cases two disparate data initializations have been used which impact on the *jki_exit* performance. First, a distribution obtained from a real application has been used [69]. Figure 5.3a shows the probability distribution of the number of iterations of the inner loop computed before it is exited. The mean value of the number of iterations in this case is $\overline{x} = 22$. Second, a random initialization has been used, where the mean number of iterations performed before leaving the accumulation loop is $\overline{x} = 48$. Figure 5.3b shows this distribution.

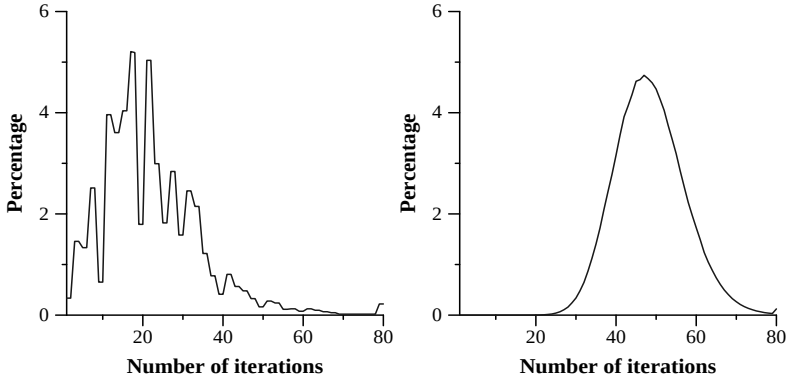


Figure 5.3: Probability distribution of the number of iterations of the inner loop computed for **a)** Real application and **b)** Random initialization

Table 5.1 shows the NC measured for a small and a large problem for the different distributions, algorithms and data types. For the random distribution, with a larger mean value $\overline{x} = 48$, the NC obtained with algorithm *jki_exit* is similar or worse than that of the *jki* algorithm. Clearly, a distribution with a lower mean value, such as the one coming from real data, can take more profit from the *jki_exit* code. Thus, for the rest of the chapter we will only consider the distribution which comes from real data with $\overline{x} = 22$.

Table 5.2 shows the NC measured for a small and a large problem for the different algorithms and data types. These results show that the use of floating-point data is always worthwhile.

5.2.1 The $NC(cpu)$ Component

Despite considerably increasing the memory requirements, using simple (4 bytes) or double (8 bytes) floating-point data is better than a simple byte (assuming a datum can be represented in a single byte). This is so because the PA-RISC 7150 processor can issue one load of a floating-point value together with one floating-point multiplication and one floating-point addition (or subtraction) per cycle.

	Small Problem			Large Problem		
	jki_exit		*jki*	*jki_exit*		*jki*
Data type	$\overline{x} = 22$	$\overline{x} = 48$		$\overline{x} = 22$	$\overline{x} = 48$	
byte	6.1	14.1	15.0	7.1	15.3	16.3
simple float	4.5	10.4	3.6	6.0	12.8	8.4
double float	4.5	10.5	3.6	7.0	15.0	13.3

Table 5.1: NC obtained for different problem sizes and data distributions: HP PA-7150

Table 5.2: NC obtained for different problem sizes

	PA-7150				AXP-21064			
	Small Problem		Large Problem		Small Problem		Large Problem	
Data type	*jki_exit*	*jki*	*jki_exit*	*jki*	*jki_exit*	*jki*	*jki_exit*	*jki*
byte	6.1	15.0	7.1	16.3	24.0	24.0	25.1	25.1
simple float	4.5	3.6	6.0	8.4	22.1	11.9	23.9	19.7
double float	4.5	3.6	7.0	13.3	23.0	20.6	29.2	21.0

On the other hand, when integer arithmetic (byte or integer data type) is used, just one instruction - a load, a multiplication, an addition or a subtraction - can be issued each cycle. Moreover, several data conversions are performed, since all the arithmetic is performed on 32 bit data. Furthermore, the multiplication is computed on the floating-point unit which requires data movements from a general purpose register to a floating-point register, and vice-versa, through memory. From these results we infer that the use of floating-point arithmetic is always beneficial.

In the jki code, the compiler applies software pipelining [114] producing an instruction scheduling which circumvents the problem introduced by data dependencies. When a conditional branch is present in the loop body, as in the jki_exit code, no software pipelining is applied. Consequently, due to the dependencies between instructions plus the extra instructions implementing the "if" statement, the $NC(cpu)$ of the jki_exit becomes larger than that of the jki even for the reduced number of iterations of the jki_exit inner loop; e.g. an average of 22 for jki_exit in our experiments as opposed to 80 for jki.

The same is basically true for the AXP-21064. The overhead of the jki_exit code is so large that this algorithm is outperformed by the simpler jki code. However, since the compiler we had available was not able to perform software pipelining, the instruction scheduling obtained was not as effective as that of the PA-7150. We will thus center our attention on the latter although results of a hand coded software pipelined version developed for the former processor will be presented in section 5.4.

5.2.2 The $NC(mem)$ Component

For a small problem the $NC(mem)$ is negligible. As the problem size grows, the $NC(mem)$ component of the NC increases while the $NC(cpu)$ remains constant (equations (5.3) and (5.4)). In order to predict the number of cache misses a code produces, it is important to take several aspects into consideration. We analyze the jki and jki_exit codes and make comments upon the relevant points.

Spatial Locality

For each inner loop iteration one data element and one prototype element are referenced. It is important to note that for both data structures, accesses to consecutive addresses (column accesses for both D and P) are performed in consecutive iterations of the innermost loop I, exploiting the spatial locality. When the first element in a line which is not present in cache is referenced, a cache miss is produced with a penalty of CPM cycles. However, since the whole line, containing L elements, is brought into the cache, the subsequent $L-1$ accesses to elements in that line are cache hits introducing no extra penalty cycles.

Temporal Locality

The elements of P as well as those of D are reused through the algorithm. Each element of D is reused once for each iteration of the middle loop K, while each element of P is reused for each iteration of the outer loop J.

For each iteration of the middle loop K a new column of P is referenced. However, a fixed column of D is reused for each iteration of loop K and will only be evicted from the cache, due to conflicts with elements of P, every $\frac{C}{E_{size} \cdot V_{size}}$ iterations of loop K, where C is the cache size in bytes and E_{size} is the element size: 1, 4 or 8 for byte, simple or double precision floating-point data respectively. Therefore, we can be reasonably certain that the elements of D are rarely involved in cache misses.

For each iteration of the outermost loop J, all the elements in P are referenced. However, for large matrices, when a new line of P is referenced in iteration $J = j$ a cache miss occurs. Despite having been accessed in iteration $J = j - 1$, the line has already been evicted from cache because accesses to the whole matrix P have been performed and conflicts appeared among its elements due to its large size.

An Analytical Model for $NC(mem)$

Taking into consideration the spatial and temporal locality of the algorithms presented above, we conclude that, for the jki algorithm, a total of $D_{size} \cdot P_{size} \cdot \frac{V_{size}}{L}$ misses occur for the prototype set P. Thus, from equation (5.1) we obtain:

$$NC(mem) \approx \frac{CPM}{L} \qquad (5.5)$$

All the statements asserted above are valid for the jki_exit code with the only difference that matrix P is not completely referenced within an iteration of loop J. Given a mean number of iterations $\overline{x}$ before the inner loop is exited, approximately $P_{size} \cdot V_{size} \cdot \frac{\overline{x}}{V_{size}}$ elements of P will be used. Assuming these data are still too many to fit in cache, $D_{size} \cdot P_{size} \cdot \frac{V_{size}}{L} \cdot \frac{\overline{x}}{V_{size}}$ misses occur. Therefore, for the jki_exit algorithm

$$NC(mem) \approx \frac{\overline{x}}{V_{size}} \cdot \frac{CPM}{L} \qquad (5.6)$$

The leftmost two columns in table 5.3 show the NCs obtained using our theoretical model. For these data, an estimation of the $NC(cpu)$ is obtained

from the NC of the small problem (equation (5.3)) shown in table 5.2. Then, applying the results in equations (5.2), (5.3) (5.5) and (5.6) we obtain an estimation of the NCs for a large problem which are very close to the empirical results shown in table 5.2. It is important to note that for each data type used (byte, simple and double precision floating-point) the $NC(mem)$ differs since L changes (32, 8, 4) — see equations (5.5) and (5.6). For this reason the usage of simple floating-point data produces better NCs than the use of double floating-point data since both have the same $NC(cpu)$. Despite producing a lower $NC(mem)$, the use of byte data results in worse NCs since its $NC(cpu)$ component is too large to make it competitive.

5.3 Block Algorithm

In this section we present a new code which exploits the locality in the algorithm considerably better, also producing an $NC(mem) \approx 0$ for large problems. Figure 5.4a shows the code of a block algorithm we propose for substituting the jki algorithm. The same idea can be applied to the jki_exit algorithm. In this code the number of loops has increased, but the arithmetic operations are the same. Consequently, the accuracy is exactly the same as that of the non-blocked algorithms. Figure 5.4b shows the Data and Computation Diagram [133] for the *block* algorithm. The shaded area shows cached data that can be reused. In the *block* code, the same column of P is referenced for each iteration of loop J , while a new column of D is accessed. The probability of cache hits for the data in D is very high, and we will assume no misses appear. For each iteration of loop K all the elements in a block of $V_{size} \times B_{size}$ elements of D are referenced. The block size B_{size} will be dimensioned so that the block fits into the cache: $B_{size} \times V_{size} < \frac{C}{E_{size}}$. If a large B_{size} is chosen, the number of intrinsic misses of P will decrease but the number of conflicts will grow. The optimal block size is considered to be approximately half the cache size C [115]. Consequently, the data in the block remain in the data cache during all the iterations of loop K. Some conflicts will arise, but their number and influence is so low to be considered null. Consequently, the $NC(mem)$ of a block algorithm is very low and can be considered insignificant.

The last two columns in table 5.3 show the experimental measures obtained for a large problem using the block algorithm proposed above for a block size $B_{size} = \frac{1}{2} \cdot \frac{256 \cdot K}{V_{size} \cdot E_{size}}$. It should be noted that the NCs obtained are almost identical to those shown in table 5.2 corresponding to a small problem (see also figure 5.5).

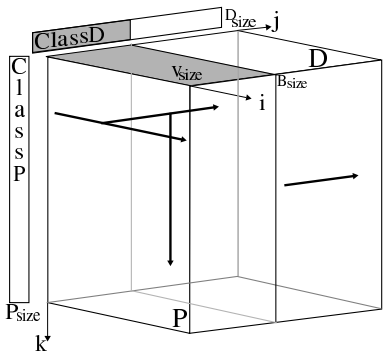
Table 5.3: NC on the PA-7150 for large problems without and with block algorithms

Data type	*jki_exit*	*jki*	*block_exit*	*block*
byte	6.4	16.1	6.6	15.2
simple float	5.7	8.0	4.8	3.7
double float	7.2	12.3	4.9	3.9

When the byte data type is used, the improvement obtained by the use of blocks is minor, because there is a large $NC(cpu)$ which was already high for the code without blocks which cannot be lowered by blocking. Moreover, there

```
MAX = 2 ** 30
DO JJ = 1, Dsize, Bsize
  DO ind = 1, Bsize
    Vaux(ind)=MAX
  ENDDO
  DO K = 1, Psize
    ind = 0
    DO J = JJ, MIN(JJ+Bsize-1,Dsize)
      ind = ind+1
      distance = 0
      DO I = 1, Vsize
         sub = D(I,J) - P(I,K)
         distance = distance + sub*sub
      ENDDO
      IF (distance.LT.Vaux(ind)) THEN
         Vaux(ind) = distance
         ClassD(J) = ClassP(K)
      ENDIF
    ENDDO
  ENDDO
ENDDO
```

a)



b)

Figure 5.4: **a)** Code for the *block* form. **b)** Data and Computation Diagram for the *block* form.

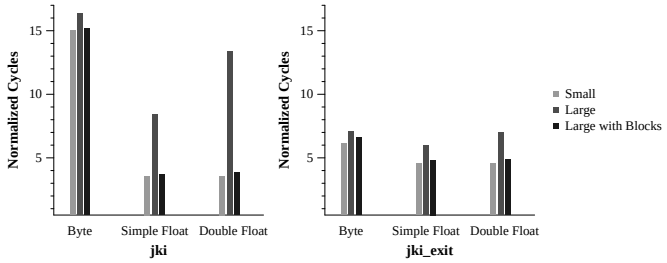


Figure 5.5: Comparison of NC for different data types (grouped by problem sizes)

exists high spatial locality due to the fact that the number of elements in a cache line is big ($L = 32$). Simple floating-point data produced slightly better results than double floating-point data (see figure 5.6) due to more efficient use of the cache line (higher spatial locality).

5.4 Optimization details

As mentioned above, the compiler did not produce very efficient code on the AXP-21064 processor. For this reason we have done the optimization by hand. We have used a high level language (Fortran). However, the code produced is targeted at this particular processor and tries to use the processor and memory in an efficient way.

We have used blocking to avoid memory problems. We have tried one block. Also, two levels of blocks, one for each cache level. Conflicts appear in the cache but can be solved by precopying data into contiguous buffers in memory before using them for the classification. In this way, we get an important performance improvement (see the leftmost three columns in table 5.4).

However, the inner kernel is not as efficient as it could be. In order to improve it we have applied software pipelining manually to produce an instruction

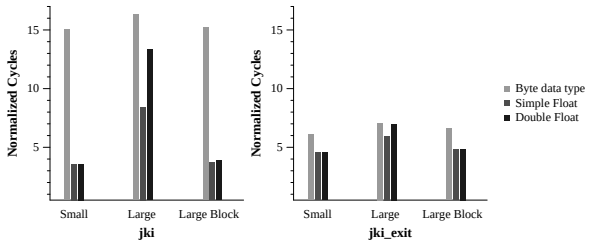


Figure 5.6: Comparison of NC for different problem sizes (grouped by data types)

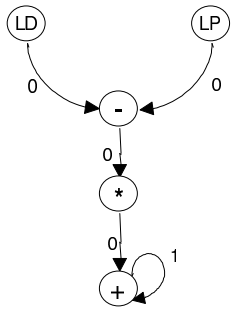


Figure 5.7: Dependence graph of inner kernel: square of Euclidean distance.

scheduling which can be executed efficiently.

The dependency graph of the square of the Euclidean distance can be found in figure 5.7. Two loads, one of element in D and one of P are done. Then, they are subtracted. Next, the result is squared, and finally accumulated. The subtraction and multiplication depend on data produced in the current iteration. The accumulation needs data reckoned in the current iteration and it also has a dependence with the operation in the previous iteration. This is the reason why the respective arcs are labeled with 0 and 1. The result of applying software pipelining to this code can be found in the fourth column in table 5.4.

Using software pipelining loops are reorganized such that each iteration in the resulting loop is made from instructions chosen from different iterations of the original loop. It is useful to reduce the time when the loop is not running at full speed [86]. When the number of instructions in the loop body is small and latencies of operations are high it can be convenient to increase the number of iterations between when we issue an instruction and when we use its result. To do this, loop unrolling can be combined with software pipelining. The limit comes from the number of available registers in the machine. In order to obtain an efficient code partial operations should be kept in registers. Thus we can consider a block at the register level (BRL).

We have tried several blocks at the register level. Figure 5.8 shows a block at the register level with 7 loads and 18 floating-point operations. Figure 5.9 shows a block with 6 loads and 27 floating-point operations. The latter reduces the number of loads and provides with more opportunities for scheduling the operations efficiently. This is the one used in the results shown in the fifth and sixth columns in table 5.4.

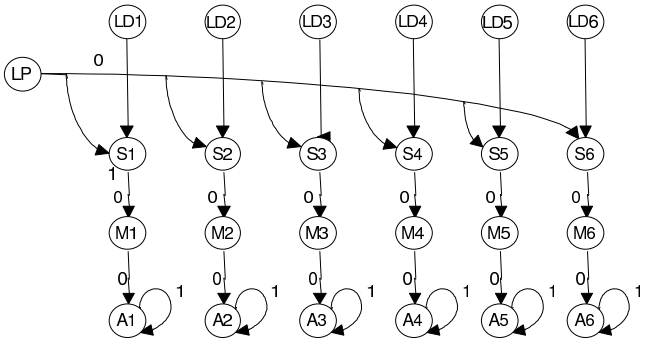


Figure 5.8: Dependence graph: rectangular block 6×1.

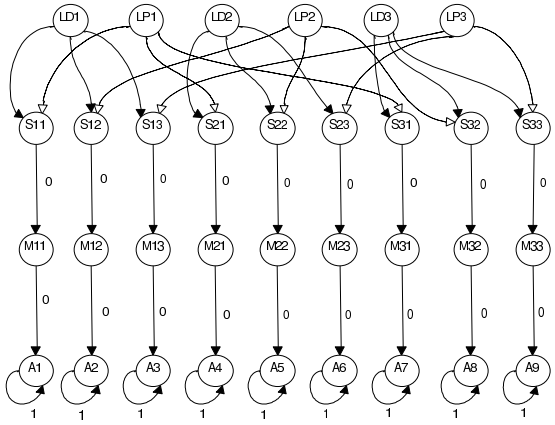


Figure 5.9: Dependence graph: square block 3×3.

Table 5.4 presents the results obtained when we hand-optimized the FORTRAN code on the Alpha AXP-21064 by the application of software pipelining techniques to improve the instruction scheduling for a better instruction level parallelism, and tiling to improve the use of data locality. We show only the results obtained for the single precision float data type since that was the one producing the best performance. SP means Software Pipelining, *num* Bl, means a *num* number of square blocks was applied, Pc implies that data precopies were done; BRL stands for Blocking at the Register Level meaning that tiling was also applied in order to improve the reuse of registers in the inner loop. The data in the table shows that the combination of well-known techniques applied to the Nearest Neighbor Algorithm produces significant improvements in performance. Figure 5.10 summarizes the results obtained on large matrices on this machine.

Table 5.4: NC obtained with hand optimized code on the AXP-21064

Problem Size	No SP			With SP		
	1 Bl.	*2 Bl.*	*2 Bl + Pc*	*1 Bl*	*1 Bl + Pc + BRL*	*2 Bl + Pc + BRL*
small	13.7	15.0	11.0	8.1	4.8	4.6
large	14.7	15.1	11.2	9.4	5.6	4.6

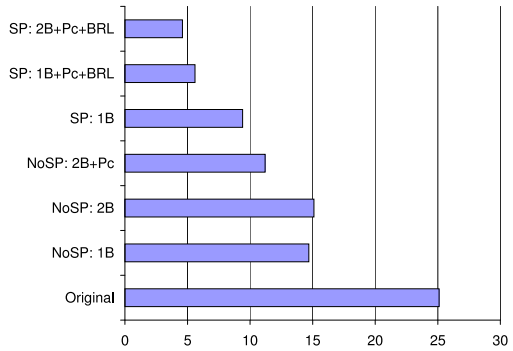


Figure 5.10: Alpha AXP-21064: Comparison of NC for different codes on a large problem.

Larger prototype vectors

The prototype vectors used in our experiments had 80 elements. In this case we do not need a block in the V_{size} direction. However, larger dimensions would require such block. We have experimented with larger prototype vectors of length 800. Using another block in this direction we could maintain the same value of NC.

5.5 Conclusions

NN classification has the significant drawback of requiring a large number of computations and data accesses which make it slow if the advantages that current computer architectures offer are not used to full advantage. Frequently, the byte data type has been used in an attempt to reduce the memory usage. In order to decrease the number of computations, an IF statement has often been added to the inner loop to test whether a better solution has already been found. In our experiments, this improved performance by a factor larger than 2. However, this is not the best solution available. Due to processor characteristics, the usage of floating-point arithmetic outperforms the use of integer arithmetic. The resulting machine code can run faster because the instruction level parallelism is higher and no data conversions are needed.

The disadvantage introduced by the use of floating-point data is the larger amount of memory used. This issue can be overcome easily by means of block algorithms. When these kinds of algorithms are used, the temporal locality of programs is better exploited resulting in low number of cache misses, allowing the computations to proceed at full speed. The use of simple floating-point data produces fewer misses than the use of double precision floating-point data due to better usage of spatial locality. However, the difference from the latter is almost negligible because of the reduced number of cache misses incurred when a block algorithm is used (see figure 5.6). In our experiments, the results obtained when a block algorithm and simple precision floating-point data are used are between 2 and 4 times faster than the algorithms which use integer arithmetic although they require 4 or 8 times more data storage. These results can be generalized for other superscalar architectures.

Acknowledgments

We would like to thank Clemente Rodríguez from "Universidad del Pais Vasco" for introducing us to the problem of NN classification and providing us with real data.

Chapter 6

POSTDATE: Performance Oriented SofTware Development And Tuning Environment

New algorithms are constantly developed in search of better or faster results. Many variants of code are often tried while searching for the best solution. When the number of code variants or possible input parameters is very high, the process of building the codes, benchmarking them, and analyzing the results can become cumbersome and error prone. For these reasons we have written a set of tools which help us in the development and benchmarking of new codes.

The process of building software for different projects and platforms can be controlled by tools such as *make*. However, the user has to write complicated Makefiles for large projects. In addition, it can be difficult to handle compilation for different platforms in presence of a networked file system. In section 6.1 we present our approach for handling the build process. This work was presented in [88].

A problem may arise if the execution of a benchmarked code lasts for a very short time due to lack of precision of timers. In section 6.2 we present a framework to ensure accurate measurements.

Some codes were very similar so we created code templates in C which can parameterized using some *cpp* (C preprocessor) macros. The code reads some standard command line flags, initialize matrices adequately, launches executions, tests them, and times them. Since each algorithm may have different characteristics, they may need different test routines or ways of defining the number of operations. Thus the code is based on some macros and in order to create a new benchmark some simple files defining such macros need to be done. Actually, macros which are not defined in the new benchmark file take default values. In this way a new code can be tested and benchmarked with a very low effort.

In section 6.3 we describe a tool for automatic benchmarking which manages a database of possible parameters and the results obtained for them. We call this

tool *BMT*. It can automatically choose the optimum code and create a target library. Our tool can handle both parameters which are used at compilation time and parameters used at execution time. Using it we have generated a library specialized in the operation on very small matrices presented in section 2.3. This work was presented in [87].

6.1 Development tools

In this section we present a new approach to writing Makefiles and a system called *maker* which helps in this process. Our main goals are: ease the process of writing user Makefiles, reuse variable and rule definitions, handle common tasks automatically (dependency tracking, preparation of code and environment for testing or debugging) and provide support for software development on heterogeneous environments (automatic creation of targets in a specific build tree for each architecture while working in the source tree; use of appropriate compiler name, flags and libraries; preparation of environment variables for finding libraries and programs). This work was presented in [88].

6.1.1 Introduction

Building large software packages is a complex task. Source code is usually scattered over many files and directories. Creation of destination files out of the source files can be eased with the help of build programs. One such program is make [53] which has become a de facto standard in the Unix world. make uses files called Makefiles to get directions on how targets have to be built [140]. There are many flavors of make [44, 56, 119, 161]. However, one of them is particularly attractive: GNU make or gmake [161]. It is freely distributed, has a largely extended functionality over conventional make versions, and it is available for many different platforms. There exists a tool called pgmake which extends gmake's utility to support distributed job execution [119].

In principle, creation of Makefiles is rather easy. However, it can become tedious work when handling projects with many directories and files. Common targets and variables are often repeated in Makefiles in different directories. For instance, a target called clean is commonly used to delete all files in the current directory that are created by building the program. A target as this will be probably found in as many Makefiles as directories in the project.

Dealing with file dependencies can become rather complicated when included header files include other files themselves. There are ways to get this dependencies from the compiler but most people either do not know this is possible or run it only at the time they create the Makefile. Unless the dependency list is updated dynamically when a change is detected, inconsistencies can occur.

Another difficulty arises when one wants to build the project in a different directory structure from where the sources are, i.e. the build tree is different from the source tree. The problem with this is the time and effort it takes to change to the directory and invoke make with the -f option followed by a possibly long path compared to the time it takes to type make in the current directory.

Finally, handling compilation in a heterogeneous environment with different platforms can be a real pain. The compiler or the library names, command line

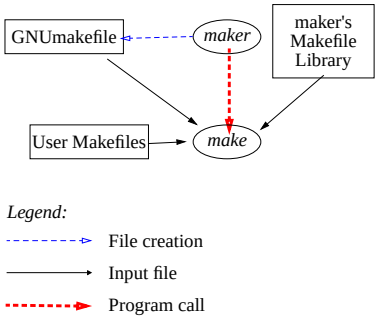


Figure 6.1: System architecture.

options or the location of files in different systems can differ substantially. Some of these problems have already been tackled by other tools:

- BSD make [44] and nmake [56] use Makefile templates which can be included from other Makefiles to allow for reuse of variable and rule definitions.
- automake [127] and autoconf [126] help in the process of writing portable code and its later distribution and installation. They also tackle the problem of dealing with a directory hierarchy and using different build trees.

In this section we present a new tool called maker: a front end to make that solves the problems stated above and greatly simplifies the creation of user Makefiles. The main advantages to maker's users are: first, user Makefiles are remarkably simple since they can reuse rules and definitions. Second, the user does not need to deal explicitly with build directories specific to each target architecture. Instead, the user can call maker from the source tree and it will automatically use adequate compiler flags and create the targets in the appropriate build tree. This is particularly interesting in a heterogeneous system of computers using a Network File System [158]. For the time being, however, maker does not address the problem of code portability and distribution.

6.1.2 System Architecture

An overview of the system architecture is shown in figure 6.1.

The system has three components: 1) a program called maker which drives the execution, 2) a set of predefined Makefiles which we refer to as maker's Makefile library, and 3) an intermediate Makefile, named GNUmakefile, built and used by maker which acts as link between the user Makefiles and the Makefile library

maker

maker is a wrapper around make (gmake). It is a program written in Perl [166] that automatically changes to the appropriate build tree and calls gmake from

there. It is responsible for passing gmake the correct parameters for operation. This includes passing the name of the Makefile in the source tree with the correct path. In order to do that, an intermediate Makefile is generated and used. This file is called GNUmakefile and needs to be generated only once, for the root of the project tree (PRJROOT). Other tasks performed by maker are the creation of certain files or directories if they do not exist and are necessary. The GNUmakefile and the build tree directories (with the same tree structure as in the source tree) are always needed. The debug build tree and debugger initialization file are only built when needed.

The GNUmakefile

The GNUmakefile is a Makefile automatically created by maker which allows for an automatic inclusion of the Makefile Library files into the user Makefiles. It defines some variables which are important for the correct operation of the system. These variables are:

- PRJROOT

  The absolute name of the project source tree root. This is useful to locate per project Makefiles or header files (if stored in "$(PRJROOT)/include" since this directory is automatically added to the search path for header files).

- PRJCWD

  The absolute name of the of the project current working directory in the source tree. When make is called the current working directory is in the build tree. Since source files come from the source tree, we need a way to specify the matching directory in the source tree where source files can be found.

There are also 3 variables that gmake uses to control its behavior:

- MAKEFLAGS

  This variable is automatically passed to a sub-make, i.e. a recursive invocation of make. Amongst data passed are directories where Makefiles can be found. This includes the system wide maker directory so that the Makefile Library files are found; directory "$(PRJROOT)/Makefiles" so that per project Makefiles can be found (this usually applies to file Makefile.prj); and the current working directory in the source tree (PRJCWD).

- MAKEFILES

  This variable keeps the name of all the Makefiles to be read on every invocation of make. These files are, in load order: Makefile.sys Makefile.prj Makefile.cfg and Makefile.lib. Thus, these files will be automatically loaded upon call to the user's Makefile in each subdirectory.

- VPATH

  Keeps the search path for all dependencies. It contains values given at the command line, plus the PRJCWD

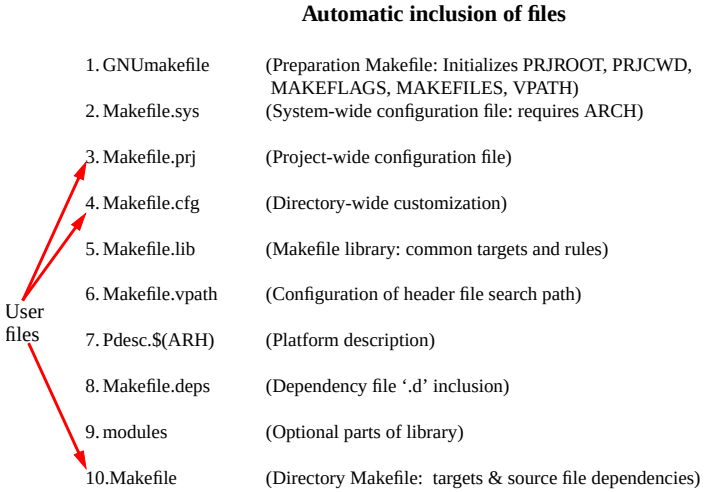


Figure 6.2: Files automatically included.

Once these variables are defined, make is invoked for the user's Makefile in $(PRJCWD)/Makefile. By the time it is loaded the Makefile library is already loaded and fully available for use as if it was coded in the user file. Only one GNUmakefile is necessary for a whole project's directory hierarchy. This file must correspond to the project root directory and can be used to prepare the build process for any subdirectory under the project root directory. This allows for finding header files in "$(PRJROOT)/include" or libraries created in some other subdirectory within the project directory hierarchy.

The GNUmakefile is created only once and reused henceforward. It is automatically created by maker if it does not exist but can also be created on demand. If the project is moved to a different directory the GNUmakefile has to be regenerated since the PRJROOT is hard coded in this file.

Makefile library

We have a set of files defining common rules and targets. These files are automatically included in the order specified in figure 6.2. The user only has to provide a maximum of 3 out of ten files.

The Makefile library is a set of predefined Makefiles which contain variable initializations and/or declaration of rules and functions. These files are read before the user Makefile is read so that all their contents are available to the user. They are presented in the same order as they are loaded:

- Makefile.sys

  This is the system-wide configuration Makefile. In this file variables controlling the build process are set to their default values. Variables could be grouped into several categories: software used, variables used as compiler or linker options, installation directories, default filenames, search paths and platform description. It assumes environment variable ARCH

is defined and its value describes the architecture where the execution is being performed. This is the only environment variable that maker needs predefined. In case it is not predefined the specific platform description will simply not be used. Instead, some variables such as compiler names or flags will be set to default values.

At this point, the project file Makefile.prj would be read if it exists. Next, the user file Makefile.cfg would be loaded if the user had provided one for the current directory being processed. Afterwards, Makefile.lib is automatically included:

- Makefile.lib

  This file defines a set of common targets and rules and is meant to be used as a Makefile library. At this time this file has about 1700 lines which are mainly common targets such as clean or install, or pattern rules such as "%.so: %.a" (which defines the way shared libraries can be created from archives) or "%.d: %.c" (for creation of dependency files for C programs). A rule for recursion into directories listed in variable SUBDIRS is also given in this file. Three files are included at the very beginning of Makefile.lib: Makefile.vpath, Pdesc.$(ARCH), and Makefile.deps. At the end, some optional parts of the library called maker modules can be loaded.

- Makefile.vpath

  This is the Makefile used for configuring the header file search path. This path includes the project current working directory in the source tree, the project include directory, a system dependent and a system independent include directories. The user can specify other directories for header file searching in a variable called USRINC which is placed before the rest of the directories in the header file search path. The order directories are added to the search path allows for file interposition: a directory or a project can use their own header files overriding the system ones with the same name.

- Pdesc.$(ARCH)

  This is the platform description. This file defines architecture/system dependent environment variables. Environment variable ARCH should be set accordingly before maker is used. Otherwise default values are used.

- Makefile.deps

  This file defines a way to include '.d' dependency files in Makefiles. This inclusion will be automatically done for files listed in variables CSRC and FSRC (for C and Fortran source files).

- maker modules

  Optionally, some parts of the library which are only used in some cases can be loaded. We call these parts maker modules. They extend the Makefile Library functionality while preserving performance. They are only read on user demand as specified on variable USE_MAKER_MODULES.

At this point, user file Makefile would be loaded and the build process would start.

User Makefiles

Per project:

- Makefile.prj

  This is the project-wide configuration Makefile and should be stored in directory $(PRJROOT)/Makefiles. Since this file is included for every subdirectory in the project, common definitions can be placed here.

Per directory:

- Makefile.cfg

  If present, this file is read by make immediately before loading file Makefile.lib and can thus be used to customize some behavior of the library. For instance, defining variable CSRC to list all C files used in that directory will trigger the automatic inclusion of dependency files which can itself trigger the automatic dependency generation in case dependency files are inexistent or need to be rebuilt.

- Makefile

  It should specify the targets for the directory as well as the source file dependencies for each target she wants to build and the way it has to be built (unless a rule already defined by gmake or our Makefile Library applies).

An equivalent solution could be achieved with a single user file per directory which had a first part with the contents of the .cfg file, followed by an explicit inclusion of file Makefile.lib, and a final part with the targets in Makefile. This solution is possible but has not been used to avoid the explicit inclusion of Makefile.lib by the user.

6.1.3 Available Operations

For the time being maker extends make basic functionality in many ways. The following list shows briefly some aspects for which maker either does it automatically, or supports it with a minimum effort from the user, such as specifying a simple option from the command line:

- Change current working directory to the appropriate directory in the build tree before calling gmake
- Generation of the project GNUmakefile
- Inclusion of makefiles: system, project, configuration, library and platform description
  - Initialization of variables with default values
  - Definition of common targets
  - Definition of new rules
- Dynamic dependence generation

  - Invoke cpp to create .d files listing all dependencies with header files for each C or Fortran files listed in variables CSRC and FSRC; recompute the .d files whenever necessary
  - Include the corresponding .d files for all files listed in CSRC and FSRC

- Definition of VPATH for file searches
- Search of header files and generation of proper compiler option for adding header file directories
- Recursion into subdirectories listed in variable SUBDIRS
- Treatment of shells or interpreters in sharp-bang (#!) lines:

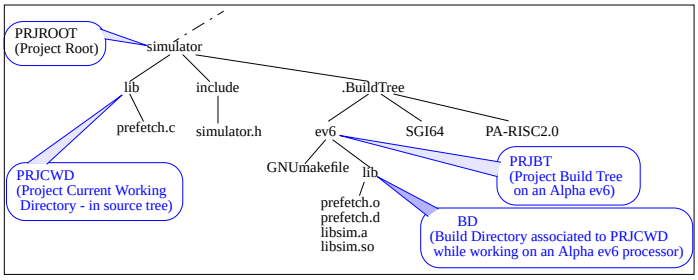
  e.g. create an executable file *namefile* using the contents of *namefile.pl* and replacing some predefined strings with adequate values (such as replace *#YOUR_PERL_INTERPRETER#* with the absolute name of Perl in the system)
- Creation of a testing environment (in the same window or in a new *xterm*) defining environment variables:
  - PRJROOT: if set to the project root directory then maker can be successfully executed from any subdirectory
  - PATH and LD_LIBRARY_PATH: extended with the proper directories in the build tree, they allow for using new executable files and dynamic libraries created in the build tree
- Creation of tag or debugger initialization files for a whole project tree
- Use compiler options for debugging and a different build tree when *maker –dbg* is issued.

Example

The upper part of figure 6.3 shows the important directories used for a build performed on an Alpha ev6 processor ($ARCH=ev6), maker variables, and files generated. Makefiles and files other than prefetch.* have been omitted to simplify the graph. The other subdirectories under ".BuildTree" would be used on builds on other systems. The lower part of figure 6.3 shows an example for a simple software project. The figure presents the Makefiles for a simple directory structure under directory "simulator" with two subdirectories called lib and include.

Using maker a number of actions would be performed for the Makefiles presented above:

- Create directory .BuildTree/$ARCH with a GNUmakefile inside
- Create a directory hierarchy mirroring the source tree hierarchy under directory .BuildTree/$ARCH (in this case it only creates one subdirectory called lib since the Makefile in directory simulator has SUBDIRS=lib and the ones in lib have no SUBDIRS



File simulator/Makefile

```
SUBDIRS=lib
TARGETS =

all: $(TARGETS)
```

File simulator/lib/Makefile

```
all: $(TARGETS)

$(LIBNAME).a: $(LIBNAME).a($(MODS))
```

File simulator/lib/Makefile.cfg

```
LIBNAME =  libsim
TARGET_LIBRARIES = $(LIBNAME).a $(LIBNAME).so
INSTALL_LIST_LIBRARIES_SYSDEP = $(TARGET_LIBRARIES)
FSRC =  stride.F
CSRC = branch.c prefetch.c
MODS = $(FSRC:.F=.o)
TARGETS = $(TARGET_LIBRARIES)
```

Directory structure

```
simulator
├ Makefile
├ lib
│  ├ Makefile
│  ├ Makefile.cfg
│  ├ branch.c
│  ├ prefetch.c
│  └ stride.F
└ include
   └ simulator.h
```

Figure 6.3: Example: User Makefiles and file system tree.

- Set maker variables: PRJROOT, PRJCWD, ...
- Change directory: chdir to $PRJBT. Nothing to be done in there other than doing recursion into SUBDIRS
- chdir into $PRJBT/lib, set PRJCWD to simulator/lib and call make with the user makefile $PRJCWD/Makefile (the Makefile library and the other user file Makefile.cfg will be loaded automatically)
- Directory $PRJROOT/includes is automatically used for header files search
- Create dependency files for all files in CSRC and FSRC (branch.d, prefetch.d and stride.d) and include them
- Compile code into object files: branch.o, prefetch.o and stride.o

- Create libsim.a
- Create libsim.so
- If the user typed "maker install" files libsim.a and libsim.so would be installed in a system dependent directory for libraries. This directory is specified in variable LIBDIR_SYSDEP, whose value is set in Makefile.sys (unless it is overridden by the user).

Drawbacks

There a number of issues that can limit the use of maker by new users:

- It requires a Perl interpreter and the definition of an environment variable $ARCH to work
- Debugging the Makefiles can become hard
- The -n option of make prints a large amount of information due to the complexity of the rule for automatic recursion into subdirectories.

6.1.4 Related Work

A number of tools exist which provide a front-end to make while extending its functionality. Amongst them there are two which are particularly relevant: imake [49] and automake + autoconf [126, 127]. It should be clear that maker was not thought as a replacement to these tools. Our main goal was not world-wide distribution of code as is their case.

Compared to these two systems Makefiles controlled by maker can be much shorter since they share many commonalities. This is particularly important when several build trees for different architectures are present in the same system. This could happen in a heterogeneous network of computers with a transparent file system access such as the one provided by NFS [158]. Using automake and autoconf a whole new set of Makefiles would be created for each architecture. Also, the user would have to be aware of building the project in a different directory each time if she wanted them to coexist.

Using maker, the only requirement is that environment variable ARCH has to be set to the name of the current platform. Taken this into account, the rest is left to maker. Using it, there is only one version of each Makefile, which is stored in the source tree. The user does not need to change to different directories for compilation since maker automatically does it. The default build tree root name is ".BuildTree/$ARCH". Consequently maker would create and use a new build tree each time a new machine was used for building the package. The proper platform description would be automatically used by maker and no changes in the Makefiles nor creation of new ones would be required. Separation into different trees automatically is very convenient: it makes things easier for the user and less error prone.

The definition of ARCH as an environment variable can be easily done in the shell initialization files. Actually, some shells already define variables such as MACHTYPE or HOSTTYPE which can be helpful. Some systems provide a command which offers useful information: psrinfo, hinv, arch, mach. In an environment with heterogeneous systems sharing files with a Network File System [158] a script can be used to automate the process.

6.1.5 Conclusions

When software is created, the process of building the final executable can be eased if a program as make is used for directing recompilation. make is directed by the information written in Makefiles. Preparation of Makefiles can become tedious or difficult as the project grows or different platforms have to be supported.

We have presented a new approach to software development based on sharing Makefiles. An architecture has been defined which is flexible and extensible. Data used for directing builds is spread over several files in a logical way. Then, the system manages these files in a way that common data is automatically shared amongst Makefiles. The system architecture presented in this section has been used to develop maker, a tool which has proved extremely useful for the author when developing large software projects.

- Usual targets and rules can be used without having to write them in user Makefiles
- User makefiles are cleaner, shorter and easier to maintain
- Developing software in a heterogeneous system with a Network File System becomes much easier

We believe maker can be very useful for people interested in developing software, not Makefiles.

6.2 Accurate Measurements

A hairy point when we want to compare the speed of two codes has to do with the precision of timers. Most current processors have hardware counters which can be used to get very accurate information on processor usage. However, there exist situations where we have not been able to use such counters. For instance on older platforms. Also, on machines where the activation of hardware counters needed privileges which we didn't have. For this reason we had to deal with the precision of timers. Usually the user can choose amongst several timing devices. Each one has a certain precision: millisecond, microseconds or nanoseconds. When the amount of work is too low for the precision of the timer, the time, or the number of operations per second, reported may be wrong (imprecise). Thus, it is common practise to repeat a given operation enough times so that the precision of the timer does not yield wrong results. When this process is handled automatically, the situation is even more critical. We need a way to determine automatically the number of iteration necessary for obtaining precise results. For instance, we want a number of Mflops with a precision of 1 Mflop. From the definition of Mflops (millions of operations per second) we get the derivative w.r.t. T (the total time). With a simple substitution we can express the number of iterations as a function of the expected Mflops of the algorithm (to play it safe we use the theoretical peak for the machine), the operations performed in a single iteration of the algorithm, the precision of the timer and the desired precision for the Mflops. Thus we obtain a formula which we have been able to express with another macro. In this way our algorithms execute a number of iterations which is reckoned at runtime and are enough to get correct

results. The number of iterations gets smaller as the problem size grows or a more precise timer is used.

6.2.1 Related work

It is well understood that collecting performance data on applications programs relying on timers with poor resolution or granularity is undesirable. Nowadays, most modern processors provide hardware counters which can provide accurate information about the performance of the applications. Thus, most performance tools rely on such counters. One problem with their use on different platforms comes from the different interfaces which have to be used. Fortunately, there have been several attempts to provide portable interfaces, such as PCL [24] or PAPI [28]. Should any of them be available, we would advice using them. However, on some systems we may not have access to such tools. This can happen on rather outdated computers for which one still wants to automatically adapt some code, or on systems where such tools are not installed and one requires privileges, which do not have, for installation. For instance, to patch an operating system kernel. Under such circumstances we may be interested in using timers.

Standards have been created for timers and clocks [103]. However, regardless of the precision (milli, micro or nanoseconds), we can have a problem of lack of precision when the benchmarked code is executed very quickly and lasts for about the timer precision. Previous work was done in order to achieve high resolution timing with low resolution clocks [40]. Such high resolution can be achieved by repeated execution of a benchmark with a number of iterations through the code. Our approach resembles theirs. However, we provide a way to determine the number of iterations automatically.

Portable Timing Routines (PTR) [148] is a Ptools project defining a standard API for measuring intervals of program execution, in terms of wallclock, user CPU, and system CPU time. We provide some routines similar to theirs which can ease the benchmarking process.

6.2.2 Theoretical foundations

The speed of a numerical algorithm is usually communicated as the number of floating point operations performed per second. Often, this number is very large and is expressed in millions using the word *Mflops*.

$$Mflops = \frac{\#flops \cdot 10^{-6}}{Time}$$

The number of flops performed by an algorithm can be calculated by the programmer. The time spent in its execution can be obtained with some system calls offered by the operating system. Each of these system calls should have its precision clearly specified in the manual. Regardless of the precision provided by any of such routines it is finite. If the number of operations computed is very low, a lack of precision can occur.

Our goal is to get a correct estimation of the Mflops or execution time[1] obtained with a program. We need to be sure that the difference between the

[1] We will center our discussion on the Mflops metrics since that is the usual way to measure the speed of numerical algorithms.

real and estimated Mflops is low. The variations in the estimated Mflops due to timing precision errors should be smaller than a certain threshold.

$$|\Delta Mflops| \leq Threshold \tag{6.1}$$

We can solve this problem by repeating the execution of a benchmark several times. By increasing the amount of operations we increase the time spent in their calculation. Consequently, the general expression for obtaining Mflops contains the number of iterations performed:

$$Mflops = \frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time} \tag{6.2}$$

The number of iterations needed becomes an issue. We need to know the number of iterations which are necessary to obtain performance results which are correct. At the same time, we want to avoid unnecessary iterations which do not produce significant improvements in timing precision and would only overload the system and delay the finalization of our benchmarks.

We need an expression which can be used at execution time to reckon an adequate number of iterations for a given input program and data. To obtain it we have defined the following process. First, we get the derivative of equation 6.2 with respect to $Time$:

$$\frac{\partial Mflops}{\partial Time} = -\frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time^2} \tag{6.3}$$

We disregard the sign in equation 6.3 since we need only to get an small error (in absolute terms). To get a practical implementation we take the discrete analog of the derivative:

$$\frac{\Delta Mflops}{\Delta Time} = \frac{\#flops \cdot Iterations \cdot 10^{-6}}{Time^2} \tag{6.4}$$

From equation 6.2 we express $Time$ as a function dependent on the other components:

$$Time = \frac{\#flops \cdot Iterations}{Mflops \cdot 10^6}$$

Then, substituting $Time$ in equation 6.4 we obtain:

$$\frac{\Delta Mflops}{\Delta Time} = \frac{Mflops^2 \cdot 10^6}{\#flops \cdot Iterations}$$

Hence:

$$Iterations = \frac{Mflops^2 \cdot 10^6}{\#flops} \cdot \frac{\Delta Time}{\Delta Mflops} \tag{6.5}$$

Before a code is benchmarked we do not know its Mflops. In practice, we can use the peak theoretical value for the target machine. This will ensure that the results are correct. If we have an estimation of the Mflops of an algorithm we can provide that value in order to reduce the number of iterations.

We use Mflops since this is the common metrics for evaluating the speed of numerical algorithms.

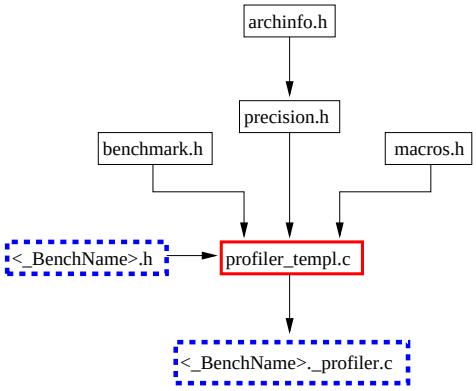


Figure 6.4: Template files in ACME and their relation.

We can use equation 6.5 to determine the number of iterations necessary to obtain results within the desired precision. For instance, consider a timing routine with a precision of 10^{-2} seconds ($\Delta Time = 10^{-2}$). We want to obtain the number of iterations of a benchmarked subroutine such that the error in the estimation of its Mflops is approximately 1 Mflop ($\Delta Mflops \approx 1$):

$$Iterations = \frac{Mflops^2 \cdot 10^6}{\#flops} \cdot \frac{10^{-2}}{1} = \frac{Mflops^2 \cdot 10^4}{\#flops} \tag{6.6}$$

6.2.3 Design

We have created a set of files which define C macros and routines which can be used to obtain accurate measurements of the performance obtained from the execution of a given code. For this reason we call this set of routines *ACME*. Figure 6.4 shows the files used and their relation. The system is mainly composed of a core file *profiler_template.c* which can be parameterized with some other files which are included via C *#include* directives. A code can be timed as long as it is callable from the C program. The user will define symbol CALL_ROUTINE indicating the way the call to her routine must be done (see the example in section 6.2.4). The files surrounded by dotted lines are supplied by the user. The rest of the files are provided by our system.

6.2.4 User files

When one wants to measure the performance of a new code and test it against another implementation, the user only has to write two additional files. A simple file which defines symbol *_BenchName* and includes the main template file *profiler_template.c*. This file is called *mtxms_profiler.c* in our example but could have any other name. Such file has the following form:

```
/* $Id: mtxms_profiler.c,v 1.1 2005/01/21 08:05:31 myusername Exp $ */
#ifndef _BenchName
#define _BenchName mtxms
#endif

#include <profiler_templ.c>
```

The only other file the user must provide is *_BenchName.h* where _BenchName should be replaced by the benchmark name (*mtxms* in the example). This is the file used to customize the benchmarking. In it, the user overrides the default macro definitions. An example follows:

```
/* $Id: mtxms.h,v 1.1 2005/01/21 08:05:31 myusername Exp $ */

#ifndef _BenchRoutine
#define _BenchRoutine \
      ad_7(_BenchName,_LDA,_LDB,_LDC,_LI,_LJ,_LK)
#endif
#ifndef _NUM_OPERATIONS
#define _NUM_OPERATIONS 2*i*j*k
#endif
#ifndef CALL_ROUTINE
#define CALL_ROUTINE \
      ad2(_BenchRoutine,_)( pdA, pdB, pdC )
#endif
#ifndef CALL_TEST_ROUTINE
#define CALL_TEST_ROUTINE \
      mtxms_test_ (pdA, pdB, pdD, &i,&j,&k, &lda,&ldb,&ldc)
#endif
#ifndef MATRIX_INITIALIZATION
#define MATRIX_INITIALIZATION \
      inimat_at_bn_(pdA,pdB,pdC,&i,&j,&k,&lda,&ldb,&ldc)
#endif

#ifndef CALL_GETINFO_BENCH
#define CALL_GETINFO_BENCH \
      ad3(getinfo_,_BenchRoutine,_)()
#endif
```

The user can use several variables declared in the system core file to denote matrices (pdA, pdB, pdC), their leading dimensions (lda, ldb, ldc) and the loop trip counts (i,j,k). Macros to compose names at compilation time (ad_2, ad_3, ...) are provided by our system. The above code defines the name of the routine to be benchmarked; the number of operations it performs; the way the routine has to be called; and the way the oracle routine has to be called. Only these definitions are compulsory.

Other symbols can be defined to allow for additional functionalities or modify the default behavior of the benchmarking system. For instance, we often include information about the parameters used at compilation time to create the executable. We can specify a routine which provides such information with CALL_GETINFO_BENCH. Defining symbol MATRIX_INITIALIZATION we can modify the default matrix initialization.

All the work necessary to drive the benchmarking is handled by the code supplied in the system files.

6.2.5 System files

Next, we present the most representative part of the files which constitute our framework.

profiler_templ.c

This file contains the template used as the main routine, the one which drives all the process. It is used to to launch benchmarks using several parameters. It is customized for a particular benchmark via a set of macros which can be defined in file _BenchName.h (where _BenchName is a preprocessor symbol which must be properly defined when cpp, the C preprocessor, is invoked).

Some symbols must be defined in _BenchName.h. In other cases the template file itself provides default values for these symbols, which can be overridden in the _BenchName.h file.

A pseudo-code with the most representative parts of this file follows:

```
/* $Id: profiler_templ.c,v 1.10 2004/11/04 15:51:27 myusername Exp $
 */

/* include header files */
...
#include <benchmark.h>
#include <macros.h>
#include <memory.h>
#include <precision.h>
...

#ifdef _BenchName
#define _FilNam <ad2(_BenchName,.h)>
#include _FilNam
#undef _FilNam
#endif

/* Default values for some preprocessor symbols */
#ifndef MATRIX_INITIALIZATION
#define MATRIX_INITIALIZATION \
      inimat_an_bt_(pdA,pdB,pdC,&i,&j,&k,&lda,&ldb,&ldc)
#endif
/* Some extra code is needed for testing some benchmarks
   but the default is not to need anything else. */
#ifndef EXTRA_DECLARATIONS
#define EXTRA_DECLARATIONS
#endif
#ifndef EXTRA_INITIALIZATIONS
#define EXTRA_INITIALIZATIONS
#endif
#ifndef EXTRA_FREEMEMORY
#define EXTRA_FREEMEMORY
#endif

#define EPSILON 10E-30

extern double validate_results_ ();

/* Global variables */
int     i, j, k, lda, ldb, ldc, it, ti;

main (argc, argv)
     int     argc;
     char   *argv[];
{
  declare_variables;
  get_parameters;
  initializations;

  it = GET_NUMITERATIONS (_NUM_OPERATIONS);

  Allocate_space;
  MATRIX_INITIALIZATION;

  if (check)
    {
      EXTRA_DECLARATIONS;

      EXTRA_INITIALIZATIONS;

      CALL_TEST_ROUTINE;

      CALL_ROUTINE;

      error = validate_results_ ( pdC, pdD, &i, &j, &ldc);
      if (error > EPSILON)
{
  printf ( xstr ( ERROR: _BenchRoutine test failed\n ) );
```

```
  printf ( "Error=%g\n", error );
  exit(-1);
} else { printf ( xstr ( OK: _BenchRoutine test succeeded\n ) ); }
     EXTRA_FREEMEMORY;
   }
  /* call BenchMarked routine  */

  GET_MFLOPS ( CALL_ROUTINE, ti, it, _NUM_OPERATIONS, mflops);

  printf ("Mflops=%f Times=%d Iterations=%d \n", mflops, ti, it );

  Free_space;
  printf ("End of Execution\n");

}
```

Some of the definitions used in this file come from other files.

macros.h

This file defines several macros for string manipulation which are useful in the creation of variable contents or routine names using preprocessor symbols at compilation time.

archinfo.h

_EX is used to specify the precision of the timing routine. PEAK_MFLOPS defines the theoretical peak performance which can be obtained on the target machine. This acts as an upper limit in the possible values for Mflops, which are not known a priori.

```
#define PEAK_MFLOPS 1000

/* Precision of timers:
 *
 * Considering precision as 10*1E-6
 *    #define _EX 1
 * Considering precision as 100*1E-6
 *    #define _EX 2
 */
#define _EX 2
```

precision.h

This file provides macros to reckon the number of iterations.

```
/* ------------------------------------------------ DESCRIPTION
 * Define "Default Mflops" in order to reckon the
 * adequate number of iterations to obtain good accuracy
 */

/* ----------------------------------------- FILES INCLUDED */
#include <math.h>
#include <archinfo.h>

/* ------------------------------------------------- BEGIN */
#ifndef _DEFMFLOP
#define _DEFMFLOP PEAK_MFLOPS * .9
#endif

/*
 * Assuming an average of _DEFMFLOP Mflops per algorithm,
 * reckon # of iterations needed to obtain enough precision: */

#define ITER_BASE \
   (_DEFMFLOP*_DEFMFLOP*(pow((double)10,(double)_EX)))
```

```
/* Example: for a MxM operation

    it= (int)
       ((unsigned long)(ITER_BASE/(unsigned long)(2*i*j*k)))+1;
    if (!it) it=1;
*/

#define GET_NUMITERATIONS(OPS) \
    (int) ((unsigned long)(ITER_BASE/(unsigned long)(OPS)))+1

#define DEBUG_PRECISION printf \
  ("DEFMFLOP=%f EX=%d IBASE=%f \n", _DEFMFLOP, _EX, ITER_BASE)

/* ------------------------------------------------------ END */
```

benchmark.h

Here we define macros which handle the timing process. The user can easily get the best time or Mflops obtained by his routine amongst a number of repetitions. The details have been omitted for the sake of brevity.

```
 /* Defines macros:
 *   GET_BEST_TIME (what,times,iterations,best_time)
 *   GET_MFLOPS    (what,times,iterations,numops,mflops)
 */

/* Examples of usage:

 GET_BEST_TIME ( ad2(_MxMtName,_)( pdA, pdB, pdC ), ti, it,
                 best_time);
 GET_MFLOPS ( ad2(_MxMtName,_)( pdA, pdB, pdC ), ti, it,
              2*i*j*k, mflops);
*/
```

6.2.6 Conclusions

We want to use hardware counters to do performance measurements whenever it is possible. However, in those cases where it is not, we still want to be able to obtain accurate performance measures using timers. To do so, we need to execute enough iterations to avoid problems of lack of precision. At the same time we do not want to perform unnecessary iterations. For a given desired precision of the result (Mflops in our case), we have shown how we can determine the number of iterations necessary to obtain it.

We have presented a framework for measuring the performance of new codes. It can be parameterized by the user to specify the way to call and test the routine being benchmarked. Then, our code will handle the benchmarking process. Depending on the problem size, it will automatically determine the adequate number of iterations.

Using this framework we have been able to validate and benchmark many linear algebra codes in a systematic way. We have been able to tune automatically our libraries for several platforms getting good performance on both dense and sparse codes.

6.3 Benchmarking tool

We have very often been comparing different codes in search for the one which provides best performance. We created an environment to ease this task.

6.3.1 Automatic benchmarking: motivation

Some times code developers need to compare multiple variants of a program. Each variant can be chosen either at compilation time or at execution time. Conditional compilation can be done in case a particular variant amongst several has to be chosen at compilation time. Conditional compilation is performed by means of preprocessor directives. Depending on whether a symbol is defined or not, or the value it has at compilation time, certain parts of code are included or excluded, or take one value or another. In order to know the exact variant of code we are using we need to keep track of all parameters specified at compilation time. Similarly, when a program is executed it can often have multiple input parameters.

Consequently, the results of the execution of a program should be kept associated to all input parameters, i.e. to all parameters used at compilation time to generate the object plus information on all inputs used at execution time. Unless we keep all this information we will not be able to do accurate studies with those results. A result can become useless if we cannot know exactly how it was obtained.

As the number of parameters grows, the number of combinations tested can become very large. Keeping all necessary data for each combination can be tedious work and error prone. Thus, it is desirable and advisable to automate this process.

6.3.2 Automatic performance optimization of libraries

In these sections we present a tool we have developed and how it has been used to optimize a library of routines specialized in operating on small matrices. We call this tool BMT, which stands for *BenchMarking Tool*.

We only know of one tool which has some similarity with BMT. We refer to a commercial product called ST-ORM [162]. This tool allows for stochastic analysis. It can be used to launch executions. It maintains a database of results and offers statistical and graphical treatment of these results. However, as far as we know, it does not offer the possibility to generate optimized libraries.

6.3.3 Features of BMT at a glance

BMT is a program written in Perl that can launch compilation of programs as well as its posterior execution. In each case, BMT prepares the parameters to be used, analyzes the compilation and execution processes searching for errors, and keeps the results in a database. Statistics and plots can be obtained. The tool can also inform about the optimal combination of parameters amongst a set of combinations. Using this information it can produce a library of optimal routines.

The user can specify a set of benchmarks to run. For each benchmark one can specify parameters for compilation, including definition of preprocessor symbols, and parameters for execution. For each parameter the user can specify a set of values to use.

6.3.4 Important aspects of automatic benchmarking

Handling a variable number of parameters

When several parameters have to be used, we need a way to generate all their combinations. The common way to do that is to define several nested loops. Each loop is associated to one parameter. Then, the induction variable in each loop gets its values from the list of values that its associated parameter can have.

In general, we need a number of nested loops that equals the number of parameters to use. However, a general benchmarking application must be able to deal with any number of parameters: the number of parameters used for one benchmark can be different from that used for another benchmark. Thus, we cannot hard-code a particular number of nested loops in the benchmarking tool since that would limit its applicability. Instead, we simulate any number of loops. We generate all combinations of the input parameters, one by one, with a single loop. For each parameter, we keep information about the next value that must be used from its list of values in generating the next combination.

Checking the correctness of an execution

When hundreds or thousands of executions are launched the possibility of getting errors increases. Some of them can be transient errors. Others can be permanent (real) errors. For instance, if a process is killed externally, a disk quota is exceeded or a machine crashes, the execution finishes abruptly. However, launching the process a second time can produce correct results. In this case we have a transient error.

If, for instance, a routine expects a positive integer as input and we feed it with a negative integer we will repeatedly get an incorrect execution. In this case we should not keep trying but, instead, keep track of the incorrectness of the parameter combination tried as input. In this way we could avoid future retries.

Enforcing robustness of programs

We want to be able to check that a program worked correctly and all the parameters it used in its execution were those we expect. To make sure that the process ended as expected we have it to output a given string just before it finishes execution, e.g. *End of execution.* Thus, we can check whether this string appears in the output of the execution. In this way we can detect unexpected termination of programs. When such an error is detected we can decide whether to launch the execution again or not.

In order to make sure that a program is using the very same parameters as we expect it to use, we force it to output all the parameters that affect its execution. Both parameters used at compilation time and parameters used at execution time. For each parameter we have it to write the parameter name and its value. Then, after an execution is completed we can parse its output and check all parameter-value pairs. If all of them match the current parameter combination then we consider the execution correct and accept to keep its results in the database. Otherwise those results are discarded and an error flagged.

Dealing with concurrency

We believe it is important to allow for concurrent compilation and execution of programs. Concurrency affects the way we handle compilations, executions and accesses to the database.

We use a different (temporary) directory for each compilation. Thus, several compilations can proceed at once. Once we obtain the executable we rename it with a name that includes all parameters used at compilation time and store it in a directory where we store all executable files handled by BMT. When each execution is launched we redirect the standard output and standard error to a file whose name starts with the name of the executable followed by all the input parameters used for its execution. Finally, the output file is analyzed looking for the information we want to store in the database of results. When the database is managed we ensure mutual exclusion by using the usual lock procedures.

Chapter 7

Conclusions and future work

Adapting the code to the target machine is fundamental for obtaining high performance implementations of an algorithm. For many numerical algorithms the matrix multiplication operation becomes the most time consuming part. Thus, it is very important to obtain efficient implementations of the matrix multiplication operation for each platform.

Creation of efficient code has traditionally been done manually using assembly language and based on a great knowledge of the target architecture. Such an approach, however cannot be easily undertaken for many target architectures and algorithms. On the other hand, the Fortran implementation of Basic Linear Algebra Subroutines (BLAS) is inefficient. Consequently, there have been attempts to produce such codes automatically. A new paradigm was created: Automated Empirical Optimization of Software (AEOS). The goal is to use empirical timings to adapt a package automatically to a new computer architecture. The ATLAS package implements this approach. However, the performance obtained is sometimes low. Consequently, a great effort has been applied to produce high performance inner kernels for matrix multiplication using hand-coded routines contributed by some experts.

We have based our approach in the creation of efficient matrix multiplication kernels adapted to the target machine. For each platform we build automatically a library called Small Matrix Library (SML). Routines in this library are specialized in the operation on small matrices (matrices which fit in the lower level cache). We provide several variants of code written in a high level language (Fortran). These variants implement the same operation using different loop orders and unroll factors. We compile and benchmark each of them, keeping the one which provides the best performance. By fixing leading dimensions and loop trip counts at compilation time we can produce very efficient codes if a good compiler is available. Thus, routines in this library work on small matrices of fixed size. Once we have an efficient routine operating on small matrices we use it in other codes which deal with matrices of any size. We use this approach in both the dense and sparse fields.

The use of new data formats for dense matrix computations is currently an active area of research. We have experimented with two nonlinear array lay-

outs: one is based on a recursive partitioning and storage of matrices using a hypermatrix scheme (HM); the other uses a square block data layout (SB). In both cases the performance obtained is highly competitive with the matrix multiplication and Cholesky factorization implementations offered by the broadly used ATLAS library. The code which works on hypermatrices has two disadvantages: first, there is some overhead in following the data structure recursively; and second, parallelism cannot be effectively exploited in an easy way, resulting in load unbalance for many matrix dimensions. The implementation based on a simple square block data layout outperforms the hypermatrix oriented codes for dense operations. Our implementation, based on the compiler-optimized codes in SML achieves good performance on a variety of platforms, approaching that of hand-optimized codes.

We have also worked on the optimization of a sparse Cholesky factorization based on a hypermatrix data structure. The use of such scheme has a drawback: the storage of and computation on zeros within data submatrices. We have studied several techniques aiming to the reduction of the overhead introduced by the operation on zeros. The most effective ones are the use of the SML routines corresponding to rectangular matrices, the use of windows, the use of 2D recursive layout of data submatrices and scheduling of the computations, and the intra-block amalgamation. Using such techniques we obtained competitive performance. Our hypermatrix Cholesky factorization outperforms TAUCS for many matrices arising in Interior Point Methods, and obtain similar performance on matrices obtained from the application of Finite Element Methods.

In this thesis we have also shown that techniques commonly used in linear algebra codes can be effectively applied to other kinds of codes such as a Nearest Neighbor classification algorithm. For example, for some classification problems data elements can be represented using a single byte. Even when data can be stored using bytes, it can be more effective to use real numbers to store such data. This is the case on machines optimized for fast floating-point operations. The additional amount of memory used could become a problem. However, using tiling techniques such possibility is avoided and high performance can be obtained even for very large classification data sets.

With the experience of having studied and optimized codes from different fields, we confirm that some of the key factors which must be taken into account when searching for high performance codes are:

- There is a trade-off between the speed of an algorithm and: the memory space used; the computation of non productive operations. Using extra space can be alleviated by tiling techniques. Allowing for some non productive operations may result in more regular codes which can, sometimes, be executed faster. Examples of this have been presented for the sparse Cholesky factorization and the Nearest Neighbor classification.
- Some aspects are fundamental in obtaining high performance implementations of an algorithm. Namely, having: data accessed with stride one; data properly aligned; store operations removed from the innermost loop.
- Current compilers can generate very efficient codes when working on simple and regular codes.

Even when some of these rules are followed, the process of optimizing a code may require a large effort. There may exist several possibilities which

should be tested in search for the optimal values. For instance, different loop orders, alignment values, or compiler flags. This can be time consuming and error prone. For this reason we have developed tools which help us to adapt programs for efficient execution on several platforms. These tools have proved effective and give us a systematic way to develop and test new computer codes, and improve their performance.

Future work

After finalizing this thesis we plan to extend this work in several directions:

- Modify our sparse hypermatrix Cholesky factorization to have data submatrices accessed with stride one by operating on an upper triangular matrix (U) rather than the lower triangle (L).
- Allow for a new data storage within hypermatrices: use supernodes to store data submatrices in order to reduce the number of non productive operations performed on zeros.
- Conduct research on sparse matrix ordering algorithms. We have observed that delaying the change from a global strategy to a local ordering algorithm can produce better performance in some cases. This needs further research to identify a good criteria for deciding the switch point for a given input matrix.
- Modify the sparse Cholesky code to produce an Incomplete Cholesky factorization which can be used as a preconditioner for iterative methods. For instance, we could cast off data submatrices with a reduced number of nonzero elements. We think that avoiding the operation on such blocks could yield a robust and high performance preconditioner.
- Create a Cholesky-like LDL^T factorization for Indefinite and Quasi-definite matrices. These kinds of matrices often have blocks with high density and could benefit from the hypermatrix approach.
- Extend our sparse hypermatrix Cholesky implementation so that it is able to factor matrices out-of-core. We would also like to use the message passing paradigm for parallelization on distributed memory machines.
- Change the search for the optimal inner kernels during the creation of our Small Matrix Library. Currently, an operation is repeated several times on the very same matrices. Consequently, data can reside in cache from the second iteration, which is not necessarily what happens when the resulting subroutine is used. We want to determine the optimum algorithm for inclusion in our library by using more representative programs: codes which reference more data using realistic access patterns.
- Study the creation of optimal inner kernels for new processors with multiple cores. Similarly, study the case of graphics processing units (GPUs).
- Evaluate the impact of our SML's matrix multiplication on multimedia codes. For instance, MPEG-2 uses the Discrete Cosine Transform (DCT) or its inverse (IDCT) which includes a matrix multiplication of matrices with 8×8 elements.

Bibliography

[1] M. J. Aftosmis, M. J. Berger, and S. M. Murman. Applications of space-filling curves to cartesian methods for CFD. In *42nd Aerospace Sciences Meeting and Exhibit*, January 2004.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Dev.*, 38(5):563–576, September 1994.

[3] Nawaaz Ahmed and Keshav Pingali. Automatic generation of block-recursive codes. In *Euro-Par 2000,LNCS1900*, pages 368–378, September 2000.

[4] George Almási, Luiz De Rose, Basilio B. Fraguela, José E. Moreira, and David A. Padua. Programming for locality and parallelism with hierarchically tiled arrays. In Lawrence Rauchwerger, editor, *LCPC*, volume 2958 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2003.

[5] B. S. Andersen, J. A. Gunnels, F. Gustavson, and J. Wasniewski. A recursive formulation of the inversion of symmetric positive defite matrices in packed storage data format. In Juha Fagerholm, Juha Haataja, Jari Jarvinen, Mikko Lyly, Peter Raback and Ville Savolainen, editors, *PARA'02, Applied Parallel Computing, Espoo, Finland*, volume 2367 of *Springer series Lecture Notes in Computer Science (LNCS)*, pages 287–296, Heidelberg, June 2002. Springer - Verlag.

[6] Bjarne S. Andersen, John A. Gunnels, Fred G. Gustavson, John K. Reid, and Jerzy Waśniewski. A fully portable high performance minimal storage hybrid format Cholesky algorithm. *ACM Transactions on Mathematical Software*, 31(2):201–227, June 2005.

[7] Bjarne S. Andersen, Jerzy Wasniewski, and Fred G. Gustavson. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software (TOMS)*, 27(2):214–244, 2001.

[8] Bjarne Stig Andersen, Fred G. Gustavson, Alexander Karaivanov, Minka Marinova, Jerzy Wasniewski, and Plamen Y. Yalamov. LAWRA: Linear algebra with recursive algorithms. In Tor Sørevik, Fredrik Manne, Randi Moe, and Assefaw Hadish Gebremedhin, editors, *PARA*, volume 1947 of *Lecture Notes in Computer Science*, pages 38–51. Springer, 2000.

[9] E Anderson, Z Bai, J Dongarra, A Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proc. of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[10] E. Anderson and J. Dongarra. LAPACK User's Guide, SIAM, Philadelphia, 1992.

[11] C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, 15:291–309, 1989.

[12] M. Ast, C. Barrado, J.M. Cela, R. Fischer, O. Laborda, H. Manz, and U. Schulz. Sparse matrix structure for dynamic parallelisation efficiency. In *Euro-Par 2000,LNCS1900*, pages 519–526, September 2000.

[13] M. Ast, R. Fischer, H. Manz, and U. Schulz. PERMAS: User's reference manual, INTES publication no. 450, rev.d, 1997.

[14] Evangelia Athanasaki and Nectarios Koziris. Fast indexing for blocked array layouts to improve multi-level cache locality. In *Interaction between Compilers and Computer Architectures*, pages 109–119, 2004.

[15] Evangelia Athanasaki, Nectarios Koziris, and Panayioits Tsanakas. A tile size selection analysis for blocked array layouts. In *Interaction between Compilers and Computer Architectures*, pages 70–80, 2005.

[16] D. F. Bacon, J. H. Chow, D. R. Ju, M. Kalyan, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94*, Toronto, Ontario, October 1994.

[17] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[18] M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on the Peano curve, 2006. submitted to Linear Algebra and its Applications (Elsevier).

[19] Tamas Badics. RMFGEN generator., 1991. Code available from ftp://dimacs.rutgers.edu in directory pub/netflow/generators/network/genrmf.

[20] P. Baglietto, M. Maresca, and M. Migliardi. Image Processing on High-Performance RISC Systems, Proceedings of the IEEE, 84(7):917–930, July 1996.

[21] S. Bandyopadhyay and U. Maulik. Efficient prototype reordering in nearest neighbor classification, Pattern Recognition 35(12):2791–2799, Dec. 2002.

[22] Ioana Banicescu and Susan Flynn Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 43, New York, NY, USA, 1995. ACM Press.

[23] Jörn Behrens and Jens Zimmermann. Parallelizing an unstructured grid generator with a space-filling curve approach. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 815–823, London, UK, 2000. Springer-Verlag.

[24] Rudolph Berrendorf and Heinz Ziegler. PCL the Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors.

[25] Ganesh Bikshandi, Basilio B. Fraguela, Jia Guo, María Jesús Garzarán, Gheorghe Almási, José E. Moreira, and David A. Padua. Implementation of parallel numerical algorithms using hierarchically tiled arrays. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2004.

[26] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57, New York, NY, USA, 2006. ACM Press.

[27] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM Int. Conf. on Supercomputing*, pages 340–347. ACM Press, 1997.

[28] S Browne, J Dongarra, N Garner, G Ho, and P Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. of High Performance Computing Applications*, 14(3):189–204, 2000.

[29] W.J. Carolan, J.E. Hill, J.L. Kennington, S. Niemi, and S.J. Wichmann. An empirical evaluation of the KORBX algorithms for military airlift applications. *Oper. Res.*, 38:240–248, 1990.

[30] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In IEEE Computer Society. Technical Committee on Computer Architecture, editor, *Proceedings, Supercomputing '92: Minneapolis, Minnesota, November 16-20, 1992*, pages 114–124. IEEE Computer Society Press, 1992.

[31] Jordi Castro. A specialized interior-point algorithm for multicommodity network flows. *SIAM Journal on Optimization*, 10(3):852–877, September 2000.

[32] S. Chatterjee, L. R. Bachega, P. Bergner, K. A. Dockser, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowski, G. K. Liu, M. Mendell, R. Nair, C. D. Wait, T. J. C. Ward, and P. Wu. Design and exploitation

of a high-performance SIMD floating-point unit for Blue Gene/L. *IBM Journal of Research and Development*, 49(2/3):377–391, March/May 2005.

[33] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 13th international conference on Supercomputing*, pages 444–453. ACM Press, 1999.

[34] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *Proc. of the 11th annual ACM symposium on Parallel algorithms and architectures*, pages 222–231. ACM Press, 1999.

[35] Z. Chi, J. Wu, and H. Yan. Handwritten numeral recognition using self-organizing maps and fuzzy rules, Pattern Recognition, 28(1):59–66, 1995.

[36] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers. In *Proc. Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. ACM Press, 1992.

[37] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 279–290, June 1995.

[38] Digital Equip. Corp. DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors - Hardware Ref. Manual, 1994.

[39] J. Czyzyk, S. Mehrotra, M. Wagner, and S. J. Wright. PCx User's Guide (Version 1.1). Technical Report OTC 96/01, Optimization Technology Center, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439, 1997.

[40] Peter B. Danzig and Stephen Melvin. High resolution timing with low resolution clocks and microsecond resolution timer for Sun workstations. *SIGOPS Oper. Syst. Rev.*, 24(1):23–26, 1990.

[41] B.V. Dasarathy. Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques, IEEE Computer Society Press, 1991.

[42] Tim Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997.

[43] Michel J. Daydé and Iain S. Duff. The use of computational kernels in full and sparse linear solvers, efficient code design on high-performance RISC processors. In *VECPAR*, pages 108–139, 1996.

[44] Adam de Boor. PMake - A Tutorial. University of California, Berkeley, CA, USA, Jul. 1988.

[45] C. Decaestecker. Finding Prototypes for Nearest Neighbor Classification by Means of Gradient Descent and Deterministic Annealing, Pattern Recognition, 30(2):281–288, 1997.

[46] A. Djouadi and E. Bouktache. A Fast Algorithm for the Nearest-Neighbor Classifier, IEEE Trans. on Pattern Analysis and Machine Intelligence, 19(3):277–282, 1997.

[47] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[48] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *The International Journal of High Performance Computing Applications*, 17(2):125–131, Summer 2003.

[49] Paul DuBois. *Software Portability with* `imake`. O'Reilly & Associates, Inc., 1993.

[50] Iain S. Duff. Full matrix techniques in sparse Gaussian elimination. In *Numerical analysis (Dundee, 1981)*, volume 912 of *Lecture Notes in Math.*, pages 71–84. Springer, Berlin, 1982.

[51] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[52] K Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[53] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(4):255–265, March 1979.

[54] R. A. Finkel and J. L. Bentley. Quad trees, A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[55] Rolf Fischer, Markus Ast, Hartmut Manz, and Jesus Labarta. A dynamic task graph parallelization approach. In *Fourth Int. Colloquium on Computation of Shell & Spatial Structures*, June 2000.

[56] Glenn S. Fowler. The fourth generation Make. In USENIX Association, editor, *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 159–174. USENIX, Summer 1985.

[57] A. Frangioni. Multicommodity Min Cost Flow problems. Operations Research Group, Department of Computer Science, University of Pisa. Data available from www.di.unipi.it/di/groups/optimize/Data.

[58] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Notices*, 32(7):206–216, 1997.

[59] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[60] J. Fu and T.S. Huang. VLSI for Pattern Recognition and Image Processing, Springer-Verlag, Berlin, 1984.

[61] G.Von Fuchs, J.R. Roy, and E. Schrem. Hypermatrix solution of large sets of symmetric positive-definite linear equations. *Comp. Meth. Appl. Mech. Eng.*, 1:197–216, 1972.

[62] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive-Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[63] Alan George. Nested disection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, September 1973.

[64] Andrew V. Goldberg, Jeffrey D. Oldham, Serge Plotkin, and Cliff Stein. An implementation of a combinatorial approximation algorithm for minimum-cost multicommodity flow. In *Proceedings of the 6th International Conference on Integer Programming and Combinatorial Optimization, IPCO'98 (Houston, Texas, June 22-24, 1998)*, volume 1412 of *LNCS*, pages 338–352. Springer-Verlag, 1998.

[65] D. Goldfarb and M. D. Grigoriadis. A computational comparison of the Dinic and network simplex methods for maximum flow. In B. Simeone et al., editors, *FORTRAN Codes for Network Optimization*, Annals of Operations Research, vol. 13, pages 83–124, 1988.

[66] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, second edition, 1989.

[67] Kazushige Goto and Robert van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Univ. of Texas at Austin, November 1 2002.

[68] Nicholas I. M. Gould, Yifan Hu, and Jennifer A. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical Report RAL-TR-2005-005, Rutherford Appleton Laboratory, Oxfordshire OX11 0QX, April 2005.

[69] P.J. Grother and G.T. Candela. Comparison of handprinted digit classifiers. Technical Report NISTR 5209, National Institute of Standards and Technology (NIST), 1993.

[70] P.J. Grother, G.T. Candela, and J.L. Blue. Fast Implementation of Nearest Neighbor Classifiers, Pattern Recognition, 30(3):459–465, 1997.

[71] John A. Gunnels, Greg Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In *International Conference on Computational Science (1)*, pages 51–60, 2001.

[72] Anshul Gupta. Graph partitioning based sparse matrix orderings for interior point algorithms. Technical Report RC 20467(90480), IBM Research Division, 1996.

[73] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse-matrix ordering. *IBM J. Res. Dev.*, 41(1-2):171–183, 1997.

[74] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSMP: A high-performance shared- and distributed- memory parallel sparse linear equation solver. Technical report, IBM Research Division, T.J. Watson Research Center, April 2001.

[75] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):179–193, 1992.

[76] F. Gustavson, A. Henriksson, I. Jonsson, and B. Kaagstroem. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *LNCS*, 1541:195–206, 1998.

[77] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[78] F. G. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM J. Res. Dev.*, 47(1):31–55, January 2003.

[79] F. G. Gustavson. Algorithm Compiler Architecture Interaction Relative to Dense Linear Algebra. Technical Report RC23715 (W0509-039), IBM, T.J. Watson, September 2005.

[80] Fred G. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In *PPAM*, pages 418–436, 2001.

[81] Fred G. Gustavson. New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms. In Jack Dongarra, Kaj Madsen, and Jerzy Wasniewski, editors, *PARA*, volume 3732 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2004.

[82] H. Han, G. Rivera, and C. Tseng. Software support for improving locality in scientific codes. In *Eighth International Workshop on Compilers for Parallel Computers (CPC'2000)*, January 2000.

[83] Y. Harnamoto, S. Uchimura, and S. Tornita. A Bootstrap Technique for Nearest Neighbor Classifier Design, IEEE Trans. on Pattern Analysis and Machine Intelligence, 19(1):73–79, 1997.

[84] Ernst J. Haunschmid, Christoph W. Ueberhuber, and Peter Wurzinger. Cache oblivious high performance algorithms for matrix multiplication. Technical report, Vienna University of Technology, September 05 2002.

[85] R. Van Der Heiden and F.C.A. Groen. The Box-Cox Metric for Nearest Neighbor Classification Improvement, Pattern Recognition, 30(2):273–279, 1997.

[86] John L. Hennessy and David A. Patterson. *Computer Architecture a Quantitative Approach, 2nd. edition.* Morgan Kaufmann, 1996.

[87] José R. Herrero and Juan J. Navarro. Automatic benchmarking and optimization of codes: an experience with numerical kernels. In *Int. Conf. on Software Engineering Research and Practice*, pages 701–706. CSREA Press, June 2003.

[88] José R. Herrero and Juan J. Navarro. Building software via shared knowledge. In *Int. Conf. on Software Engineering Research and Practice*, pages 861–867. CSREA Press, June 2003.

[89] José R. Herrero and Juan J. Navarro. Improving Performance of Hypermatrix Cholesky Factorization. In *Euro-Par 2003,LNCS2790*, pages 461–469. Springer-Verlag, August 2003.

[90] José R. Herrero and Juan J. Navarro. Optimization of a statically partitioned hypermatrix sparse Cholesky factorization. In *Workshop on state-of-the-art in scientific computing (PARA'04),LNCS3732*, pages 798–807. Springer-Verlag, June 2004.

[91] José R. Herrero and Juan J. Navarro. Reducing overhead in sparse hypermatrix Cholesky factorization. In *IFIP TC5 Workshop on High Performance Computational Science and Engineering (HPCSE), World Computer Congress*, pages 143–154. Springer-Verlag, August 2004.

[92] José R. Herrero and Juan J. Navarro. Adapting linear algebra codes to the memory hierarchy using a hypermatrix scheme. In *Int. Conf. on Parallel Processing and Applied Mathematics. LNCS 3911*, September 2005.

[93] José R. Herrero and Juan J. Navarro. Efficient implementation of nearest neighbor classification. In *Int. Conf. on Computer Recognition Systems (CORES). Advances in Soft Computing, Vol XVIII*, pages 177–186, May 2005.

[94] José R. Herrero and Juan J. Navarro. Intra-block amalgamation in sparse hypermatrix Cholesky factorization. In *Int. Conf. on Computational Science and Engineering*, pages 15–22, June 2005.

[95] José R. Herrero and Juan J. Navarro. A study on load imbalance in parallel hypermatrix multiplication using OpenMP. In *Int. Conf. on Parallel Processing and Applied Mathematics. LNCS 3911*, September 2005.

[96] José R. Herrero and Juan J. Navarro. Compiler-optimized kernels: An efficient alternative to hand-coded inner kernels. In *Proceedings of the International Conference on Computational Science and its Applications (ICCSA). LNCS 3984*, pages 762–771, May 2006.

[97] José R. Herrero and Juan J. Navarro. Sparse hypermatrix Cholesky: Customization for high performance. In *Proceedings of The International MultiConference of Engineers and Computer Scientists 2006*, June 2006.

[98] D. Hilbert. Über die stetige abbildung einer linie auf ein flachenstück. *Math. Annalen*, 38:459–460, 1891.

[99] E. Im and K. A. Yelick. Optimizing sparse matrix-vector multiplication for register reuse. In *International Conference on Computational Science*, May 2001.

[100] Intel. Intel(R) Itanium(R) 2 processor reference manual for software development and optimization, 2004.

[101] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, 1988. ACM Press.

[102] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. In *ICCS 2002,LNCS2330*, pages 335–344. Springer-Verlag, April 2002.

[103] ISO/IEC 9945-1:1996. [ANSI/IEEE Std 1003.1, 1996 Edition] Information technology - Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language].

[104] Bo Kågström, Per Ling, and Charles van Loan. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302, 1998.

[105] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, 1994.

[106] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*, September 1998.

[107] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[108] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 20(4):869–916, 1998.

[109] T. Kisuki, P.M.W. Knijnenburg, and M.F.P O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.

[110] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The high performance Fortran handbook*. Scientific and Engineering Computation. MIT Press, 1994.

[111] Roger Koenker and Pin Ng. SparseM: A Sparse Matrix Package for R, 2003. http://cran.r-project.org/src/contrib/PACKAGES.html#SparseM.

[112] T. Kohonen. The self-organizing map, Proc. of the IEEE 78(9):1464–1480, 1990.

[113] M. Kudo, N. Masuyamaa, J. Toyamaa, and M. Shimbob. Simple termination conditions for k-nearest neighbor method, Pattern Recognition Letters, 24(9-10):1203–1213, June 2003.

[114] M. Lam. Software Pipelining: An Effective Technique for VLIW Machines, Proc. of the SIGPLAN'88, pp 318–328.

[115] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS'91*, pages 67–74, 1991.

[116] E.W. Lee and S.I. Chae. Fast Design of Reduced-Complexity Nearest-Neighbor Classifiers Using Triangular Inequality, IEEE Trans. on Pattern Analysis and Machine Intelligence, 20(5):562–566, 1998.

[117] Y. Lee and J. Orlin. GRIDGEN generator., 1991. Code available from ftp://dimacs.rutgers.edu in directory pub/netflow/generators/network/gridgen.

[118] Jin Li, Anthony Skjellum, and Robert D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency - Practice and Experience*, 9(5):345–389, 1997.

[119] Andrew Lih and Erez Zadok. PGMAKE: A portable distributed make system. Technical Report CUCS-035-94, Computer Science Department, Columbia University, May 1994.

[120] Cheng-Lin Liu, H. Sako, and H. Fujisawa. Performance evaluation of pattern classifiers for handwritten character recognition. *International Journal on Document Analysis and Recognition*, 4:191–204, 2002.

[121] J. H. W. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[122] J. W. Liu, E. G. Ng, and B. W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(1):242–252, January 1993.

[123] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

[124] J. W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34:82–109, 1992.

[125] Mary E. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[126] D. MacKenzie and B. Elliston. *Autoconf: Creating Automatic Configuration Scripts., User Manual, Edition 2.12, for Autoconf version 2.12. Free Software Foundation*, December 1998.

[127] D. MacKenzie and T. Tromey. *GNU Automake, User Manual, for Automake version 1.4, Free Software Foundation,*, April 1999.

[128] A. C. McKellar and Jr. E. G. Coffman. Organizing matrices and matrix operations for paged memory systems. *Communications of the ACM*, 12(3):153–165, 1969.

[129] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing*, pages 425–433. ACM Press, 1999.

[130] M. F. Mokbel, W. G. Aref, and I. Kamel. Analysis of multi-dimensional space-filling curves. *GeoInformatica*, 7(3):179–209, September 2003.

[131] G. M. Morton. A computer-oriented geodetic data base and a new technique in file sequencing. IBM Ltd. Ottawa, Canada, 1966.

[132] Wahid Nasri and Denis Trystram. A poly-algorithmic approach applied for fast matrix multiplication on clusters. In *IPDPS*, pages 234–241. IEEE Computer Society, 2004.

[133] J.J. Navarro, A. Juan, and T. Lang. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Computations, ACM Int. Conf. Supercomputing, 1994, pp. 354–363.

[134] Juan J. Navarro, E. García, and José R. Herrero. Data prefetching and multilevel blocking for linear algebra operations. In *Proceedings of the 10th international conference on Supercomputing*, pages 109–116. ACM Press, May 1996.

[135] Juan J. Navarro, Antonio Juan, and Tomas Lang. MOB forms: A class of Multilevel Block Algorithms for dense linear algebra operations. In *Proceedings of the 8th International Conference on Supercomputing*. ACM Press, 1994.

[136] NetLib. Linear programming problems. http://www.netlib.org/lp/.

[137] Esmond G. Ng and Barry W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14(5):1034–1056, 1993.

[138] A. Noor and S. Voigt. Hypermatrix scheme for the STAR–100 computer. *Comp. & Struct.*, 5:287–296, 1975.

[139] OpenMP. URL. http://www.openmp.org.

[140] A. Oram and Steve Talbott. *Managing Projects with Make*. O'Reilly & Associates, Inc., second edition, 1991.

[141] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 181–190, New York, NY, USA, 1984. ACM Press.

[142] Hewlett Packard. PA-RISC 1.1 Architecture and Instruction Set Reference Manual, 1994.

[143] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, 1999.

[144] PARASOL. (EU ESPRIT IV LTR Project No. 20160).

[145] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Trans. Parallel and Distrib. Systems*, 14(7):640–654, 2003.

[146] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Math. Annalen*, pages 157–160, 1890.

[147] J.R. Pilkington and S.B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, March 1996.

[148] PTR. Parallel tools consortium working group on portable timing routines.

[149] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[150] F. Ricci and P. Avesani. Data Compression and Local Metrics for Nearest Neighbor Classification, IEEE Trans. on Pattern Analysis and Machine Intelligence, 21(4):380–384, 1999.

[151] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. *LNCS*, 1575:168–182, 1999.

[152] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, 1998.

[153] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. University of Maryland.

[154] Edward Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Sci. Comput.*, 17(3):699–713, 1996.

[155] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Sci. Comput.*, 15(6):1413–1439, November 1994.

[156] Hans Sagan, editor. *Space-Filling Curves*. Springer-Verlag, 1994.

[157] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[158] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the SUN Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130, Portland, Oregon, June 1985. Usenix Association.

[159] Jeremy G. Siek and Andrew Lumsdaine. A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library. In *Object-Oriented Technology, ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 468–469. Springer, 1998.

[160] SSE2. Streaming SIMD Extensions 2 for the Pentium 4 processor. http://www.intel.com/software/products/college/ia32/sse2.

[161] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directing Recompilation, for GNU Make Version 3.79.1.* GNU Press, 2002.

[162] ST-ORM User's Manual, 1999. EASi Engineering GmbH.

[163] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Supercomputing*, pages 410–419, 1993.

[164] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.

[165] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–839, August 2002.

[166] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, California, 3rd edition, July 2000.

[167] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 211–217. IEEE Computer Society, Nov 1998.

[168] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw, Pract. Exper*, 35(2):101–121, February 2005.

[169] David S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20(4):195–199, May 1985.

[170] David S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In *Euro-Par 2000,LNCS1900*, pages 774–783, September 2000.

[171] David S. Wise and Jeremy D. Frens. Morton-order matrices deserve compilers' support. Technical Report TR 533, Computer Science Department, Indiana University, 1999.

[172] M. Wolfe. More iteration space tiling. In ACM, editor, *Proceedings, Supercomputing '89: November 13–17, 1989, Reno, Nevada*, pages 655–664. ACM Press, 1989.

[173] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36. ACM Press, 1995.

[174] Yin Zhang. Solving large-scale linear programs by interior-point methods under the MATLAB environment. *Optim. Methods Softw.*, 10(1):1–31, 1998.