Computing Explicit Matrix Inverses by Recursion

Lars Karlsson

February 15, 2006
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Robert Granat
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Making the most of the available resources is the ultimate goal of any optimization. For computers this often means optimization with respect to amount of work per time unit.

Numerical linear algebra algorithms are usually built on highly optimized standard subprograms, e.g. level 1-3 BLAS. The development of deep memory hierarchies spawned interest in reorganizing algorithms to make better use of operations with high operation to area ratio. During the last decade, recursion has been applied as a means of blocking such algorithms. The result is portable performance through automatic variable blocking for an arbitrary number of memory layers.

This report has applied the recursive approach to matrix inversion. LAPACK–style algorithms were developed for triangular and square matrix inversion. A custom recursive kernel was demonstrated to be superior to the LAPACK level 2 kernel on modern processors, typically with a speedup of two. Both the triangular and square inversion algorithms showed consistent, increasing, and portable performance outperforming LAPACK for large matrices.

Parallel algorithms using the shared memory paradigm were developed for two of the three major stages in the square inversion algorithm. The task parallelism available in triangular inversion together with large BLAS operations that have inherent data parallelism gave the algorithms good efficiency and scalability. Furthermore, explicit data copying was used to induce task parallelism in the other implemented major step. This typically increased the performance for small matrices, but for large matrices the geometric nature of the work distribution in the recursive algorithm gave good performance even without the extra parallelism gained by explicit copying.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This chapter gives a short history of the matrix concept and the matrix inverse. Chapter 2 discusses recursive algorithms, Chapter 3 introduces deep memory hierarchies. Chapter 4 presents some known methods of inversion, and this chapter is followed by the presentation of a recursive triangular inversion algorithm and its SMP parallelization in Chapters 5 and 6. Chapter 7 presents a recursive algorithm, and an SMP parallelization of it, that multiplies an upper triangular matrix with a lower triangular matrix. That multiplication is one of the major steps in the method used in this report for inverting a dense square matrix. Some performance and accuracy results are presented in Chapter 8. Finally, Chapter 9 gives some conclusions of the work presented.

1.1 History of Matrices

Matrix theory, which is today considered one of the first high level mathematical concepts taught at universities, has a surprisingly short history. It took centuries of studies of determinants, linear transformations, and other matrix related fields until the concept of a matrix, as it is known today, was born. Arthur Cayley studied linear transformations, and in the 1850s introduced the matrix to simplify his notation [21]. This had been done before of course; there was nothing new with arranging numbers in a rectangular array. J.J. Sylvester was the first to use the term matrix to refer to this kind of arrangement [10]. Sylvester and Cayley were friends, and Cayley made use of Sylvester's terminology in his papers from 1855 and 1858 [20]. In the latter paper, he introduced single-letter notation for matrices. He defined the basic arithmetic operators $+$, $-$, $\times$, the identity matrix and the inverse, among other things. Cayley took the definitions of the operators directly from related operations in linear transformation theory. Most notably, multiplication is the equivalent of combining two linear transformations.

The decades following Cayley's definition of the linear algebra saw some additional developments in the field, but matrix theory received its current status in the 20th century. The birth of machine computation; the invention of the computer radically increased the interest in matrices. They suddenly became the prevailing abstraction, perhaps because of the easy representation in computer memory. It was the study of determinants and linear transformations that led to the development of the matrix concept, but in modern high education the matrix is primary and is used to introduce determinants and linear transformations. For example, in "Elementary Linear Algebra" [4] Chapter 1 deals with matrices, Chapter 2 with determinants, Chapter 7 and 8 with

eigenvalues and linear transformations respectively.

1.2 Theory of Matrix Inverses

Given a square matrix A, then a square matrix B with the property that $BA = AB = I$, where I is the identity matrix of appropriate size, is called the *inverse* of A and is denoted by A^{-1}. A matrix A is called *invertible* if such a B exists. Moreover, the inverse of a matrix is unique. Invertible matrices are called *non-singular* as opposed to non-invertible matrices which are called *singular*. A number of statements are equivalent to A being invertible. An important subset of these are:

- There is a square matrix B such that $AB = BA = I$
- A^T is invertible
- The determinant is nonzero, $det(A) \neq 0$
- A has linearly independent rows, $rank(A) = n$
- $Ax = 0$ has only the trivial solution $x = 0$
- Zero is not an eigenvalue of A

The inverse of upper (lower) triangular matrices are upper (lower). If a triangular matrix has unit diagonal, the same holds for the inverse [8].

It turns out that although the matrix inverse, and invertibility, is very important theoretical concepts, the explicit inverse does not have many applications in practice. To take just one example, the solution to $Ax = b$ is $x = A^{-1}b$ but is not numerically computed this way because of both excessive computations and numerical instability. Instead, the equation is transformed using Gaussian elimination. This has the effect of applying the inverse of A to b without the need to form the inverse explicitly. However, Sun et al have a nice set of references in [24] citing applications such as statistics and stable subspace computation in control theory. Higham [15] also enumerates some applications.

1.3 Numerical Libraries

Linear algebra algorithms can usually be factored into a number of basic operations, such as vector scaling, dot product, outer product, matrix multiplication, and so on. These operations are easy to describe and has many usages, but the efficient implementation of them is largely dependent on the machine they are to be run on. For this purpose the Basic Linear Algebra Subprograms, BLAS, were defined to create a portable interface to these implementations. The LAPACK library is a set of implementations of more complex linear algebra problems ranging from eigenvalue problems to matrix inversion. LAPACK uses BLAS to do the bulk of the work, thus achieving good performance wherever an efficient BLAS implementation is present. It also blocks data to make extensive use of the memory efficient level 3 BLAS, i.e., matrix–matrix operations.

The BLAS library is divided into three levels based on the asymptotic time complexity of the routines. There are three levels: 1, 2, and 3, which correspond to vector–vector operations, matrix–vector operations, and matrix–matrix operations, respectively. Out of these, the third level is extra important because the number of entries is $\mathcal{O}(N^2)$,

whilst the number of flops performed is $\mathcal{O}(N^3)$. This difference means that there is a possibility to reuse matrix entries, and this possibility is far better for level 3 compared to level 1 and 2 BLAS.

Besides BLAS implementations from machine vendors, there exists a number of independent implementations. Most notable among those is the ATLAS project. It uses a similar approach to the `GEMM`-based BLAS implementation of Kågström, Ling, and van Loan [12], in which they built all BLAS operations on one high performance general matrix multiply and add, `GEMM`, implementation. The ATLAS project uses recursion and a `GEMM` kernel that is optimized by experiments during the installation process to produce a BLAS implementation with good performance on a wide range of architectures [26]. Another popular implementation is the gotoBLAS library by Kazushige Goto. It also supports multiple architectures, but it focuses on minimizing TLB misses and makes use of assembler kernels [9].

Chapter 2

Recursion and Divide and Conquer

A function in a programming language is called recursive if it computes the result by calls to itself. Two functions are called mutually recursive if they call each other. The programming language is required to store the details of each call, termed an *activation record*, for recursion to be possible. In many early languages, FORTRAN in particular, this was simply not possible, either because it was not wanted or because it was not possible because the activation record was stored at a particular memory location, thus prohibiting recursive calls.

ALGOL60 is probably the first programming language to include recursive function definitions. Derivatives of this language, like C and Pascal, also have this functionality. FORTRAN, however, did not include recursion until the FORTRAN90 standard. But then there was already a huge amount of legacy code, and compilers for FORTRAN77 must have been more widely deployed.

It must be mentioned that recursion can always be emulated by an iterative function using a stack to emulate the activation records, a method employed by Gustavson in [13]. The overhead of an actual call compared to emulation via a stack is thought to be very small, hardly justifying conversion to iteration for the sake of performance.

Two types of recursion can be emulated without a stack. *Tail recursion* is used to denote a function with only one recursive call as the very last statement. The transformation is straightforward: simply replace the call with a loop that at the end resets the input parameters to emulate the call. The closely related type with the recursive call at the very beginning of the function has no known name since it generally cannot be transformed. It can be transformed if the input parameters can be computed in reverse order. The transformation is similar: start with the base-case input, place a loop around the body and at the end of it compute the preceding set of input parameters.

2.1 Divide and Conquer

There are many paradigms in algorithm design. Backtracking, dynamic programming, and the greedy method to name a few. One compelling type of algorithm is called Divide and Conquer, or D&C for short. Algorithms of this type split the problem into subproblems. After the sub-solutions are found they are combined to form the solution

to the bigger problem [16]. The problem is thus broken down into smaller, and therefore in some sense, easier problems.

When the subproblems are of the same type as the original problem, the same *recursive* process can be carried out until the problem size is sufficiently small. This special type of D&C is referred to as D&C recursion.

Recursive execution is usually described using the notion of a recursion tree. Each node in such a tree represents one application of the recursive function, and its children are the recursive calls spawned by it. See Figure 2.1 for an example. The number in each

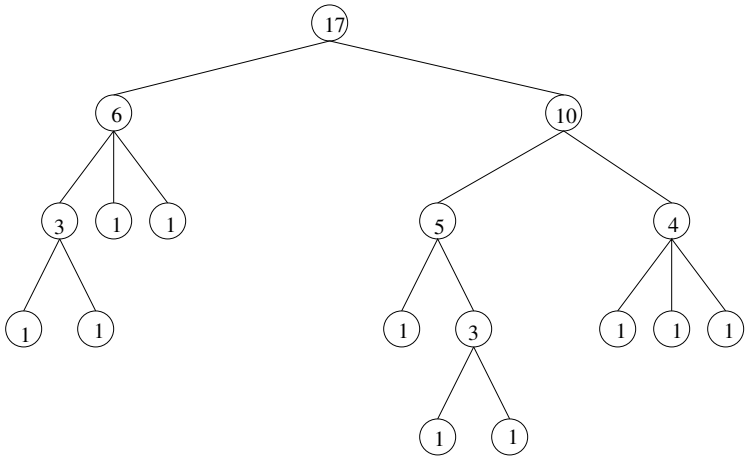


Figure 2.1: Example recursion tree

node is the amount of work associated with it. The three most important characteristics of a recursion tree (and therefore of the recursive algorithm) is its *width*, *height*, and *work distribution*. The width is usually summarized by a limit on the branching factor (in the example figure it is three), with the common limit of 2 yielding the class binary recursion trees. The height is the number of levels in the tree, which in this example is five. The height and width together quantify the overhead of making recursive calls, because they set a limit on the number of calls made. The work distribution indicates how much of the work is done at various levels of the tree, and therefore determines the importance of a high performance base-case solver implementation. The more amount of work done deep down the tree, the more important a high performance base-case solver becomes. In Figure 2.1, the total amount of work is 58 of which more than half belongs to the first two levels.

2.1.1 Time Complexity of Recursive D&C Algorithms

The recursive nature of many D&C algorithms makes it easy to express their time complexity as recurrences. Consider a D&C algorithm working on input size N. It divides its input into a subproblems of size N/b. The dividing and conquering takes time $f(N)$. The base-case is $N = 1$ and is solved in constant time. The time complexity of this class of algorithms can be expressed as

$$T(N) = \begin{cases} \mathcal{O}(1) & \text{if } N = 1, \\ aT(N/b) + f(N) & \text{otherwise.} \end{cases} \tag{2.1}$$

The *master theorem* for recurrences [27] can in some instances be used to give a tight asymptotic bound for the complexity.

As already mentioned, the distribution of work over the recursive tree is important in determining the importance of an efficient base-case implementation. The number of leaves in the recursion tree associated with algorithms having time complexity as in (2.1) is given by

$$a^{\log_b(N)}. \tag{2.2}$$

In the field of linear algebra, some values of a and b are very common. For example:

- $a = b = 2$ is typically found in recursive blocking of triangular matrices, resulting in $2^{log_2(N)} = N$ leaves.
- $a = 4, b = 2$ can be found in recursive blocking of rectangular matrices, resulting in $4^{log_2(N)} = N^2$ leaves.
- $a = 8, b = 2$ for the `GEMM` operation applied to square matrices, resulting in $8^{log_2(N)} = N^3$ leaves.

2.1.2 Top Heavy Recursion Trees

For the sake of future discussion, it is important to introduce the notion of a *top heavy recursion tree*. Intuitively, such a tree has a work distribution which has the bulk of its work in the topmost levels. One reasonable definition is that the work at a level is at most a fraction of the level above it. For example, the work at level $i + 1$ is at most 0.8 times the work at level i. This definition is reasonable because the first level in such a tree will contain at least a certain percentage of the total work no matter how large the input is.

One interesting question is whether top heavy recursion trees will be the result when the recursion tree has asymptotically fewer leaves than operations under reasonable assumptions.

2.2 Recursion and Linear Algebra

Gustavson et al [12] and Kågström, Ling, and van Loan [22] used the rationale of building on a few building blocks is preferable and recast the level 3 BLAS operations into mostly `GEMM` operations. This approach is also used by the ATLAS project which uses an automatically tuned `GEMM` together with recursively blocked level 3 BLAS operations to create a full set of level 3 BLAS whose performance is mostly decided by the performance of the `GEMM` kernel [26]. Below we give some examples of recursive linear algebra algorithms.

2.2.1 Recursive LU Factorization

Sivan Toledo analyzed the superiority of a recursively blocked LU factorization routine to a conventional right-looking algorithm in 1997 [25]. The same year Fred Gustavson published an analysis and implementation of a recursively blocked LU factorization routine in the style of LAPACK `GETRF` [13]. In this paper, he demonstrated the variable and "squarish" blocking induced by the recursion. By summing up the areas of the BLAS operands it was demonstrated that the recursive blocking had substantially smaller area

than the conventional fixed blocking. This is important because every operand must be brought to the processor at least once. Small BLAS area means more density in flops which in turn gives a bigger potential for reuse. So it can be argued, as Gustavson does, that smaller area is synonymous with a superior algorithm from a memory utilization point of view. The implementation must be able to take advantage of the theoretical increase in available reuse for the claim to be valid.

Gustavson went on to describe the geometric nature of the work distribution in the recursion tree. The work is dominated by a few operations high in the tree. The number of flops is shown to be reduced by a factor $r > 1/2$ for each level. This results in a dramatic imbalance in work resulting in the relative insignificance of what happens far down the tree. Gustavson demonstrated that a pure recursive LU factorization algorithm could rival even the highly tuned algorithm `GEF` in ESSL [13]. He noticed, though, that for optimal performance recursion should be combined with a blocking parameter so that small cases, typically those that fit in L1 cache, are solved faster.

2.2.2 Recursive Sylvester–type Matrix Equation Solver

Recursion has also been applied to Sylvester and Lyaponov equations in papers by Jonsson and Kågström [18, 19]. For one-sided equations the recursive algorithms perform the same number of flops. However, for the two-sided equations the algorithms needed more flops and more workspace.

The fast kernels that were developed together with the `GEMM`–operations induced by the recursive blocking, made the algorithms fast on uni-processor systems, including a ten-fold speedup, and showed speedups of around 2.8 on a four processor SMP system using SMP BLAS and task parallelism via OpenMP where possible. Even considering the larger amount of flops for two-sided equations, the recursive algorithm out-performed conventional algorithms for large enough problems [18].

2.2.3 Recursive Cholesky Factorization

Jonsson and Gustavson presented a Cholesky factorization routine based on recursion [14] in 2000. What is remarkable in this case is that it worked on packed format, transforming it into a recursive data layout. This made it possible for the recursive algorithm to use level 3 BLAS operations instead of level 2 for the conventional packed algorithms. As reported in their paper, the performance of their packed routine rivals even the level 3 unpacked LAPACK routine, even when including transformation to and from the recursive layout. This makes the routine practical as a portable substitute for the slower packed LAPACK routine.

2.2.4 Recursive QR Factorization

QR factorization has also been the subject of study for recursive algorithm developers. Elmroth and Gustavson implemented serial and SMP parallel algorithms for QR factorization based on recursive blocking [6]. Their dynamic load balancing scheme showed high efficiency on a four processor SMP machine. Even though the algorithm used more flops than conventional algorithms as found in LAPACK the performance was systematically better.

Chapter 3

Deep Memory Hierarchies

Since the dawn of computers, the speed in which data is read or written have been insignificant compared to the overall execution speed. The limiting factor has usually been the speed of the processor. Since a couple of decades, the gap between computation and memory speed has been growing in favor of computation. This has meant that in some areas the focus has shifted from optimizing away computational overhead to optimizing the usage of the memory hierarchy. This is particularly true for numerical linear algebra which are data intensive and only a few operations per data element.

3.1 Technologies

It is a fact of life that you get what you pay for. In terms of memory this means that fast memory is expensive and slow memory is cheap. It is thus expensive to have large amounts of fast memory.

During the years a number of technologies have been developed to create the illusion that the programmer has access to a large and fast memory, while in fact the bulk of the memory is relatively slow. A *cache* is a small and fast memory between the processor and the rest of the memory. Data is loaded into the cache when it is needed and unloaded when it does not fit anymore because other data is being used. A typical cache memory does not only load the referenced memory cell, but a consecutive number of them called a *cache line*.

The *memory hierarchy* is a term used to denote the hierarchical architecture used in the memory subsystem. The bottom is made up of cheap large memory with slow access time usually called main memory. Above this memory one might find one or more levels of cache, and finally the registers. The memory hierarchy is important because data is accessed through it, so that data not present at the top must be brought there through the underlying layers. The more layers, the deeper the memory hierarchy, and the more important a good utilization of it becomes.

Another invention called *virtual memory* solved two problems at once. It made it possible to write programs that could be loaded at any address and it also made it possible to create the illusion that a system has more main memory than it actually has by swapping memory pages to and from disk.

Virtual memory introduces a separate address space which programs use to reference data and code. This address space is mapped by the operating system to actual memory addresses. The mapping is done in chunks called *pages*. For practical reasons the page

size is quite large, several kilobytes. The mapping also makes it possible to store the requested memory address on disk and load it when needed. This creates the illusion of virtually unlimited main memory.

A system with virtual memory typically has a *translation look-aside buffer*, TLB, that stores recent mappings. This creates another level in the memory hierarchy in which spatial and temporal data locality is important (see below). The page size in virtual memory systems is typically $4KB$ or bigger, while the number of entries in a TLB is typically not more than a couple of hundreds.

3.2 Locality

When discussing memory hierarchies the issue of *reuse* is central. Once data is brought high up in the hierarchy it should remain there and be reused as much as possible before being pushed down again.

The functionality of the cache gives rise to two important notions of locality. Because data will occasionally be unloaded, all data brought into the cache should be used again in the near future (*temporal locality*). The existence of cache lines and pages favors the use of nearby memory cells which are brought in "for free" (*spatial locality*).

Each algorithm has a specific *memory reference pattern*. This is the order in which memory cells are read and written. Since the technologies discussed above favors algorithms with good spatial and temporal locality, algorithms with those properties will in general utilize a deep memory hierarchy more efficiently.

3.3 Blocking

Numerical linear algebra algorithms operate on matrices, vectors, and scalars. Matrices are two-dimensional while the memory addressing of the underlying machine is one-dimensional. This requires a mapping which historically has been done in either row-major or column-major format. The first variant stores each row consecutively (typified by C), the latter each column consecutively (typified by FORTRAN). The rest of this report will assume the FORTRAN layout if nothing else is explicitly stated.

It turns out that most algorithms in linear algebra can be reformulated to work on matrix blocks [8]. Blocking groups data into smaller regions. This improves the spatial locality which is important, particularly for the TLB and L1 cache. To match L2, or even L3, cache requires at least two levels of blocking. This can be a tedious task and the result will be highly architecture dependent.

3.4 Recursive Blocking

Recursive blocking is another way to partition a matrix into blocks. With regular blocking, all blocks are of similar sizes. With multiple levels of blocking, there are a fixed number of similar block sizes. Recursive blocking gives an automatic multilevel blocking in which the number of levels as well as the block sizes are not fixed but instead depend on the problem sizes [5]. Recursive blocking partitions the matrix (or matrices) into a small number of large blocks. This partitioning then dictates what operations need to be done on them. In many cases, some sub-operations are of the same type as the main problem which means that blocks may be recursively blocked again. A description

of the way in which the recursive partitioning is performed is typically referred to as a *recursive template*.

Recursive blocking has many advantages. For example, it has the potential of matching every level of a deep memory hierarchy with regards to temporal locality. This is because the operations performed in the recursive application on one sub-block use data in that block only. This groups the operations, with respect to time, to concentrated areas in the input matrix (or matrices). It also keeps the operands "squarish" [13], which maximizes the ratio of flops to operand area with the potential of reducing movement through the memory hierarchy. When the work distribution (see Section 2.1) is top heavy the importance of lower levels decreases quickly, making even pure recursion feasible for large enough matrices (see for example [13]). Higher performance is gained by stopping recursion a bit higher in the tree and using a highly optimized solver routine. This requires a blocking parameter, but it only affects the lowest level of the tree, keeping all other operations oblivious to the blocking parameter. This is in stark contrast to the fixed blocking approach where the blocking parameter affects the dimensions of almost every operand. The importance of picking a good blocking parameter is thus much lower for top heavy recursive algorithms.

Gustavson emphasized the geometric nature of the recursive algorithms in his paper on LU factorization [13]. He observed that the number of flops at each level in the recursion corresponds to a geometric series with varying ratio. This is in contrast with conventional fixed blocked algorithms in which the flops at each iteration either increases or decreases linearly. This will be discussed further with regards to triangular inversion, which is the main subject of this report.

3.5 Recursive Data Layout

Although good temporal locality throughout deep memory hierarchies is provided automatically with recursive blocking, it does not in general give good spatial locality. In modern systems spatial locality is important in caches due to cache lines and in the TLB due to pages. Cache lines are typically very small, a few double words[1] in size. The pages, however, are very large, $4KB$ or 512 double words or even more.

Substantial speedup potential in several applications by using other data layouts has been observed. This will increase the data locality, mainly in the TLB. It has long been observed that storing data in a blocked format improves data locality. It has even been shown to be intrinsically superior to the common row or column formats with regards to TLB performance [23]. A recursive data layout described in [11] uses a layout which matches how the algorithm later works on the data. It is believed to be a data layout that is close to optimal for those algorithms. In some cases, the resulting algorithm is faster than conventional algorithms even when transformation to and from the recursive layout is included.

One of the most impressive results was achieved by Gustavson and Jonsson in [14], where they used conversion from triangular packed storage to a recursive triangular storage and thereby created an opportunity to use level 3 BLAS instead of only level 2 operations as the conventional storage imposes.

[1] A double word in this context is two 32 bit words, or 8 bytes

Chapter 4

Methods of Inversion

The methods used to invert a square matrix can be divided into two types: iterative and direct. In this report, we only consider the latter type. The direct methods for triangular matrices all have a flop count of order $N^3/3$. The flop count for dense square matrices is $2N^3$.

4.1 Methods for Inverting Triangular Matrices

Unblocked methods for triangular inversion are based on *successive bordering*; computing either a column or a row of the inverse at a time.

For a lower triangular matrix L, Algorithm 1 gives a procedure for computing the inverse one column at a time from left to right. It successively partitions the remaining part of L into a, $\boldsymbol{b}$, and C, as described in Figure 4.1. The white columns in the Figure are the already computed columns of L^{-1}. By using Equation (5.1) developed later it is easy to see that

$$C\boldsymbol{x} = -a\boldsymbol{b}$$

solved for x computes the $\boldsymbol{b}$ part of the i:th column in the inverse. This scheme is employed in Algorithm 1.

Algorithm 1 Unblocked inversion

```
1: for i = 1 : N do
2:    Partition L as in Figure 4.1      % Successive bordering
3:    a = 1/a                           % Invert diagonal entry
4:    b = −ab                           % Update RHS
5:    Solve Cx = b                      % Solve for x, and overwrite b with x,
6:                                      % using forward substitution
7: end for
```

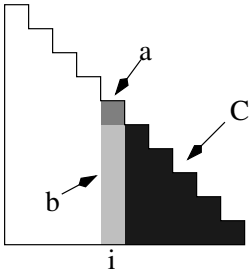


Figure 4.1: Successive bordering snapshot for the i:th iteration

4.2 Methods for Inverting a Square Matrix

4.2.1 By Elementary Row Operations

An invertible matrix A can be transformed into the identity matrix by a series of *elementary row operations* [4]. There are three types of elementary row operations:

1. Interchanging two rows
2. Adding a multiple of one row to another
3. Scaling by a nonzero constant

An *elementary matrix* is any matrix that results by applying one elementary row operation to the identity matrix I [4]. Such matrices are commonly denoted by E, and multiplying such a matrix on the left has the same effect as to apply the elementary row operation directly. It can be shown that any matrix can be represented as a finite product of elementary matrices [4]. Let $B = E_k \cdots E_2 E_1$ such that $BA = E_k \cdots E_2 E_1 A = I$. This implies that $A^{-1} = E_k \cdots E_2 E_1 I$. Thus, by applying the operations that transforms A into I to the identity matrix produces the inverse of A. This is the theory for the common manual algorithm for inversion described in Algorithm 2.

Algorithm 2 Manual inversion

Require: A is $N \times N$

1: Form the augmented matrix $[A|I]$
2: Apply elementary row operations to $[A|I]$ until it has the form $[I|B]$
3: If that is impossible then A is singular
4: Otherwise, B is the inverse of A

4.2.2 By LU Factorization

A technique employed by both the LAPACK `GETRI` routine and the MATLAB `inv` function is the four step approach described as Algorithm 3.

An LU factorization of A with partial pivoting is an LU factorization of the matrix PA, where P is a *permutation matrix*[1]. PA is similar to A as it contains the rows of A but possibly in a different order. PA is invertible if and only if A is invertible and

[1] A permutation matrix is any matrix formed by applying row interchanges to I.

Algorithm 3 Inversion by LU factorization

1: Factorize A as $PA = LU$
2: Invert U by triangular inversion
3: Solve $(PA)^{-1}L = U^{-1}$ for $(PA)^{-1}$
4: Unscramble the columns of the computed $(PA)^{-1} = A^{-1}P^{-1}$ to find A^{-1}

$PA = LU \Rightarrow (PA)^{-1} = U^{-1}L^{-1}$. The inverse to PA can be computed first, and then the permutation is reversed to find A^{-1}. The reversal is performed by swapping columns since the permutation in $(PA)^{-1} = A^{-1}P^{-1}$ is applied from the right.

4.3 Gauss–Jordan Elimination

Gauss–Jordan Elimination (GJE) is similar to the LU approach but performs the four steps combined so that the algorithm computes the inverse of A one column at a time. GJE is often described as an algorithm that solves matrix equations. Finding the inverse is simply a special case since it requires finding the solution to $AX = I$. The sparsity structure of I can be handled to avoid computations with zeroes. The computational cost of GJE is approximately the same as for the LU approach. It can also be computed *in situ*[2]. Algorithms with and without partial pivoting were presented and evaluated in [24]. One obvious drawback of GJE for uniprocessors is that at each iteration the GJE algorithm sweeps through the entire matrix. Even though blocked versions exist the situation is the same. This does not mean that the algorithm is useless, only that it is probably not suitable for deep memory hierarchies.

[2]In situ, or in–place, is a term used to denote algorithms which do not need auxiliary storage. The term is fuzzy because most algorithms needs loop variables or a small stack for recursive calls, but most often it is clear which algorithms that can be said to have this property.

Chapter 5

Recursive Triangular Inversion

This chapter presents a recursive approach to triangular inversion. It introduces a new super-scalar kernel to avoid the overhead of repeated level 2 BLAS calls.

5.1 Recursive Inversion of Triangular Matrices

The inverse of an upper or lower triangular matrix is itself upper or lower triangular, respectively [4]. Thus, if A is a lower triangular matrix partitioned as

$$A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$$

then let B be a matrix of equal dimensions and partitioned as A. The equation $AB = I$ holds if and only if A is invertible and then $A^{-1} = B$ [4]. The equation yields the following block equations:

$$\begin{aligned} &A_{11}B_{11} = I \Rightarrow B_{11} = A_{11}^{-1} \\ &A_{22}B_{22} = I \Rightarrow B_{22} = A_{22}^{-1} \\ &A_{21}B_{11} + A_{22}B_{21} = 0 \Rightarrow B_{21} = -A_{22}^{-1}A_{21}A_{11}^{-1}. \end{aligned}$$

In block form, this is expressed as

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & \\ -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{bmatrix}. \tag{5.1}$$

The analogous result for upper triangular matrices is

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ & A_{22}^{-1} \end{bmatrix}.$$

5.1.1 Recursive Algorithm

The above facts lead to a recursively blocked algorithm where the diagonal blocks are inverted recursively and combined with the rectangular block to form the full inverse. For upper triangular matrices the rectangular block B_{12} of A^{-1} is determined by the

equation $B_{12} = -A_{11}^{-1} A_{12} A_{22}^{-1}$. This form of the equation suggests two triangular matrix multiply, `TRMM`, operations. There are other arrangements possible:

$$\begin{aligned} B_{12} A_{22} &= -A_{11}^{-1} A_{12} \\ A_{11} B_{12} &= -A_{12} A_{22}^{-1} \\ A_{11} \left(B_{12} A_{22} \right) &= -A_{12}. \end{aligned}$$

The first and second suggests one `TRMM` followed by one triangular solve with multiple right-hand sides, `TRSM`, operation, the third uses two `TRSM`s to compute the rectangular block of the inverse. There are even more possible arrangements, but these are the ones considered in this report. The double `TRMM` method is used throughout for performance measurement, whereas all four methods are compared later in terms of accuracy.

Let the size of the rectangular block A_{12} be $M \times N$. The two `TRMM` operations in the original form of the equation have a combined flop count of $NM^2 + MN^2$, which in the special case of $M = N$ is $2N^3$.

Algorithm 4 gives a pure recursive procedure that inverts a lower triangular matrix.

Algorithm 4 RECTRTRI(A)

Require: A is $N \times N$ and lower triangular
1: **if** $N = 1$ **then**
2: A is scalar so $A := 1/A$
3: **else**
4: Partition $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$
5: RECTRTRI(A_{11}) % Invert A_{11}
6: RECTRTRI(A_{22}) % Invert A_{22}
7: $A_{21} := A_{21} \times A_{11}$ % `TRMM` operation
8: $A_{21} := -A_{22} \times A_{21}$ % `TRMM` operation
9: **end if**

It is generally a good idea to split the input matrix in half because that makes most headway towards the base-case $N = 1$ and thus yields the shallowest recursion tree which minimizes the recursion overhead. It also gives other positive effects on the performance, such as producing BLAS operations with a high flop over area ratio. When the dimension is even, a cut in half is straightforward. When the dimension is odd, one diagonal block will be of one higher dimension than the other. There is a need to establish a bound as to how skewed the recursion tree will be if a policy of always assigning the higher dimensionality to the right block is used.

THEOREM 1: The sizes of the nodes at a level differ by at most one when N is even and split into $m = n = N/2$ and when N is odd is split into $m = \lfloor N/2 \rfloor$ and $n = \lceil N/2 \rceil$.

Proof. The proof is by induction on the depth k. For $k = 0$ there is only one node, so the statement trivially holds. Assume that the statement holds for $k \geq 0$. Let the smallest of the dimensions at level k be N. The nodes at level $k+1$ that get generated are either of dimension $N/2$ if N is even, or $\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor + 1$ if N is odd. If there is another dimension at level k it must be $N + 1$ because of the assumption. These nodes generate dimensions of size $(N+1)/2 = \lfloor N/2 \rfloor + 1$ if N is odd, and $\lfloor (N+1)/2 \rfloor = N/2$

and $\lfloor (N+1)/2 \rfloor + 1 = N/2 + 1$ if N is even. Thus, the four dimensions that can be generated for level $k+1$ are in fact only two different ones: $\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor + 1$, which differ by at most one. Thus, the statement is true for depth $k+1$ if it is true for depth k. By induction, the statement holds for all $k \geq 0$. □

A consequence of Theorem 1 is that the depths of all the dimension 1 nodes will differ by at most one. The tree is therefore well balanced.

5.1.2 Recursive Time Complexity

We consider the case where A is $N \times N$ and $N = 2^k$ for some integer k and derive the flop count for the recursive algorithm. The algorithm divides the input matrix into half sized blocks (since N is a power of 2). The diagonal blocks are computed recursively unless their size is 1×1 in which case the inverse is computed by a division. Let $F(N)$ be the flop count for the recursive algorithm operating on input size N. The two multiplications that make up the combination step takes $(N/2)^3$ flops each. The recursive count is determined by the recurrence relation

$$F(N) = \begin{cases} 1 & \text{if } N = 1, \\ 2F(N/2) + N^3/4 & \text{otherwise.} \end{cases} \tag{5.2}$$

The method of substitution[16] yields the closed form $F(N) = \frac{N^3 + 2N}{3}$.

For N not a power of two, the same flop count holds, which is shown in Theorem 2.

THEOREM 2: The number of flops to compute the triangular inverse of A of size $N \times N$ is $F(N) = \frac{N^3 + 2N}{3}$ regardless of splitting strategy.

Proof. The proof is by induction on N. For $N = 1$ there is only one element which is inverted by one operation. Since $F(1) = 1$ the statement is true in this case. Assume that the statement is true for all $N < k$, where $k > 1$. For $N = k$ let the matrix be split into one $m \times m$ and one $n \times n$ matrix, with $m + n = k$ and $m, n \geq 1$. The flop count for this scenario is the sum of the two triangular inversions and the two multiplications: $F(m) + F(n) + mn^2 + m^2 n = \frac{m^3 + 2m}{3} + \frac{n^3 + 2n}{3} + mn^2 + m^2 n$. Substituting $m = k - n$ and simplifying gives $\frac{k^3 + 2k}{3} = F(k)$, so the statement holds for $N = k$ as well. Thus, by induction the statement is true for all $N \geq 1$. □

5.2 Inversion Kernel Issues

For small matrices the performance penalty for making calls to BLAS routines as well as recursive calls can be substantial. For microprocessors that do not have vector computation facilities, simply implementing the inversion using nested loops can easily outperform a routine that uses level 2 BLAS for small matrices.

To achieve even higher performance, register blocking via loop unrolling is a common practice. However, the algorithms for triangular inversion that use successive bordering from left or right have many data dependencies between columns. Experiments with manual loop unrolling of those algorithms seem to confirm that these dependencies limits the potential boost of applying the register blocking technique.

5.2.1 Kernel Implementation

The installed BLAS is presumably highly tuned and high performing. However, the high performance can be substantially limited by the overheads caused by n calls to `TRMV` (triangular matrix times vector) and n calls to `SCAL` (vector times scalar). The calls also effectively take away the compiler's opportunity to exploit concurrency of the two operations performed for each column. The overheads can get so big that a straightforward triple-nested loop with compiler optimizations outperforms LAPACK's `TRTI2` (triangular inversion using level 2 BLAS) for small matrices. The three loops can be unrolled to further enhance performance. However, the data dependencies and the inherently bad flops/area ratio in the level 2 BLAS operations limit the performance achievable by this approach. The alternative is to continue with the recursion down to another cutoff, this time using a lightweight replacement of the BLAS operation `TRMM`. For the new base-case, a fully unrolled version of the three loops can be used. As shown in Figure 5.1, the performance can be two to three times that of `TRTI2`.

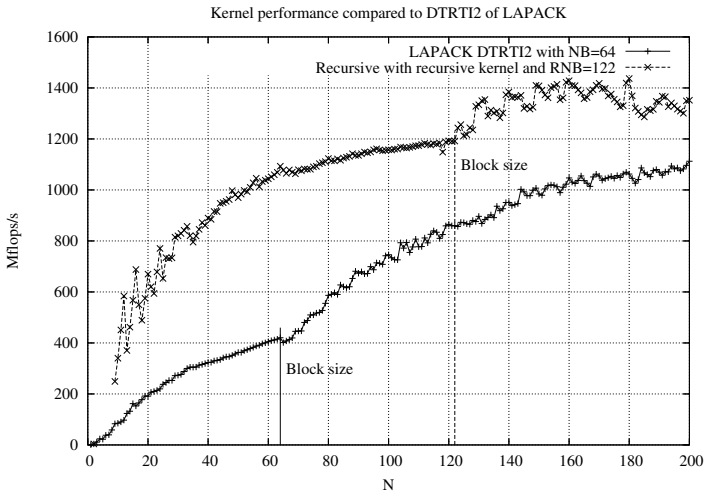


Figure 5.1: Kernel performance on an AMD Athlon XP 2200+ compared to LAPACK TRTRI using TRTI2 for $N \leq 64$.

`TRMM` Implementation

The BLAS implementations usually give a significant penalty for small matrices. It is believed that this is caused by parameter checking and branching. A penalty may also occur if the BLAS library is dynamically linked, in which case the call is directly followed by a jump to the actual code. To the author's knowledge there is no documented study of the precise cause of these penalties.

To overcome the overly slow BLAS operations for small matrices very slim `TRMM` implementations were constructed. The general `TRMM` performs one of these four operations:

$$A = \begin{cases} \alpha A op(L) \\ \alpha A op(U) \\ \alpha op(L) A \\ \alpha op(U) A \end{cases}$$

where $op(A)$ is A or A^T and L, U are unit or regular lower and upper triangular respectively.

There are eight variants used in the inversion routine variants (upper/lower, unit/non–unit, and $\alpha = \pm 1$). They were implemented using the same approach (see Figure 5.2) using 4×2 unrolling. The result is suboptimal but fairly portable performance with no overhead except the function call.

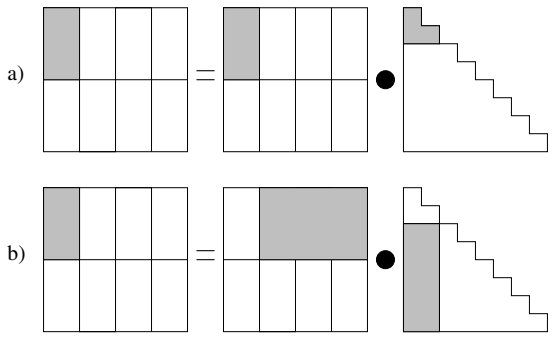


Figure 5.2: The calculation of one block consists of two steps. a) a very small `TRMM` completely unrolled, and b) a `GEMM` operation, performed as a sequence of outer products.

Leaf Implementation

For matrices of size $N \leq 8$, a completely unrolled version of Algorithm 1 is used. A program was implemented to do the job of filling in the correct indices in the correct order. The variant of the algorithm that works row by row instead of column by column was used in the kernel for lower triangular matrices. This approach allows a piece of code designed for $N = x$ to be used for every case where $N \leq x$ by branching out from the right spot. This was however not used in the final implementation because it served very little purpose when the biggest case was $N = 8$. Instead, separate pieces of code where used for every case of eight or smaller.

5.2.2 Kernel Performance Impact

For small matrices the various overheads are badly amortized over the computations and thus the performance will not be optimal. It is of interest to examine how the kernel will impact the overall performance.

Some simplifications are introduced in order to model the impact. The kernel is assumed to perform at P_K flops/s, even though it probably depends on the input size. The rest of the computations are assumed to perform at P_T flops/s, even though the same complication is true here. The kernel operates on $M \times M$ sized matrices, and for simplicity it is assumed that N is a multiple of M so that $N = kM$ for some integer k. The flops performed overall is $F = \frac{N^3 + 2N}{3}$. The kernel computes $K = k\frac{M^3 + 2M}{3} = \frac{NM^2 + 2N}{3}$ flops. The multiplications thus make up $F - K = \frac{N^3 - NM^2}{3}$ flops. The total performance can now be modeled by $P = \frac{F}{\frac{F-K}{P_T} + \frac{K}{P_K}}$. By making the kernel performance a fraction of the other performance, $P_K = fP_T$ the total performance can

be approximated by $P \approx \frac{fk^2}{fk^2 + (1-f)} P_T$. Figure 5.3 plots the ratio P/P_T for various f. A high kernel performance is always beneficial, but when the kernel performance

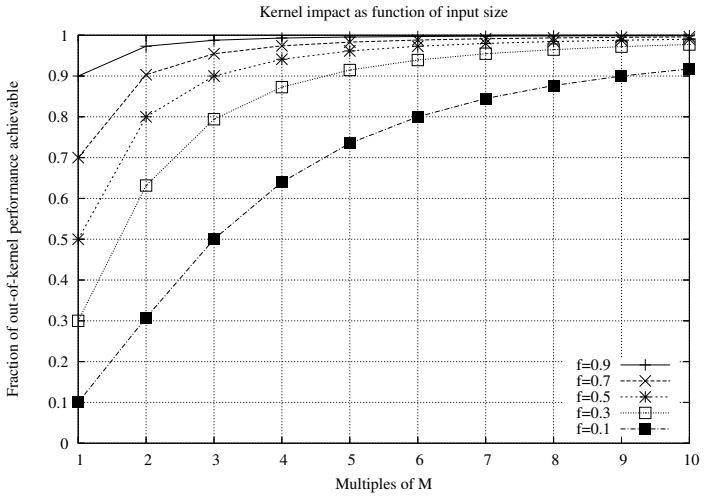


Figure 5.3: Kernel impact for various P_K/P_T ratios

reaches one half of P_T the gain of further increasing performance in the kernel is so low that it is questionable if it is worth the effort. The reduced importance around $f = 0.5$ can be seen in Figure 5.4 where the number of multiples of M needed to reduce the kernel impact to 95% of P_T is plotted against f. Around $f = 0.5$ the graph's slope is small, meaning that any further increase in f will only slightly reduce the input size needed to get to 95%, thus making it less fruitful than for $f < 0.5$.

The model predicts that for a reasonably well-performing kernel ($f > 0.5$) the overall performance will quickly overcome the slow kernel and then level out and approach the out-of-kernel performance.

An experiment was performed on an AMD Athlon XP computer using ATLAS BLAS. By testing the performance of `TRMM` on typical sizes a value of $P_T = 1600 \times 10^6$ was found to be a reasonable approximation. The kernel performance ratio f was approximated by the ratio $P_T/P_K(122)$ where $P_K(122) = 1095$ was the performance of the kernel for $N = \text{RNB} = 122$. Figure 5.5 shows the modeled performance compared to actual measurements. The model fairly accurately matches the measured performance.

5.3 Detecting Erroneous Results

A numerical method will always be subject to rounding errors because of the mapping between the infinite domain of real values to the finite domain of floating point numbers. Methods are considered stable or unstable depending on how accurate their results are. The research field called *numerical analysis* deals with this aspect of algorithms.

Apart from inaccurate results, erroneous results may also occur. These are typically produced by generating values that fall outside the representable range of floating point numbers, or by making arithmetic operations that have no reasonable representation.

One of the most common floating point standards today, the IEEE 754 binary floating

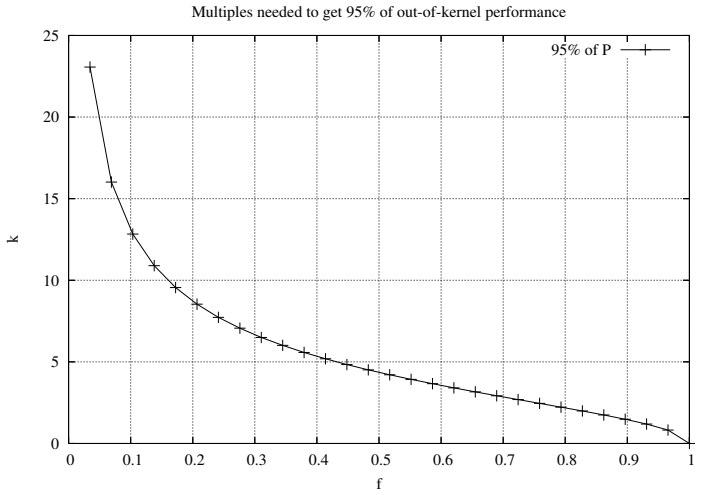


Figure 5.4: Multiples of M ($N = kM$) needed in order for performance to be 95% of P_T as a function of the kernel's performance fraction of P_T

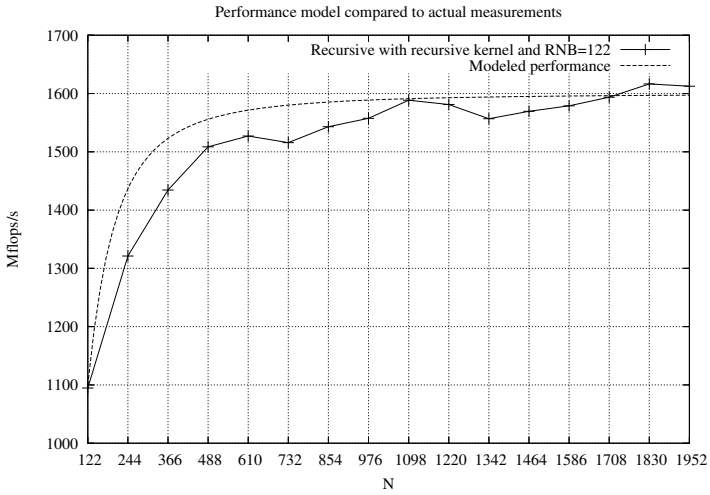


Figure 5.5: The kernel impact model vs actual performance on an AMD AthlonXP 2200+ running at 1.8GHz

point standard, defines a number of special quantities to handle such situations [7]. The default behavior of IEEE arithmetic is to produce a result and continue, even in situations of overflow and undefined operations. Two of those quantities are important in this discussion: NaN (Not a Number) and INFINITY. Table 5.1 lists some of the operations that produces NaNs and INFINITYs.

Operation	Produces NaN	Produces $\pm\infty$
$+$	$\infty + (-\infty)$	Overflow, $x + \infty$, $\infty + \infty$
$\times$	$0 \times \infty$	Overflow, $x \times \infty$, $\infty \times \infty$
$/$	$0/0, \infty/\infty$	Overflow, $x/0$, ∞/x, $\infty/0$

Table 5.1: Operations that produce NaN and INFINITY in IEEE 754 arithmetic. The variable x always represents a regular represented quantity different from 0. NaN is also propagated so that if an operand is NaN the result will also be NaN.

Assume that the input to the recursive triangular inversion is free of NaN and INFINITY. When pure recursion is applied, division only occurs at the leaves by the operation $1/x$ (x is non-zero because otherwise the triangular matrix is singular). Table 5.1 says that this operation can yield an INFINITY if overflow occurs. The triangular matrix multiplies can therefore have a triangular operand that contains INFINITYs. These may propagate into the rectangular result of the multiplication as INFINITYs or NaNs. It should be clear that if overflow occurs somewhere in the computation the final result will contain at least one INFINITY and zero or more NaNs. Therefore, testing for INFINITY in the result is enough to detect any overflow generated during the computation.

In IEEE 754 arithmetic, INFINITY and NaN can be detected by checking if $x - x = 0$ (they can also be detected directly from the bit-pattern but most programming languages do not have facilities for this), since these two special quantities are the only ones that have this property. This might not be portable to other floating point systems. Another approach is to use LAPACK's portable way of determining an overflow threshold T and test for overflow by the comparison $x > T$ which is true if x has overflowed. That method is used in the implementation of the algorithms in this report.

5.4 Comparison with an Iterative Method

As already pointed out, the flop distribution of top heavy recursive algorithms tends to follow a variable geometric series. This is also true for triangular inversion as can be seen by the following.

At level i in the recursion there are 2^i nodes with problems of size $N/2^i$. The operations that occur at each node are two `TRMM` of half that size, so $2(N/2^{i+1})^3 = N^3/2^{3i+2}$. Combining the work of all nodes at that level results in $N^3/2^{2i+2} = N^3/4^{i+1}$. That is, for each descended level the flops per level goes down to one fourth. The flops are distributed according to a geometric series with ratio 1/4.

To see the relevance of this we consider the iterative approach used by fixed blocked algorithms such as `TRTRI` in LAPACK. In that routine there are N/NB iterations each involving $(NB^3+2NB)/3+(i-1)(NB)^3+(i-1)^2(NB)^3$ flops. When recursion is combined with blocking there will be N/NB leaves and $N/NB-1$ internal nodes. In the iterative method one iteration contains two zero-sized operations resulting in $N/NB-1$ actual operations. Thus, the methods are very similar in that respect.

To illustrate the relation between the recursive and iterative method the distribution of work over the iterations in the iterative method and the internal nodes in the recursive method is compared in Figure 5.6. The number of internal nodes is equal to the number of iterations, so by sorting the internal nodes by level the figure gives a reasonable comparison. Since both methods perform the same amount and size of kernel operations those are not considered in the comparison.

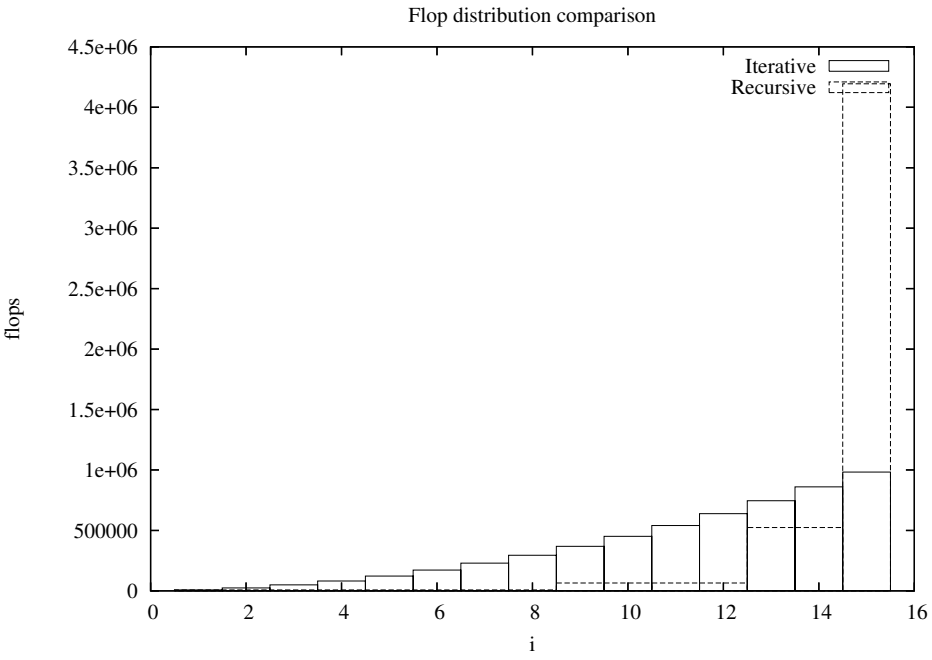


Figure 5.6: Comparison between the flop contents in the iterations and internal nodes of the iterative and recursive methods, respectively.

It is clear that all but the largest internal node contain less work than the corresponding iteration. This work difference is then made up in the topmost node making very large BLAS operations possible. As this property stems from the top heavy work distribution, this property shows up in other recursively blocked algorithms as well.

Chapter 6

SMP Parallelization of Triangular Inversion

A Symmetric MultiProcessor system (SMP system) is one kind of shared memory multiprocessor system. Programs which can be parallelized by introducing multiple threads may benefit on such systems since each thread can be assigned to different processors.

6.1 Task Parallelism in Triangular Inversion

The recursive triangular inversion algorithm (Algorithm 4) presented in this report contains task parallelism since the two recursive calls are independent. However, approximately 75% of all flops are in the two `TRMM` operations at the top of the recursion tree. Any parallelization will have to exploit parallelism in these operations. This can be achieved either by linking with an SMP-aware BLAS library or by explicit blocking. One drawback with linking with a special library is that only BLAS operations are parallelized, not recursive calls or kernel functions. By exploiting the inherent task parallelism this is resolved, but relies on the SMP BLAS library not to use more threads than necessary. On the other hand, using explicit blocking to produce parallel tasks has none of these drawbacks, but comes at a greater programming cost and probably exhibits slightly worse performance when looking at a single `TRMM` operation.

Since the result overwrites one of the operands, independent calculation can be made in one dimension only. This means that meaningful partitions will be either block-row partitioned or block-column partitioned, depending on which side the multiplication occurs.

6.2 Task and Data Parallelism using OpenMP

OpenMP is the de-facto standard for SMP parallelization using threads [2]. It abstracts away the details of thread handling into compiler directives. The OpenMP directives are focused around loop-level parallelism with facilities for different load balancing schemes. Parallelizing divide and conquer algorithms with OpenMP can rely on nested parallelism where one parallel region is contained within another. One thread is spawned for each recursive call. A limit on the number of created threads must be set. In OpenMP this is achieved by enabling nested parallelism and setting a maximum number of created

threads. The standard for OpenMP allows for a compiler to serialize nested parallel regions, so this method relies on a compiler which does not serialize those regions.

After the recursive calls are completed in parallel, two `TRMM` operations must be parallelized. At the top level all P processors could be used, but only two threads have been created to exploit the task parallelism earlier. This requires the destruction of two threads and the creation of P new ones. A similar situation arises for the first couple of levels in the recursion. Nested parallelism for this application will have significant penalties due to thread creation and destruction.

One solution is to spawn all P threads at once. At the first level of recursion, $\lceil P/2 \rceil$ threads go down the left branch, and $\lfloor P/2 \rfloor$ go down the right branch. Once all threads return from the recursion they start their collaborative `TRMM` operations.

6.3 SMP Algorithm

The idea presented above will be discussed below. There are a number of questions that need to be answered:

1. How is the work partitioned during `TRMM` operations?
2. The triangular multiplies can not begin until the input arguments are computed. Hence, synchronization is needed before every such parallel operation. How is synchronization before `TRMM` operations performed?
3. How is the load balanced when the number of threads in two sibling trees is unequal?
4. How is the issue of too fine-grained partitioning resolved?

For left `TRMM` operations each column is independent whereas each row must be calculated bottom up or top down depending on the type of triangular matrix. Similarly, for right `TRMM` operations where each row is independent whereas the columns must be calculated in one of two ways, again depending on the type of triangular matrix. This means that the left operation must be partitioned along the columns and the right operation along the rows. The amount of work is the same for all rows and columns so each thread in a P thread `TRMM` operation gets assigned N/P columns or rows, where N is the number of columns or rows.

The second item above is handled through barriers which synchronize all threads. This works fine for the top node in the recursion where all threads participate in the same operations. As for the lower nodes, this will be a suboptimal synchronization since it will make the slowest working thread on that level determine the speed of all threads. It is also necessary to make sure that all threads agree on how deep in the recursion they should go so that all threads execute all barriers in the same order.

As for the third item the answer is: it is not. The cross-branch synchronization effectively throws away some of the benefits of having more threads. This is, however, only true some levels down the tree where the amount of work is small. It is therefore to be expected that the loss will be small. It is important to note that the topmost node (where the bulk of the work is) still can use all threads. The loss will be most severe for $P = 2^k - 1$ since the number of threads effectively idle at level i is $2^i - 1$ as can be seen in Figure 6.1. Because of the cross–branch synchronization each level will appear to run as fast as the smallest team size. When $P = 2^k - 1$ there will be only one team of

small size at each level. For instance, on level 2 there are 7 processors but the smallest team size is 1, making the level appear to have only 4 effective processors. A number

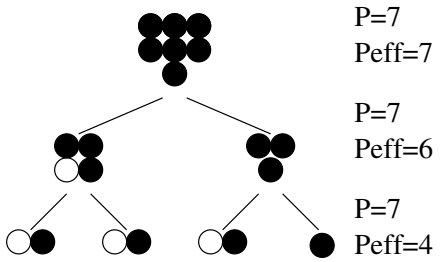


Figure 6.1: Illustration of the effective waste caused by cross–branch synchronization

of more or less sophisticated ideas to cut down on the idling can be thought of, all of them relaxing the cross-branch synchronization. Since the imbalance in the number of threads between two siblings is at most one and there are two collaborative operations per node, one thread could work in the left subtree for the first operation and then in the right subtree on the next. Since the performance gained this way will be relatively small it has not been tested in this report.

To see the sanity of wasting resources this way, notice that the number of ineffective threads at level i is independent of the number of processors. Thus, for increasing P the relative loss defined as the ratio between ineffective processors and P decreases. The loss is most severe for $P = 3$ and then quite rapidly drops to acceptable levels (see for example Section 8.4 where the overhead for $P = 3$ is clearly visible while $P = 7$ and $P = 15$ do not suffer from the same degradation).

There are two things to answer about the fine-grained partitioning. When partitioning the `TRMM` operations there is a possibility that utilizing P threads on chunks of size N/P is actually faster than utilizing $P + 1$ threads on chunks of size $N/(P + 1)$. This is because the chunks may be so small that the performance on such a chunk is worse than on the slightly bigger chunk obtained with fewer threads. It can be shown that the drop in performance need only be relatively small before it becomes unwise to use more threads. This threshold will probably depend on system parameters as well as on the input size. However, it is thought that some sensible smallest size can be used as an approximation. If $N/P < NB_{min}$, then only $P' = \lfloor N/NB_{min} \rfloor$ threads will be used. This can result in $P' = 0$ and other nasties like $P = 2, P' = 1$ with the lone thread executing on almost twice NB_{min}. This is overcome by changing $\lfloor \quad \rfloor$ to $\lceil \quad \rceil$, which is done in the implementation of the algorithm.

The other issue with fine-grained partitioning is when to stop the recursive splitting of collaborative groups. For some size $N < PNB$ it will be more beneficial to use only one thread serially than multiple threads concurrently. This can happen at any level in the recursion tree. This must be combined with the imperative need for agreement on when to stop recursion. Theorem 1 states that the sizes of the nodes on a level differ by at most one. By using the structure of the proof as a template, it is easy to add a flag to the parameter list of the parallel inversion routine so that it knows if it is "big" or "small". All threads can thus agree on what numbers to base their stopping decision on.

A detail related to the issue of agreement is that recursion needs to stop when one node has only one thread in its team. Other nodes may or may not have two threads

at the same level, which is why a similar bookkeeping as described in the previous paragraph is used to keep track of when one node at the current level has only one thread.

OpenMP supports barriers for the whole thread group only, not for subgroups [2]. This application would benefit from having subgroup barriers. While it is reckoned that this can be emulated through locks and busy-waiting, this would only serve to obscure the otherwise simplistic algorithm.

There are many ways to implement the team splitting. One of the simpler methods is to number each thread consecutively as $0, 1, \ldots, P-1$. Each team is represented by a consecutive subinterval. The team interval is passed along with the parameters. If a thread finds itself in the left half of the interval it will be a part of the left subtree, otherwise the right.

Since the details regarding agreement have already been discussed, it will not be mentioned in the algorithm description. Algorithm 5 outlines the parallel SMP algorithm.

Algorithm 5 PRECTRTRI(A, P)

```
1: if Base-case then
2:    TRTRI(A)                                   % Serial execution
3: else
4:    Split A in half: A = [ A11  A12 ]          % Partition A
                           [ A21  A22 ]
5:    if part of left subtree then
6:       PRECTRTRI(A11, ⌈P/2⌉)
7:    else if part of right subtree then
8:       PRECTRTRI(A22, ⌊P/2⌋)
9:    end if
10: end if
11: BARRIER
12: Perform part of left TRMM                    % Collaborative TRMM
13: BARRIER
14: Perform part of right TRMM                   % Collaborative TRMM
```

6.4 Coping with Slow Threads

Algorithm 5 presented above exhibits natural load balance. If the threads execute at different speeds this balance is not enough for high performance. Since the `TRMM` operations are seen as black boxes of indivisible labour, the only way to alleviate slow threads and achieve dynamic load balancing is to make a more fine-grained partitioning. This, however, comes at a great expense. Reducing the partitioning will usually not affect performance considerably until the slice is smaller than a certain threshold. This threshold is typically significantly greater than one, for many machines in the neighborhood of 32. Splitting into partitions smaller than this can be very costly, inducing a practical limit on how fine-grained partitions can be made. Another (potentially more serious) problem is memory contention. The triangular operand must be swept through for every slice. Increasing the number of slices may cause even more memory contention. On a non-dedicated system the only practical solution is dynamic load balancing so the

drawbacks of finer partitions must be accepted. This report does not evaluate dynamic load balancing.

Chapter 7

Extension to Dense Square Matrices

Inverting a dense square matrix by first forming the LU factorization has seen adoption in LINPACK, LAPACK, and MATLAB [15].

A square matrix A can be factorized into $PA = LU$ where P is a permutation matrix, L is lower unit triangular, and U is upper triangular, if and only if A is invertible. However, an LU factorization is not unique; U can just as easily be made unit triangular, and the permutation need not be unique either. We use the LAPACK routine to factorize A, and because of this we will discuss the topic in light of the details of how this routine works. Algorithm 6 outlines a procedure functionally analogous to LAPACK's `GETRI` which assumes an LU factorization by `GETRF`.

Algorithm 6 RECGETRI(A, $IPIV$)

1: Invert U and L by calls to `RECTRTRI`
2: Multiply U by L
3: Unscramble the columns of A by swapping columns based on the pivoting information in $IPIV$

7.1 Factorization with LAPACK `GETRF`

The routine `GETRF` factors a general $M \times N$ matrix using partial pivoting with row interchanges [1]. The routine works on a dense matrix, storing U in the upper triangular part and L in the lower triangular part, not explicitly storing the unit diagonal. Extra storage of size N is needed for the permutation information. For a square $N \times N$ matrix the vector `IPIV` has length N and for each $1 \leq i \leq N$ says that row i was interchanged with row `IPIV`(i).

7.2 Inversion with LAPACK `GETRI`

LAPACK provides the routine `GETRI` which takes as input a factorization by `GETRF` and computes A^{-1} by inverting the triangular matrix U and solving $A^{-1}L = U^{-1}$ for A^{-1}.

The triangular shape of the right hand side is not matched by any BLAS routine, so `GETRI` uses a fixed blocked method to solve the system. The compact storage of U and L forces this method to store one block column in a temporary matrix, increasing the memory requirements slightly.

7.3 Multiplying U by L

Multiplication of one upper triangular by one lower triangular matrix yields a dense square matrix. The formula for multiplying two $N \times N$ triangular matrices U and L with L being unit triangular is given by:

$$a_{ij} = \begin{cases} \sum_{k=i}^{N} u_{ik} l_{kj} & \text{if } j < i, \\ u_{ij} + \sum_{k=j+1}^{N} u_{ik} l_{kj} & \text{if } j \geq i. \end{cases} \tag{7.1}$$

Consider the case where $j < i$ for a while. The set of a_{ij} satisfying this condition all belong to the L part of A. The elements of U and L in the computation of a_{ij} are $\{u_{ix}, l_{xj} : x \geq i\}$. That is, the i:th row of U and the column below (and including) l_{ij}. Now let $j \geq i$. The set satisfying this condition belongs to the U part of A. The elements needed are $\{u_{ij}, u_{ix}, l_{xj} : x \geq j\}$. In other words, the row to the right of (and including) u_{ij} and the j:th column of L. The conclusion is that when computing the product top-down left-to-right there is no need for auxiliary storage.

The flop count to multiply U by L is derived from Equation (7.1) as

$$F(N) = \frac{4N^3 - 3N^2 - N}{6}. \tag{7.2}$$

7.4 U Times L Recursively

By partitioning U and L according to

$$U = \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix}$$
$$L = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix},$$

the product UL in block representation becomes

$$UL = \begin{bmatrix} U_{11}L_{11} + U_{12}L_{21} & U_{12}L_{22} \\ U_{22}L_{21} & U_{22}L_{22} \end{bmatrix}.$$

The shape of the operations is shown in Figure 7.1.

Algorithm 7 describes this recursive blocked procedure more formally.

7.4.1 SMP Parallelization of U Times L

The task dependency between the operations in the recursive blocking of UL when they are stored together in A admit little parallelism. Figure 7.2 shows two recursive task graphs. Thick circles represent another node of this type. Part a) shows the limited parallelism available when computing without workspace. Part b) shows how parallelism is induced by making explicit backups of some parts. Only the two `TRMM`

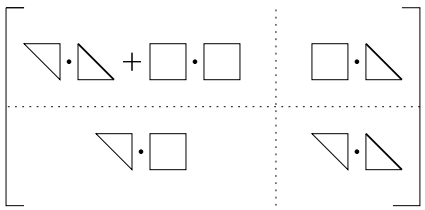


Figure 7.1: Shape of the operations in the UL product

Algorithm 7 RECTRTRM(A)

Require: U and L are stored packed in A

1: Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
2: RECTRTRM(A_{11}) % Recurse on A_{11}
3: $A_{11} := A_{11} + A_{12} \times A_{21}$ % `GEMM` operation
4: $A_{12} := U_{12} \times L_{22}$ % `TRMM` operation
5: $A_{21} := U_{22} \times L_{21}$ % `TRMM` operation
6: RECTRTRM(A_{22}) % Recurse on A_{22}

operations can be executed in parallel, everything else will be serialized. This is not so big a deal because the BLAS operations contain enough parallelism in themselves to provide reasonable speedup. When many threads work together this will most likely make the problem more visible, because the serial work imposes more combined idle-time and the vast amount of barriers will also hurt performance.

The only thing that prohibits parallel execution of the left and right recursive calls is an anti-dependence between the input to the triangular multiplies and the right recursive call. By making a backup of A_{22} and routing the triangular multiplies to use the backup instead, the problem is solved. The recursive application of the function means that more areas need to be backed up. This is shown in Figure 7.3 where the grayed areas show portions that need to be backed up when $P = 8$. Part a) contains grayed areas. These are the areas that need to be copied, but they are also recursive in nature so only the three labeled regions A, B, and C need explicit copying. Part b) shows where in the recursion tree these regions are used. Part c) shows how an array indexed by the level in the recursion tree can be used to select which workspace to use. The nested areas seen in both region A and B do not need explicit copying because they are part of their respective bigger region. Thus, for $P = 8$ only three regions need to be backed up. For $N = 2^k$ the extra workspace entries can be quantified as $\frac{N^2}{3} - \frac{N^2}{3 \times 4^{\log P}}$, which in the limit approaches $\frac{N^2}{3}$.

The task diagram (Figure 7.2) implies that the backup must be done first, and can thus be done in parallel by all threads. This reduces the overhead of the backup.

Algorithm 8 outlines a version that uses no task parallelism and instead completely relies on data parallelism within `TRMM` and `GEMM` operations. Note that linking with an SMP BLAS implementation achieves the same thing.

The workspace is indexed based on the level in the recursion as described earlier. Algorithm 9 lists the modified `PRECTRTRM` to work with workspaces. It follows the same algorithmic idea as `PRECTRTRI`.

Algorithm 8 PRECTRTRM(A)

Require: U and L are stored packed in A

1: **if** $N <= PNB$ **then**
2: One thread calls RECTRTRM(A) % Only one thread executes this
3: **else**
4: Partition $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
5: PRECTRTRM(A_{11}) % Recurse on A_{11}
6: BARRIER
7: $A_{11} := A_{11} + U_{12} \times L_{21}$ % Perform part of `GEMM` operation
8: BARRIER
9: $A_{12} := U_{12} \times L_{22}$ % Perform part of `TRMM` operation
10: $A_{21} := U_{22} \times L_{21}$ % Perform part of `TRMM` operation
11: BARRIER
12: PRECTRTRM(A_{22}) % Recurse on A_{22}
13: **end if**

Algorithm 9 PRECTRTRMCOPY(A, P, L, $WORK$, B)

1: Partition A in half: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
2: **if** B is null **then**
3: $B := WORK(L)$
4: **else**
5: Partition B similar to A as $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
6: $B := B_{22}$
7: **end if**
8: **if** Part of left subtree **then**
9: PRECTRTRM(A_{11}, $\lfloor P/2 \rfloor$, $L+1$, $WORK$, B) % Recurse on A_{11}
10: **else if** Part of right subtree **then**
11: PRECTRTRM(A_{22}, $\lceil P/2 \rceil$, $L+1$, $WORK$, B) % Recurse on A_{22}
12: **end if**
13: BARRIER
14: $A_{11} := A_{11} + U_{12} \times L_{21}$ % Part of `GEMM`
15: BARRIER
16: $A_{12} := U_{12} \times B_L$ % Part of `TRMM`, B_L is lower triangular part of B
17: $A_{21} := B_U \times L_{21}$ % Part of `TRMM`, B_U is upper triangular part of B

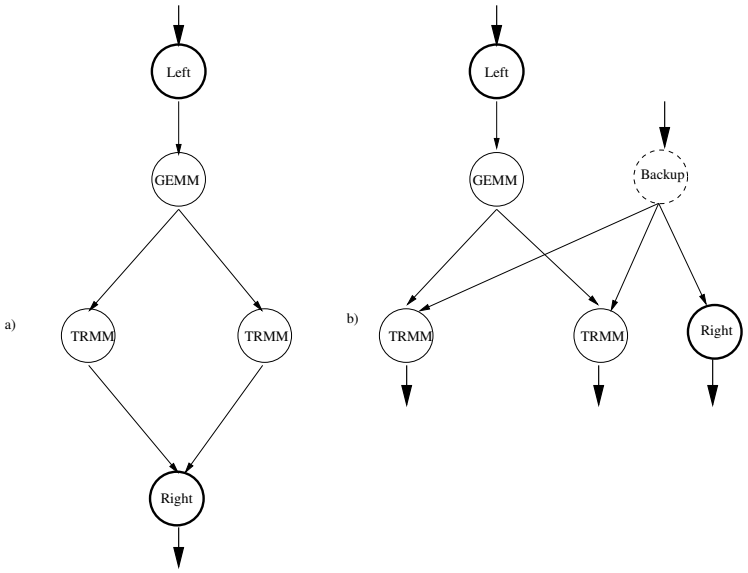


Figure 7.2: General nodes in the recursive task graphs

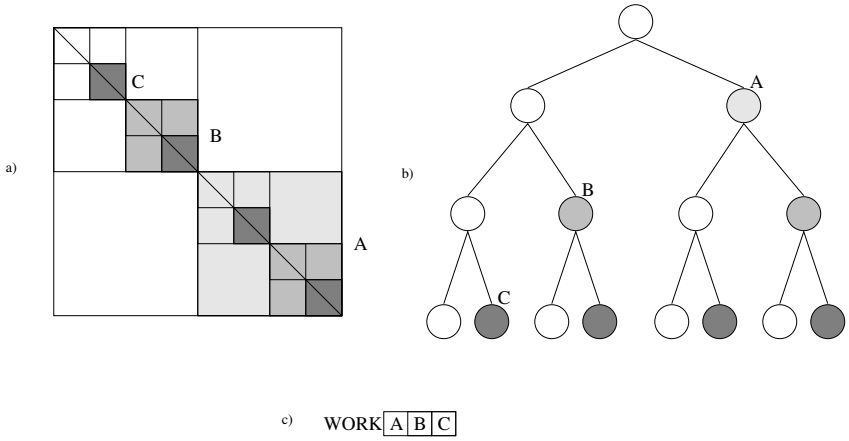


Figure 7.3: Creating task parallelism by using workspaces

Chapter 8

Performance and Accuracy Results

This chapter presents and discusses performance and accuracy of the recursive blocked uniprocessor and SMP implementations presented in this report.

8.1 Testing Methodology

Both uniprocessor and SMP timings were extracted using a test program written in C. The wall clock time of one function call was measured with the `gettimeofday` function 5 times for the uniprocessor tests and 2 times for SMP tests. This function has a microsecond resolution, although the accuracy is implementation dependent. For all but the smallest matrices this resolution and accuracy is more than enough. The minimum time of these tests was then used as the official value because it is assumed to minimize the effect of unlucky scheduling. The wall clock time was chosen above process time because it better reflects the actual usage of the routines. This is especially true for the SMP system where the large number of processors makes the algorithm more susceptible to system interruptions.

8.1.1 Machines Used

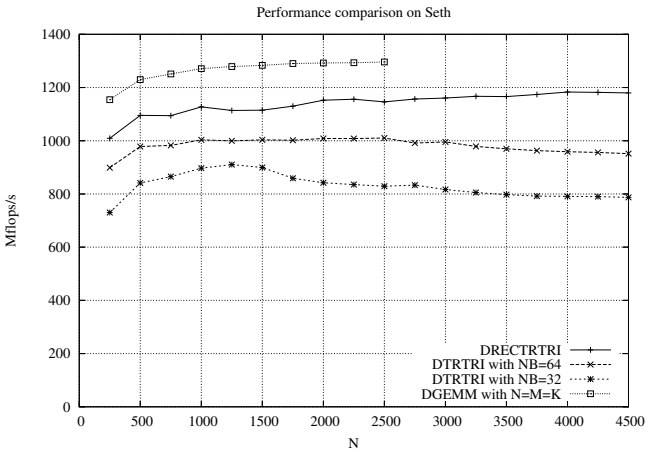
Three different machines were used to produce the results. The uni-processor tests were run on one node of the Seth cluster at HPC2N. It has 120 nodes with dual Athlon MP2000+ processors and 1GB memory per node. TLB performance was measured on Chips at HPC2N which is an IBM Power3 based SMP system running at 375MHz and 4GB of memory. The SMP machine that was used has 32 processors and is located at UPPMAX, Uppsala University. It is a Sun Fire E10000 server.

8.2 Uni-processor Results

This section presents timings for the `RECTRTRI` routine as well as the `RECGETRI` routine and compares them to their counterparts in LAPACK. All experiments were run on one node of the Seth cluster at HPC2N in Umeå.

8.2.1 RECTRTRI Timings

The serial algorithm is expected to give good temporal locality in all levels in the memory hierarchy. Combined with a high performance kernel this will provide more performance than is lost due to recursion overhead.



N	Ratio
250	1.12
500	1.12
750	1.11
1000	1.12
1250	1.11
1500	1.11
1750	1.13
2000	1.14
2250	1.15
2500	1.13
2750	1.17
3000	1.17
3250	1.19
3500	1.20
3750	1.22
4000	1.23
4250	1.24
4500	1.24

Figure 8.1: RECTRTRI performance compared to LAPACK TRTRI. The table shows performance ratio of RECTRTRI to TRTRI with block size 64.

Several points can be made from Figure 8.1. Performance of the recursive algorithm tends to increase with N and is substantially closer to GEMM performance than the LAPACK routine TRTRI is. Note that the GEMM performance could not be measured for all input sizes due to memory shortage. The importance of picking a good blocking parameter for the LAPACK routine is evident from the figure. As previously mentioned the blocking parameter in traditional algorithms affect the size of most BLAS operands, while in the recursive case the blocking parameter only affects the leaf nodes, where in the case of top heavy trees a very small portion of the work is performed. This means that although picking a good parameter for hybrid recursive algorithms is important to achieve the best performance the impact is *reduced* with increased problem size. The same may or may not be true for conventional blocking techniques, but Figure 8.1 does provide reason to suspect that the impact is not reduced in a similar way.

8.2.2 RECGETRI Timings

Recall that RECGETRI inverts a general matrix based on an already computed LU factorization, which is functionally equivalent to GETRI in LAPACK. The implementation is different though. The LAPACK routine requires linear additional workspace. There are three steps in RECGETRI (two triangular inversions and one multiply) which are all recursive. The triangular inversion has already been shown to be somewhat superior on the test machine. Figure 8.2 shows results for RECGETRI on the same machine. The tests performed are similar to the triangular inversion. The recursive algorithm outperforms the serial, which is also sensitive to the block size throughout the input size range. Notice that the serial algorithm seems to suffer from a significant dip in performance,

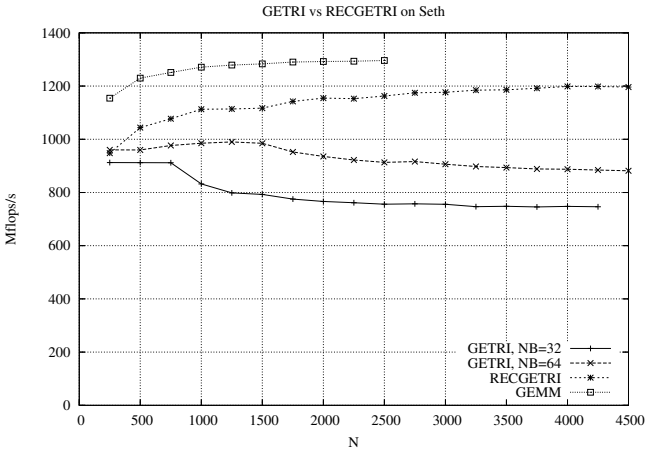


Figure 8.2: `RECGETRI` performance compared to LAPACK `GETRI`. The table shows performance ratio of `RECGETRI` to `GETRI` with block size 64.

N	Ratio
250	0.99
500	1.09
750	1.10
1000	1.13
1250	1.13
1500	1.13
1750	1.20
2000	1.23
2250	1.25
2500	1.27
2750	1.28
3000	1.30
3250	1.32
3500	1.33
3750	1.34
4000	1.35
4250	1.35
4500	1.36

while the recursive tends to increase and flatten out. The block sizes for the serial algorithm where chosen arbitrarily, but a block size of 128 was tested to assess how far from optimality they are. The results were almost identical to the 64 block size case.

8.3 Serial Fraction

Amdahl's law is based on the assumption that a program's execution time can be divided into a serial part and a complete parallel part. The fraction serial time to total time can be used as a diagnostic tool.

When the number of processors increases on a fixed problem size, the efficiency is decreased. This is true for almost all parallel algorithms as the overhead is typically increased with the number of processors. This shows up in efficiency graphs as a downward slope. This slope may or may not indicate how fast the overhead changes with increased number of processors. The serial fraction is a measure which highlights the overhead of a parallel algorithm and is therefore used in this chapter to discuss the scalability of the parallel algorithms.

8.4 SMP Parallel Triangular Inversion

Figure 8.3 shows the experimentally determined serial fraction of `PRECTRTRI`. As previously discussed, the loss due to idling threads when subtrees have unequal team sizes becomes less relevant when P increases. Moreover, odd P is worse than even because even P creates balanced teams at both of the two topmost levels, while odd P is balanced only at the root level. Both of these predictions are confirmed by Figure 8.3, where peaks occur on all odd P and the fluctuations in the intervals $2 \ldots 4$, $4 \ldots 8$, and $8 \ldots 16$ decreases.

Tests were performed for input sizes $500, 1000, \ldots, 10000$ for $P = 1, 2, \ldots, 24$. The

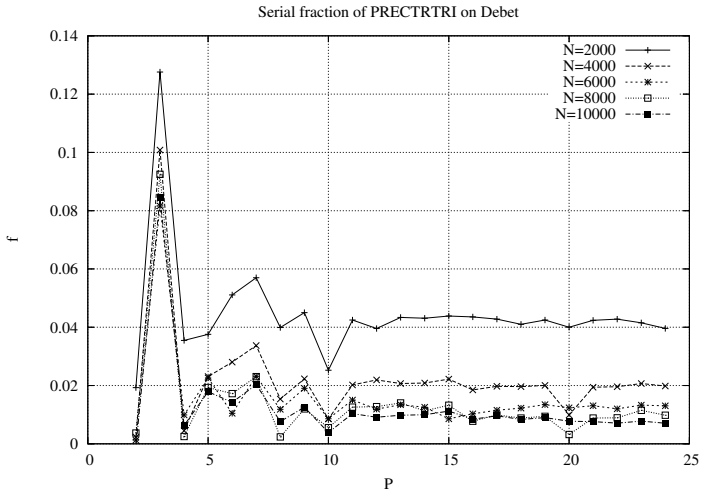


Figure 8.3: Serial fraction of the parallel triangular inversion algorithm PRECTRTRI

highest attained efficiency for $P = 24$ was 0.86 when $N = 10000$. Table 8.1 lists the smallest input for a given P that yielded an efficiency of at least 0.86. For brevity only some P are listed. By studying this table the hypothesis that doubling both the input dimension and processors keeps the efficiency constant seems plausible. So the algorithm is scalable in the sense that increasing both P and N will keep the efficiency constant, but doing so would require doubling the memory needed per processor since a doubling in dimension means quadrupling the memory footprint.

P	N
1	500
2	1000
4	1500
5	2000
8	3000
10	4000
16	5500
20	8000

Table 8.1: Input sizes needed for a parallel efficiency $E = 0.86$

8.5 SMP Parallel U Times L

This section presents and discusses performance figures for the SMP parallel version of U times L.

The extra workspace is used to create more parallelism and reduce the amount of synchronization. The lack of parallelism without workspace is most serious when the current problem size is small. At some point the algorithm (`PRECTRTRM`) switches to serial execution because it will run faster, effectively serializing the leaves of the recursion tree. Because of the geometric properties of the work distribution the relative amount

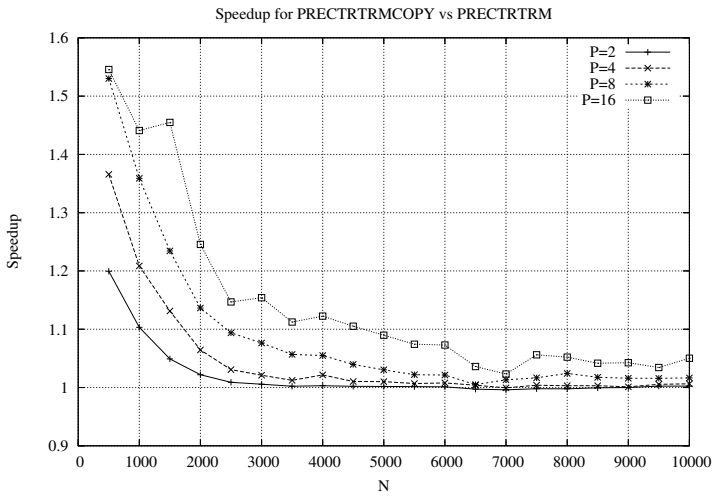


Figure 8.4: Speedup gained when using workspace compared to working without

of serialized work will diminish quite rapidly with increasing problem size. Although the rest of the computations can improve by more processors, the serial part cannot, so the benefit of using workspace ought to increase with P.

Figure 8.4 shows the speedup of `PRECTRTRMCOPY` relative to `PRECTRTRM`. As was expected the speedup quickly vanishes and is substantially bigger for increasing P, but probably damped quite a bit by the overhead of memory management and copying.

8.6 Accuracy of RECGETRI

An analysis of the numerical properties of the inversion algorithms developed herein are beyond the scope of this report. Instead a number of experiments were performed to compare the accuracy of LAPACK's `GETRI` to `RECGETRI`.

The accuracy of the computed inverse B of A was numerically estimated by:

$$\frac{\max(||I - AB||, ||I - BA||)}{||A||}. \tag{8.1}$$

We have previously mentioned the four methods which form a subset of those possible to compute the triangular inverse. Algorithm 10 gives the four methods named A–D explicitly.

To assess the impact of these methods on square inversion a set of 100 inversions on matrices of size 100×100 with elements uniformly distributed in the $[-1, 1]$ range was performed. The mean and standard deviation of the measured accuracy for the iterative (LAPACK) algorithm and the four methods are reported in Table 8.2.

All methods give very accurate results, about 13 decimal figures. Methods B–D give somewhat better accuracy, around 14 decimal figures.

Judging from Table 8.2, the safest bet is to go with Method D, although the differences are small.

Algorithm 10 RECTRTRI(A, METHOD)

Require: A is $N \times N$ and lower triangular

if $N = 1$ **then**
  A is scalar so $A := 1/A$
else
  Partition $A = \begin{bmatrix} A_{11} & \\ A_{21} & A_{22} \end{bmatrix}$
  if METHOD = A **then**
    RECTRTRI(A_{11}) % Invert A_{11}
    RECTRTRI(A_{22}) % Invert A_{22}
    $A_{21} := A_{21} \times A_{11}$ % `TRMM` operation
    $A_{21} := -A_{22} \times A_{21}$ % `TRMM` operation
  else if METHOD = B **then**
    RECTRTRI(A_{22}) % Invert A_{22}
    $A_{21} := -A_{22} \times A_{21}$ % `TRMM` operation
    Solve for A_{21} on LHS of $A_{21}A_{11} = A_{21}$ % `TRSM` operation
    RECTRTRI(A_{11}) % Invert A_{11}
  else if METHOD = C **then**
    RECTRTRI(A_{11}) % Invert A_{11}
    $A_{21} := -A_{21} \times A_{11}$ % `TRMM` operation
    Solve for A_{21} on LHS of $A_{22}A_{21} = A_{21}$ % `TRSM` operation
    RECTRTRI(A_{22}) % Invert A_{22}
  else if METHOD = D **then**
    Solve for A_{21} on LHS of $A_{22}A_{21} = A_{21}$ % `TRSM` operation
    Solve for A_{21} on LHS of $A_{21}A_{11} = A_{21}$ % `TRSM` operation
    RECTRTRI(A_{11}) % Invert A_{11}
    RECTRTRI(A_{22}) % Invert A_{22}
  end if
end if

	Mean ($\times 10^{-15}$)	Std dev ($\times 10^{-15}$)
Iterative	13.4	50.5
Method A	19.0	63.0
Method B	8.3	16.0
Method C	7.7	20.0
Method D	7.0	13.0

Table 8.2: Accuracy of `GETRI` compared with recursive inversion using four different methods of triangular inversion

8.7 TLB Performance

The Performance API (PAPI [3]) was used to measure the number of TLB misses on the IBM Power3 machine named Chips at HPC2N. As Jonsson points out it is important to know what one is measuring [17]. By consulting the result of the list of available events on that particular machine it was decided to measure the `PAPI_TLB_TL` event. Figure 8.5 shows TLB misses for both `TRTRI` and `RECTRTRI`. For medium sized matrices there are actually more misses for the recursive algorithm. As the problem size grows the recursive algorithm clearly performs better. The graph of the iterative algorithm have distinct shifts in the graph at regular intervals. The first gap (although not visible on this graph) is at $N = 210$. The offset between shifts is 510 for all but one which is 520. This interesting regularity probably corresponds to the fact that a page holds 512 double words on this system, but the origin of these shifts is not known. Is the pattern

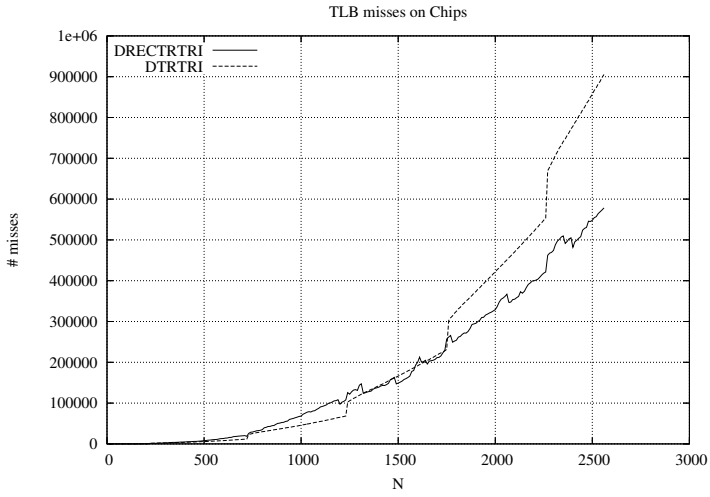


Figure 8.5: Number of TLB misses measured on a matrix with leading dimension 2560 ($= 5 \times 512$, so that each column is exactly five pages)

in this graph representative for other systems as well? This question is hard to answer, partly because of the difficulty in getting access to hardware performance counters. The PAPI project runs on many platforms but requires kernel patching for Linux systems.

It can be noted that using NetLib's reference BLAS implementation gives the opposite conclusion: the iterative method give less TLB misses for large matrices. This

gives a reminder that the BLAS implementation interacts with the algorithm in ways which may overthrow the theoretical predictions, such as lower memory traffic and TLB misses. Evaluating this complex interaction might be attempted from a theoretical point of view, but will probably be more accurately evaluated using hardware performance counters.

Chapter 9

Conclusions

Matrix inversion via recursion shifts most operations into only a few large BLAS operations. It also matches arbitrary levels in the memory hierarchy with respect to temporal locality. The spatial locality might be inferior to the compared iterative method depending on the BLAS implementation. However, tests with the ATLAS BLAS implementation showed that the recursive algorithm is superior.

There is a multitude of arrangements in the computation of triangular inverses which lead to slightly different numerical behavior of square inversion. The difference is small for the tested class of matrices, but in a practical setting testing could be done on the actual matrix class and then the most accurate implementation strategy could be selected. The selection should not alter the performance in any significant way, but the parallel algorithm works with only some of them (those with no dependence between the computation of the left and right subtrees).

The recursive algorithms shifting towards large BLAS operations make it possible to create large chunks of work between synchronization. When the subtrees are independent their computation can be performed in parallel with natural load balance. Even without independent subtrees parallel versions are possible either by using an SMP–aware BLAS implementation or by explicitly partitioning the BLAS operations. The implementation is easy with OpenMP, but would be almost trivial if OpenMP supported divide and conquer via team parallelism for example. One important note regarding team parallelism and SMP–aware BLAS is that it becomes imperative to limit the creation of threads in BLAS routines to the team size. This is easily accomplished if the implementation took a P parameter for example.

9.1 Future Work

Using a more suitable data layout will probably improve on the TLB performance. This has not been tested in this report because the algorithm performs relatively few flops per data cell. It would however be interesting to study the impact of another layout in conjunction with the minimal storage Cholesky factorization. This would possibly result in a minimal storage positive definite inversion algorithm with level 3 BLAS performance characteristics.

In the recursive blocking of U times L splitting not in the middle but slightly skewed to either side will either increase or decrease the amount of flops carried out in `GEMM` operations. This skewing will somewhat increase the overhead of recursion and limit the

theoretically available reuse in operations by making them more rectangular in shape. However, with BLAS implementations like libGOTO and ATLAS the performance for non–`GEMM` operations seem to be significantly smaller. In such circumstances additional speed may be gained by making such a skewed blocking.

9.2 Acknowledgments

Many thanks goes to Robert Granat for guiding the work, giving valuable feedback and ideas. I would also like to thank Bo Kågström as well as Robert Granat for spending time and effort to improve the quality of the report.

This research was conducted using the resources of High Performance Computing Center North (HPC2N).

We would like to thank Sverker Holmgren and UPPMAX in Uppsala for providing access to their computer resources.

References

[1] LAPACK User Guide. See website: `http://www.netlib.org/lapack/lug`. Visited November 2005.

[2] OpenMP. See website: `http://www.openmp.org`. Visited November 2005.

[3] Performance Application Programming Interface. See website: `http://icl.cs.utk.edu/papi`. Visited November 2005.

[4] Howard Anton and Chris Rorres. *Elementary Linear Algebra: Applications Version*. John Wiley & Sons Inc., 2000.

[5] 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.

[6] Erik Elmroth and Fred G. Gustavson. New Serial and Parallel Recursive QR Factorization Algorithms for SMP Systems. In *PARA98*, pages 120–128, 1998.

[7] David Goldberg. What Every Computer Scientist Should Know About Floating–Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[9] Kazushige Goto and Robert van de Geijn. On Reducing TLB Misses in Matrix Multiplication. Technical report, CS, University of Texas at Austin, Austin, Texas, US, 2002.

[10] Ivor Grattan-Guinness. *Companion encyclopedia of the history and philosophy of the mathematical sciences*, volume x. Routledge, 1994.

[11] Fred Gustavson, Andre Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms. In *PARA98*, pages 195–206, 1998.

[12] Fred Gustavson, Andre Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Superscalar GEMM-based Level 3 BLAS — The On-going Evolution of a Portable and High-Performance Library. In *PARA98*, pages 207–215, 1998.

[13] Fred G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Develop.*, 41(6):737–755, 1997.

[14] Fred G. Gustavson and Isak Jonsson. Minimal-storage high-performance Cholesky factorization via recursion and blocking. *IBM J. Res. Develop.*, 44(6):823–850, 2000.

[15] Nicholas J. Higham. *Accuracy and stability of numerical algorithms, 2nd edition.* SIAM, 2002.

[16] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms.* W. H. Freeman and Company, 1998.

[17] Isak Jonsson. Analysis of Processor and Memory Utilization of Recursive Algorithms for Sylvester-Type Matrix Equations using Performance Monitoring. Technical report, Dept. of Comp. Sc., Umeå University, Umeå, Sweden, 2003.

[18] Isak Jonsson and Bo Kågström. Recursive Blocked Algorithms for Solving Triangular Systems — Part I: One-Sided and Coupled Sylvester-Type Matrix Equations. *ACM Transactions on Mathematical Software*, 28(4):392–415, December 2002.

[19] Isak Jonsson and Bo Kågström. Recursive Blocked Algorithms for Solving Triangular Systems — Part II: Two-Sided and Generalized Sylvester and Lyapunov Matrix Equations. *ACM Transactions on Mathematical Software*, 28(4):416–435, December 2002.

[20] Victor J. Katz. *A history of mathematics, an introduction, 2nd edition.* Addison–Wesley, 1998.

[21] Morris Kline. *Mathematical thought from ancient to modern times*, volume x. Oxford University Press, 1990.

[22] 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*, 24(3):268–302.

[23] Neungsoo Park, Bo Hong, and Viktor K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003.

[24] Xiaobai Sun, Enrique S. Quintana, Gregorio Quintana, and Robert van de Geijn. Efficient Matrix Inversion via Gauss–Jordan Elimination and its Parallelization. Technical report, CS, University of Texas at Austin, Austin, Texas, US, 1998.

[25] Sivan Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[26] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. http://www.cs.utsa.edu/ whaley/papers/spercw04.ps.

[27] Wikipedia. Master Theorem. See website: `http://en.wikipedia.org/wiki/Master\_theorem`. Visited November 2005.