Master's Thesis: Explicitly Blocked Algorithms for Some Triangular Discrete-time Matrix Equations on Shared Memory Platforms

Roland Lidström
Umeå University,
SE-901 87 Umeå, Sweden.
dva00rlm@cs.umu.se

Abstract

This thesis investigates the scalability of explicitly blocked algorithms for solving discrete-time matrix equations for shared memory environments when implemented using high-performance GEMM-updates, a fast kernel solver from the HPC library RECSY and different blocksizes. The algorithms have been parallelized using OpenMP.

We consider two different algorithms, which uses two different strategies for traversal of the righthand side matrix. We also show by numerical experiments that the second algorithm with the more advanced anti-diagonal solving method is the most efficient of the two. Both serially as well as parallel.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Latin for womb is *matrix*. Generally, in English it means anyplace in which something is formed. Mathematical matrices has its origin in the study of systems of simultaneous linear equations. The first known example of matrix methods are from ancient China [11]. They where used to solve linear equations and are almost identical to how we solve linear equations today [21]. A matrix in its most basic form is where the number of unknowns equals the number of equations. The matrix in Table 1.1 has four rows and four columns, but the rows and columns in a matrix does not have to be equal.

Table 1.1: Mathematical Matrix

5.33	0.198	0.397	0.066
0.412	12.7	5.443	0.935
1.432	0.02	0.325	0.061
3.14	1.22	11.896	2.863

In more recent times James Joseph Sylvester(1814-1897) coined the actual term matrix. He also made many discoveries in matrix theory. In fact so many that some objects were named after him [14, 16].

In this Master's Thesis we consider and implement a blocked shared memory solver for the discrete-time Sylvester equation (SYDT),

$$AXB^T - X = C, \tag{1.1}$$

where A of size $m \times m$ and B of size $n \times n$ are matrices in real Schur form. C and X are matrices with real entries of size $m \times n$, typically X overwrites C on output from a solving algorithm.

The reason for this work is to see if recently developed algorithms [23] for solving Equation (1.1) on distributed memory machines can be ported to a shared memory machine using a fast kernel solver from the HPC library RECSY [26]. It is also of interest to investigate how efficient and scalable such an algorithm

would be [17]. The difficulty in getting great speedup when parallelizing explicitly blocked Sylvester-type matrix equation solvers, lie in the high level of data dependency in the problem. The speedup for ScaLAPACK-style algorithms in a distributed memory environment is usually an integer multiple of $\sqrt{P}$ where P is the number of processors used to solve the problem [23]. A multithreaded shared memory implementation was provided in [26]. Unfortunately it did not have as high speedup as was hoped.

This thesis tries to exploit the explicitly blocked algorithm from the distributed memory environment with the HPC library RECSY as kernel solver, to see if this makes for good speedup.

Chapter 2

In-depth study

In this Master's Thesis we have studied Fortran, shared memory machines and OpenMP as in-depth study. BLAS, RECSY and GEMM-updates have also been investigated for the in-depth study. The subject of shared memory programming was also investigated.

We began this work by studying FORTRAN and OpenMP. We also had to take a look at Linear Algebra to refresh the math dealing with matrices. When that was done, we dived into all the articles that have been written on the actual subject. Small tests with FORTRAN and OpenMP was conducted at an early stage and also test with the math software MATLAB to try out the algorithms. When this had been done, the implementation part in FORTRAN began. This part took a long time.

2.1 Fortran history

Fortran began its life at IBM in 1954 when John Bakus and his team started to design the FORmula TRANslator System. It is known as FORTRAN 0. All work was done in assembly code or machine code so the work went on slowly but was finished in 1957.

Fortran I, the successor of Fortran 0, had many special features special for the IBM 704 processor, the computer it was designed for. FORTRAN II included separate compilation of subroutines and was released in 1958. FORTRAN III was never released to the public. It made it possible to use of assembly code in the middle of the FORTRAN code. This type of programming can be more efficient but you lose both portability and easy of use.

FORTRAN IV was released in 1962. It was mostly consider a cleaner version of FORTRAN II and improvements was made to the COMMON and EQUIVALENCE statements. Other improvements was also done to remove machine-dependent language constructs [10]. FORTRAN IV was to remain the standard until the ANSI FORTRAN standard was released in 1977 [20], also known as FORTRAN 77.

The first FORTRAN standard (X3.9-1966) is known as FORTRAN 66. This standard is very short, only 36 pages. The standard for FORTRAN 77 consisted of 300 pages and for FORTRAN 90 almost 400 pages.

While FORTRAN 77 (X3.9-1978) was under development, it became clear that more facilities in the area of input/output was needed. This lead to new features for file handling. This was a big leap in the development of Fortran as a important language for I/O. Automatic array facilities where considered over a period of four year, but the array function was rejected. Today many modern FORTRAN 77 compilers support this extension, but it is not part of the FORTRAN 77 standard.

One feature of FORTRAN 66 that was removed in FORTRAN 77 was the 'Hollerith'. Hollerith is named after the inventor of hole cards Herman Hollerith. It was a way to store strings on hole cards [7], and with the invention of magnetic tapes, hole cards was obsolete and so was the function. The removal of this function was a more radical approach then the standard committee have taken with how FORTRAN 90 handle removal of obsolete functions. In FORTRAN 90 no features have been removed but only marked as candidates for removal in the next version of FORTRAN.

The work on FORTRAN 90 started in 1978. It was a common opinion that a third revision of FORTRAN was needed to make the language more responsive to changes in hardware. The many feature requests that had been received during FORTRAN 77's comment period added to this opinion.

To determine the needs of the users from a modern FORTRAN, there where many tutorials and surveys in the community. Between 1978-1985 the standard committee laid the foundation for the core of Fortran. When the user community realized that it was possible to have a modern FORTRAN, interest in the project increased. The project gained many members and with more members objections surfaced.

Many of the new members were more politically skilled then technically involved. It is the general rule that standards committees should try to reach consensus before presenting a new standard. Many of the new members had objections, which resulted in many discussions. After three years of discussions that did not lead to any large changes being made to the standard. The standard was now called 8x with the hope of it being release as Fortran 88. Due to all the infighting in the standards committee no consensus was ever reached.

After four years of public review the standard (FORTRAN 90) was finally finished in 1990, but further delayed to February 1991.

FORTRAN 90 is a superset of FORTRAN 77. No features of FORTRAN 77 was removed in FORTRAN 90.

FORTRAN 95 was published in 15 December 1997 and FORTRAN 2003 was published 18 November 2004 [2]. It contains only minor changes to the language compared to FORTRAN 90.

2.2 Fortran in short [6]

- Dates back to 1954
- Fortran II, IV, 66 (released in 1972)
- Fortran 77 (released in 1980)
- Fortran 90 (released in 1991)
- HPF (High Performance Fortran) (not an official standard)
- Fortran 95 (released in December 1997) - largely a "bug-fix" release
- Fortran 2003 (released in November 2004)

2.3 Shared Memory machines and OpenMP

A normal PC has one processor that executes a program that resides in the main memory of the PC. The processor access the program in memory by knowing its memory address. A memory address is a number and this number tells the processor where in memory to look.

In a shared memory (SM) machine, many processors are accessing the same memory address space. They can all access any address in the memory, but of course not at the same time.

The connection between the processor and the memory is called interconnection network. For a small number of processors (say eight) this is usually a single bus architecture. This means that all processors and memory is attached with the same set of wires. This architecture can only support a small number of processors because the bus only allows one processor at a time to use the bus, see Figure 2.1. If the system shall consist of a large number of processor, the need to construct more specialized interconnection networks arise. All this means that large SM machines is specially designed computer systems. This special design is very costly and have been the biggest hindrance for SM machines to dominate parallel computations.

The other way to construct parallel machines are called distributed memory (DM) machines. In this setup, each machine has its own memory and the processors exchange information via message passing. This exchange of information is done through a network.

Each processor plus its attached memory is called a node. A node is usually a selfcontained unit, similar to a tabletop PC. All the nodes and the network that connects them are sometimes called a cluster. There are nothing practical to stop the cluster to be as big as one can afford, but with every added node the strain on the network increases. The time it takes for the processors to exchange information over the network is the main bottleneck for the distributed memory machines. But a normal cluster usually contains a lot more processors then a regular SM machine. Thus, the average cluster have a lot more computational power then the regular SM machine.

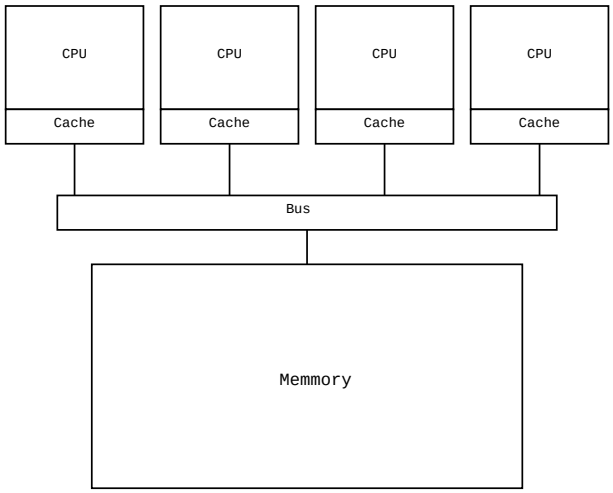

Figure 2.1: SM machine with single bus architecture.

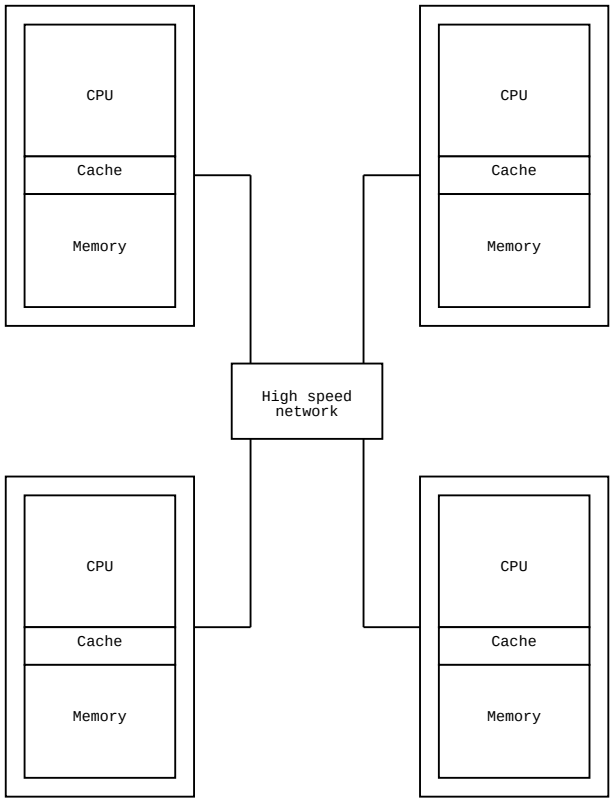

Figure 2.2: Distributed Memory Machine.

2.4 Programming in shared memory in general

Shared memory machines can be divided in two main classes, UMA and NUMA. Uniform Memory Access (UMA) is the most common SM machine. It have identical processors. The processors have equal access and access time to all parts of the common memory. Non-Uniform Memory Access (NUMA) is usually made by linking two or more SM machines. This makes the accesstime to the memories diffrent for diffrent processors. The memory in NUMAs are in general arranged in a hierarchical manner.

This work is conducted on a UMA machine.

The main advantage of programming in a shared memory environment is the user-friendly global address space perspective to the memory. This makes the datasharing between tasks fast and uniform when it comes to the communication cost.

The shared address space makes asynchronous reading and writing of memory possible, therefor mechanisms to control access to the shared memory is necessary. These exist in the form of locks and semaphores. The lack of explicit communication between tasks makes software development simplified, however a disadvantage with this model is that it becomes more difficult to manage *spatial locality*. Spatial locality is when you access one element in a local cache it is likely that you will also access the element next to the accessed element and it resides in the cache as well.

2.5 Implementations of Shared memory programming

When it comes to the actual implementations in this area, we have three main alternatives on the market today. They are as follows.

Parallelizing compiler

Here the process is fully automatic. The compiler analyzes the source code and reveals opportunities for parallelization. It is also important that the compilers analysis recognizes where parallelism improve performance. This is done with the help of identifying inhibitors and cost weighting of these. The most common target for automatic parallelization are loops. Most available tools for this type of parallelizing compilers are for FORTRAN [8].

OpenMP

OpenMP (see also 2.7) is a programmer directed parallelizing process. The programmer uses compiler directives to explicitly tell the compiler where and how to parallelize. It can be used with some form of automatic parallelization too.

The used compiler must support OpenMP, otherwise the parallelizing directives will be ignored.

Posix Threads

The standard way to program threads are to use the Posix Thread API, also known as Pthreads. Pthreads are supported by most vendors. The greatest advantage of using threads is that it allows the programmer to control the parallel execution in detail. All the threads are explicitly created by the programmer, this gives the programmer a very high level of control over all the parallel processes [1]. This level of control means that Pthreads requires more lines of code. It is this increase in complexity of the software code that increase the chance for bugs [27].

When using Pthreads, much of the bugs surface around synchronizing the threads. Data access and the scheduling of the threads are usual areas where problems arise [1].

2.6 Shared memory machines today

AMD has recently shown that the market is moving towards SM parallelism for household machines and small business machines. With this in mind the need for low cost parallelism is going to grow [3]. Many of the programs that are being executed on serial processors could make great use of a parallel SM machine. But the great mass of programs will only move in this direction if the cost in man hours to parallelize programs is low. Because the complexity to use Posix threads is great, that is not the way to go (see 2.5).

Open-Multi-Processing (OpenMP) will make the transition for many serial algorithms to a more parallel approach much smoother. OpenMP specifies a set of compiler directives and library routines that can be used for shared memory parallelism. There is also a few environment variables to help with control of the parallelism. OpenMP currently supports the languages Fortran and C/C++. The compiler directives in OpenMP defines parallel regions, work sharing and synchronization. It also include support for sharing and privatization of data. With the help of the library routines, the program can control and query the run-time execution environment. General purpose locking routines and portable timer routines are also parts of the library. The environment variables help with the runtime execution environment.

It was the common practice that each vendor developed their own parallel code library for use with its machines. The library was then used by the programmer that wanted to use the vendors machine efficiently. The side effect of this was that it was very costly to move to a new machine. All programs had to be rewritten. Sometimes from scratch. This is not very practical and soon attempts to make a standard started. An early attempts at a SMP standard (ANSI X3H5) was never fruitful because it was not supported by the vendors. The distributed

memory machines was also on the rise and their standards (PVM,MPI) looked more attractive then SMP machines, at that time.

A new effort to develop a standard in a shared-memory programming interface was started in 1997. This new attempt was fueled by a new interest from vendors to make SMP machines. They also had the opinion that parallelization of programs using message passing interfaces was to complicated.

The result was OpenMP which is now considered the de-facto standard by the industry. The standard specifies how to achieve shared memory parallelism in Fortran and C/C++ [4].

The biggest advantages of writing code in OpenMP is that the code is semantically consistent with the serial equivalent. It is actually hard to write code that is not. This point is important to developers that need to support both serial and parallel versions of their code.

2.7 OpenMP Programming in detail

To write OpenMP(OMP) code in Fortran programs you need an OMP-aware compiler. Then, to actually make your code parallel is rather simple if you have a serially functional code of what you want to do.

OMP works with directives that are hidden from a compiler that is not OMP aware. This is done by making the first character of each line, that holds a OMP directive, into a comment character. So a normal compiler would treat the entire line as a comment. A compiler that can handle OMP on the other hand will identify the line and process it.

The most important OMP directive is the *parallel region* constructor. In Fortran it looks like this.

```
!$OMP PARALLEL
....
!$OMP END PARALLEL
```

The code between the directive pair will be executed by all threads. The section between the directives is therefore known as a parallel region. The code before and after is only executed by a single thread and is therefore called serial region. The thread running in the serial region is labeled master when it comes into a parallel region.

The master thread creates the other threads in the parallel region but they all function the same, computational wise, inside the region.

It is possible to add clauses in the beginning of the parallel region. The clauses can specify the way in which the parallel regions should execute. It can be especially useful for how the region shall handle variables, threads and any special treatment for variables. If there is some special way in which how the iterations should be distributed, this is also handled with clauses.

At the end of the parallel region all threads except the master thread are destroyed. This include the threads private variables. The master thread waits for all the other threads to exit before it continues. This creates a implied synchronization of the code, which can be costly.

There is actually only two rules to follow when making code OMP compliant. The first rule is that all parallel regions have to contain a start directive and an end directive.

The second rule is that there can not be any jump in and out of the serial code. That is, the parallel region must be a structured block of code.

2.8 BLAS, GEMM and blocking of matrices

BLAS (Basic Linear Algebra Subprograms) was developed in the late 1970s.

The first version of BLAS was called level 1 and had vector-scalar and vector-vector operations. The memory on computers in the late 1970s operated with the same speed as the cpu. This meant that flop count was an accurate estimate of performance.

BLAS level 2 was developed in the 1980s to early 1990s. The trend in mainframe computing was then to use vector architecture based machines. A vector machine can simultaneous perform an operation on every element in a vector of certain length. BLAS level 2 performs matrix-vector operations and made use of this feature.

In the early 1990s, processor speed started to increase while memory speed did not. This made memory access a costly operation. Processor developers created *cache memories* to minimize slow memory access. A cache memory is a small and fast memory, that holds the last data the processor needed. Often, the same data is needed several times for the same computation. If the data is then already in cache a slow memory access is avoided. This is called *temporal locality*, and is one of the main advantages of cache memories.

Later systems have begun to make use of several levels of cache memory, where every level have different size and speed. This type of architecture is called *hierarchical memory systems*. To get the most out of a hierarchal memory system with matrices in mind, one must make use of blocking. The idea here is to fit each block into a level of the cache memory. If this is accomplished, the temporal locality gain is optimal.

Blas level 3 was developed to work with matrix-matrix operations. One commonly used operation is the GEMM (GEneral Matrix Multiply and add), see Equation (2.1) [9].

$$C \leftarrow \beta C + \alpha AB \tag{2.1}$$

The recent development in processor architecture have increased the dependence on the cache memory hierarchy. Which has lead to the development of level-3

based software libraries as LAPACK [15] and RECSY [26].

2.9 Limitations

The adopted limitations in this thesis are that the A and B matrices in Equation (1.1) are expected to be in square form (i.e. NxN). The given blocksize used in the blocking is expected to be such that N is an integer multiple of the blocksize. There have also been very little investigation in false sharing.

These limitations have been taken to save time and they do not change the outcome of the tests.

Chapter 3

RECSY - HPC library for Sylvester-Type matrix equations

Solving triangular matrix equations is one step in the classical Bartels – Stewart method for solving standard continuous time Sylvester equation (SYCT) [5], see Equation (3.1). The Bartels – Stewarts method can be generalized to SYDT and many similar problems.

$$AX - XB = C \tag{3.1}$$

Where A of size $m \times m$ and B of size $n \times n$ are in real Schur form. X and C are matrices with real entries of size $m \times n$.

The idea behind the HPC library RECSY is to use recursion to block the matrices in a matrix equation. This will make it possible to automatically match the size of the blocks to the cache sizes. If this is fulfilled, the gain in speed for the solver is great [24, 25].

A visualization of this can be found in Figure 3.1.

RECSY is only one part of solving the complete dense matrix equation problem. When strictly looking at flop count, the factorization part in the Bartels – Stewart solving method, should now be the major time consumer. The majority parts of computations in RECSY are done as GEMM operations. RECSY can solve eight different matrix equations [26], which are displayed in Figure 3.2. The acronyms in Figure 3.2 are the ones we use. RECSY is implemented in FORTRAN 90 and has been parallelized for SM machines using OpenMP. The gain in performance can be further improved be linking with SMP-BLAS.

There are three major steps in Bartel – Stewart method of solving Sylvester-type matrix equations. Step 1 is to transform the matrices to (generalized) Shur form. In step 2 a triangular solver (e.g. from LAPACK or RECSY) is used to solve

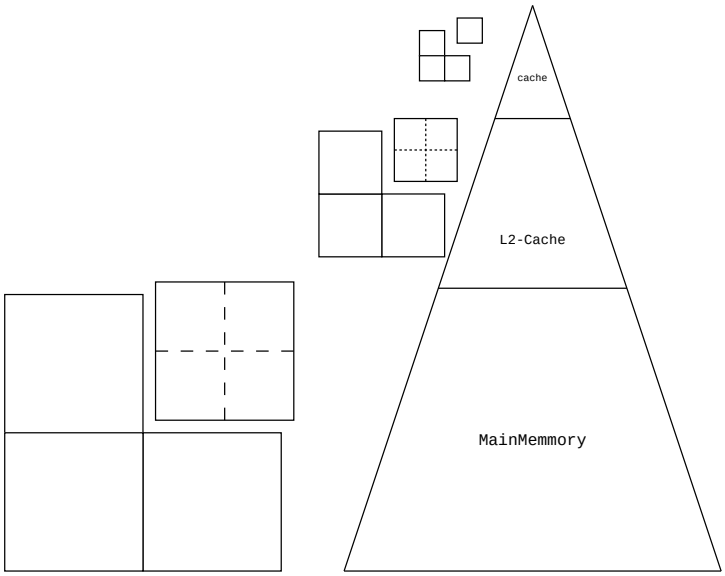

Figure 3.1: The recursive splitting of the matrices performed by RECSY.

the reduced triangular matrix equation. Then, in step 3 the reduced triangular system is transformed back to the original coordinate system.

RECSY considers three different cases when it is splitting the matrices in SYDT. As in Equation (1.1), A is of size $M \times M$ and B is of size $N \times N$ and X is of size $M \times N$. The first case considered is for $1 \leq N \leq \frac{M}{2}$. Then A is split by rows and columns and C by rows, see Equation (3.2).

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} B^T - \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}. \tag{3.2}$$

This can be rewritten as

Figure 3.2: The eight matrix equations considered in RECSY, op(A) means either A or Transpose A.

Name	equation	acronym
Standard Sylvester (CT)	$op(A)X \pm Xop(B) = C$	SYCT
Standard Lyapunov (CT)	$op(A)X + Xop(A^T) = C$	LYCT
Generalized coupled Sylvester	$(op(A)X \pm Yop(B) = C,$ $op(D)X \pm Yop(E) = F)$	GCSY
Standard Sylvester (DT)	$op(A)Xop(B) \pm X = C$	SYDT
Standard Lyapunov (DT)	$op(A)Xop(A^T) - X = C$	LYDT
Generalized Sylvester	$op(A)Xop(B) \pm op(C)Xop(D) = E$	GSYL
Generalized Lyapunov (CT)	$op(A)Xop(A^T) - op(E)Xop(E^T) = C$	GLYCY
Generalized Lyapunov (DT)	$op(A)Xop(E^T) + op(E)Xop(A^T) = C$	GLYDT

$$\begin{aligned} A_{11}X_1B^T - X_1 &= C_1 - A_{12}X_2B^T + X_2 \\ A_{22}X_2B^T - X_2 &= C_2. \end{aligned} \tag{3.3}$$

This splits the problem into two subproblems of triangular SYDT equations. Here X_2 is solved first and C_1 updated with respect to X_2 and then is X_1 solved for.

For the second case $1 \leq M \leq \frac{N}{2}$, B is split by rows and columns and C by columns only, see Equation (3.4). It is solved similar as the first case.

$$A \begin{bmatrix} X_1 & X_2 \end{bmatrix} \begin{bmatrix} B_{11}^T & 0 \\ B_{12}^T & B_{22}^T \end{bmatrix} - \begin{bmatrix} X_1 & X_2 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \end{bmatrix}. \tag{3.4}$$

In the third case $\frac{N}{2} < M < 2N$, all matrices are split by rows and columns.

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \begin{bmatrix} B_{11}^T & 0 \\ B_{12}^T & B_{22}^T \end{bmatrix} - \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}. \tag{3.5}$$

By splitting in both dimensions we get four new SYDT equations

$$\begin{aligned} A_{11}X_{11}B_{11}^T - X_{11} &= C_{11} - A_{12}(X_{21}B_{11}^T + X_{22}B_{12}^T) - A_{11}X_{12}B_{12}^T \\ A_{22}X_{12}B_{11}^T - X_{12} &= C_{12} - A_{12}X_{22}B_{22}^T \\ A_{22}X_{21}B_{22}^T - X_{21} &= C_{21} - A_{22}X_{22}B_{12}^T \\ A_{22}X_{22}B_{22}^T - X_{22} &= C_{22}. \end{aligned} \tag{3.6}$$

RECSY starts by solving X_{22} recursively. Then C_{12} and C_{21} is updated with respect to X_{22}. This makes X_{12} and X_{21} solvable. Both updates and solves are independent operations and can be done in parallel. C_{11} is updated with respect to X_{21},X_{22} and X_{12}. Finally the last equation is solved for X_{11}.

When M and N for the matrix in the recursion becomes smaller then four, RECSY uses a kernel solver to solve the leaf in the recursion tree. A Kronecker product representation are done for each matrix equation. The linear system $Zx = c$ are then solved using a optimized kernel with partial pivoting and overflow guarding. If there is risk for singularity or overflow the kernel aborts. When the solver aborts the routine backtracks and constructs a new Kronecker product representation, witch is solved with complete pivoting [26, 23, 28].

In Equation (3.6) some matrix sums products can be arranged such that no unnecessary computations are done [23]. However this is not done in RECSY.

Chapter 4

Implementation

We have considered two different blocked algorithms in this thesis. Both algorithms are based on Equation (4.1) below. The difference between the two algorithms are how the computations are organized.

4.1 Our first Algorithm

The first algorithm we have implemented is described in [23]. When blocked, Equation (1.1) can be rewritten as

$$A_{ii}X_{ij}B_{jj}^T - X_{ij} = C_{ij} - \sum_{k=i}^{D_a}\sum_{l=j}^{D_b} A_{ik}X_{kl}B_{jl}^T,\ (k,l) \neq (i,j). \tag{4.1}$$

In Equation (4.1) D_a are the number of blocks in a row and D_b are the number of blocks in a column in the blocked matrix C/X.

This leads to many smaller SYDT equations. When solving Equation (4.1) for the X_{ij}, we begin with the bottom right block in C/X. This because the bottom right blocks solution do not depend on any other blocks solution.

But all the X_{kl} with $1 < k < i$ and $1 < l < j$ are dependent on the result from X_{ij}. One can see that the speed of the updating phase is very important. If Equation (4.1) is implemented directly we will perform redundant computations. To avoid this, we make use of a temporary storage for the matrix products that are needed several times. This gives a complexity of $O(m^2n + mn^2)$ [23], which is equivalent to the computational cost of solving the equation without blocking.

The implementation

We have based the implementation on the algorithm in Figure 4.4 from [23]. First we solve the bottom right block. The result is then used to update a temporary block sized array.
This is then done for the whole row. Of importance to note is that not the entire temporary matrix is updated every time, see Figure 4.2. Only those blocks in the temporary matrix that are to the left of the solved block is updated.

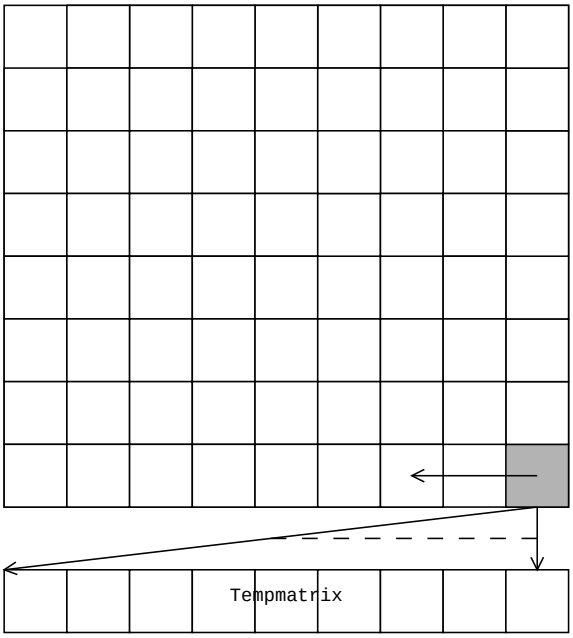

Figure 4.1: Solve the bottom left block first, and update tempblockarray.

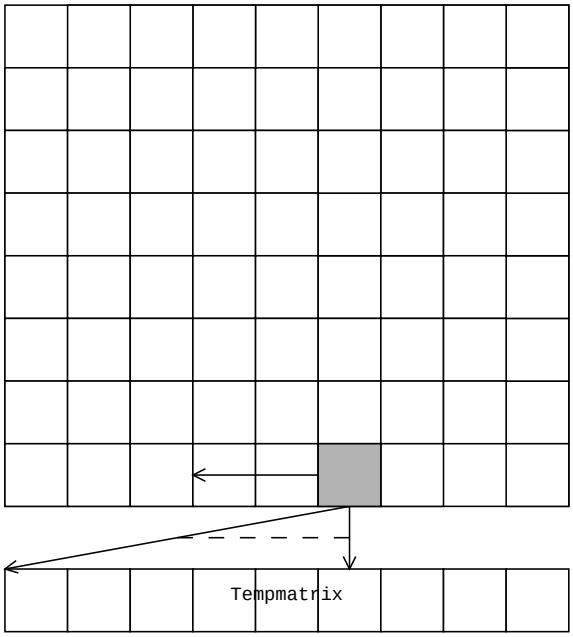

Figure 4.2: Do this for the whole row.

When the entire row has been solved for, the temporary matrix is used to update the solution matrix, see Figure 4.3. The update is only applied to the right hand

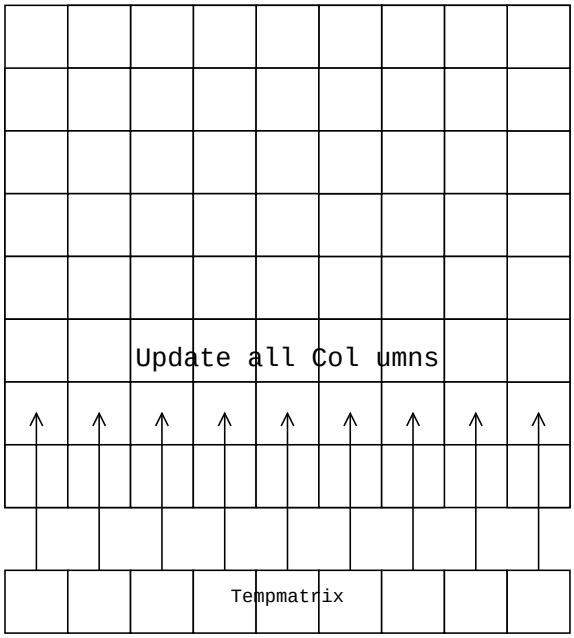

Figure 4.3: When one row is done, update the whole matrix.

side matrix above the row that was just solved for. This because we solve the row directly on the matrix. All updates here are independent operations and can be performed in parallel. All the subsolutions of the subsystems are computed with the RECSYDT routine from RECSY. The updates are computed with the DGEMM routine from the BLAS library.

The algorithm as whole is outlined in Figure 4.4.

4.2 Our second Algorithm

The second algorithm is blocked as the first algorithm and works in a similar way as the first algorithm. The difference here is that the solving of blocks are done in a diagonal fashion [28]. This makes the temporary matrix as big as the original right hand side matrix. The idea for diagonal solving is to increase the parallel part of the algorithm. This makes the algorithm more suited for a distributed memory machine system.

The algorithm is also suited for shared memory machines because the increase of parallelism improves the performance and efficiency even for shared memory machines, as the results of the numerical tests will show.

The implementation

The second algorithm we have implemented is also based on [23], see Figure 4.5. The updates are performed on the columns above the blocks that where

Figure 4.4: Block algorithm for $AXB^T - X = C$ (SYDT), A and B in real Schur form.

```
for i = D_a, 1, -1
    for j = D_b, 1, -1
        {Solve the (i,j)^th subsystem}
        Solve A_ii X_ij B_jj^T - X_ij = C_ij for X_ij
        {Update all blocks E_k, such that k ∈ [1, jend]}
        jend = j
        if (i = 1) jend = j - 1
        for k = 1, jend, 1
            {If going for a new block row, set E_k to zero }
            if (j = D_b) Set E_k = [0]_{mb×nb}
            Compute E_k = E_k + X_ij B_kj^T
        end
        {If we are on the end of a blockrow, update C_lm, l ∈ [1, i-1], p ∈ [1, D_b] }
        if (j = 1)
           for l = 1, i - 1, 1
              for p = 1, D_b, 1
                 Compute C_lp = C_lp - A_li E_p
              end
           end
        {If we are not on the end of a blockrow, prepare for next solve}
        else
           Compute C_{i,j-1} = C_{i,j-1} - A_ii E_{j-1}
        endif
    end
end
```

currently solved for, see Figure 4.6.

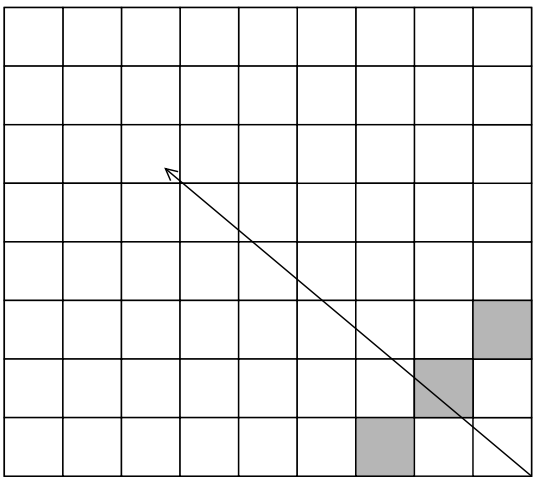

Figure 4.5: Solve the blocks in diagonal.

The updates are also performed in parallel over the temporary matrix columns.

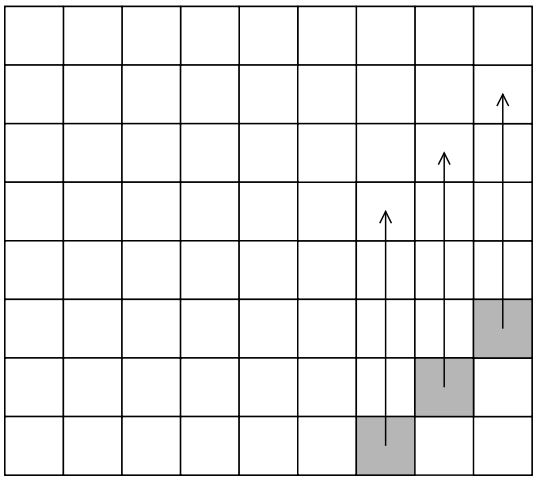

Figure 4.6: When one row is done, update the whole matrix.

Figure 4.7: Parallel block algorithm for the diagonal solving $AXB^T - X = C$: SM machine implementation.

for $k = 1$, number of block diagonals in C
    {Solve the current block diagonal in parallel}
    Solve $A_{ii}X_{ij}B_{jj}^T - X_{ij} = C_{ij}$ for X_{ij}
    {Update all $E_{ik}, k \in [1, j]$ in parallel}
    $jend = j$
    if($i = 1$) then $jend = j - 1$
    for $k = 1, jend, 1$
        if ($j = D_b$) then Set $E_{ik} = [0]_{mb \times nb}$
        Compute $E_{ik} = E_{ik} + X_{ij}B_{kj}^T$
    end
    {Update block column j in parallel}
    for $l = 1, i - 1, 1$
        Compute $C_{lj} = C_{lj} - A_{li}E_{ij}$
    end
    {Prepare to solve for next block diagonal in parallel}
    if ($j > 1$) then
        Compute $C_{i,j-1} = C_{i,j-1} - A_{ii}E_{i,j-1}$
    end
end

4.3 Experiments

The ideas in this thesis has been implemented in Fortran 90. The Fortran compiler used is the IBM Fortran 90 compliant XLF95_r compiler. The testing

have been done on an IBM Power3 system. The parallelization have been done with OpenMP and the GEMM-updates are done with IBM's implementation of BLAS.

All experiments were carried out on the IBM Power3 based SMP system at HPC2N [12]. The system known as *chips* have four 375 MHz Power3 processors and 4096 MB of memory. Due to the fact that all system libraries have been compiled to 32-bit format, only 2096 MB is available to a single user. This fact have limited the tests to the maximum matrix size of 6000×6000 .

In Table 4.1 we see that the the blocksize makes a big difference in the time it takes to solve the problem. Through experiments we found out that a blocksize of 240 gave the fastest solution on one processor with algorithm 1. This blocksize was then used for the other experiments, except for the tests associated with Table 4.2, where we see that the fastest blocksize for one processor is not the optimal blocksize for four processors. In Tables 4.3 and 4.4 we present how the two algorithms behaves with different problemsizes on one processor.

In Tables 4.5 – 4.8 we present how the two different algorithms behaves with different problemsizes on two and four processors. We also show how the efficiency behaves in these tables.

In Table 4.9 and Table 4.10 we see how RECSYs SMP implementation behaves on one and four processors.

In all tables res. stands for

$$\| C + X - AXB^T \|_F \ . \tag{4.2}$$

Where we measure the backward error with the Frobenius norm [19, 22].

4.4 Results

From the Tables 4.5 – 4.8 and in Figure 4.8 we see that both algorithms have high efficiency on two processors. Algorithm 2 is always faster. In Figure 4.8 we see that algorithm 2 have a more graceful behavior when the problem size increases.

Table 4.1: Algorithm 1 on one processor and matrixsize of 6000.

blocksize	res.	Time(s)
50	1.2E-11	539
100	1.7E-11	442
200	2.1E-11	418
300	2.4E-11	426
500	2.7E-11	440
600	1.2E-11	457
1500	1.8E-11	568
2000	2.2E-11	594
3000	2.6E-11	666

Table 4.2: Algorithm 2 on four processors and matrixsize of 6000.

blocksize	res.	Time(s)
50	1.2E-11	194
100	1.7E-11	160
200	2.1E-11	169
300	2.4E-11	181
500	2.7E-11	209

One thing that is very interesting here is that in Table 4.3 the increase in time when the matrix increase from 5040 to 5280 is very small. The exact reason for this is something we have not investigated thoroughly. But the most likely explanation is that the blocking size fitted very well with the machines cache size and this made the computation of the updates work very well.

The same effect can be seen in Table 4.4 where the time actually decreases with 11 seconds. Why algorithm 2 is faster on one processor then algorithm 1 is most likely due to differences in the code implementation of the parallelization in the two algorithms. This is further explained later.

4.5 Analysis

In Figure 4.8 the cache effect can be seen even more clearly. When the matrix reaches a size of 5040 there is a jump in speed. This effect can be seen in all the tables. The reason for this effect is most likely due to cache effects. The really interesting thing to see is that the effect is larger when only one processor is in use then four with algorithm 2. There is also a bigger effect for algorithm 2 then for algorithm 1.

As can be expected, the efficiency for two processors are higher than for four processors. This is natural with the small problemsizes. What is interesting is how big difference there is between algorithm 1 and the algorithm 2 on two processors. Usage of two processors seams more fitting for the problemsizes. This can be seen in that the efficiency quickly levels off and only slowly increases with the increase in problem size. When four processors are used the efficiency increase is much larger. The efficiency for algorithm 2 on four processors goes from 31% to 62%.

It can also be seen that the problem size need to be relatively large for the efficiency to creep above 50%. Unfortunate there was no possibility to increase the problemsize larger then 6000 due to hardware limitations. Despite this it is possible to see that algorithm 2 is scalable. When two processors are used the efficiency is 62% for a problemsize of 1200 and when we have four processors and a problem size of 6000 the efficiency also goes up to 62%.

It is interesting to see that the efficiency for the SMP implementation of RECSY hovers around 60 – 64 %. The difference in time that we see for SMP_RECSY

Table 4.3: Algorithm 1 solving on one Processor.

m=n	res.	Time(s)
960	8.4E-13	4
1200	1.4E-12	7
1440	2.1E-12	11
1680	2.8E-12	17
1920	3.6E-12	23
2160	4.6E-12	31
2400	5.6E-12	42
2640	6.7E-12	53
2880	7.8E-12	66
3120	9.1E-12	86
3360	1.0E-11	99
3600	1.2E-11	118
3840	1.3E-11	142
4080	1.5E-11	165
4320	1.7E-11	193
4560	1.9E-11	223
4800	2.0E-11	257
5040	2.3E-11	332
5280	2.5E-11	333
5520	2.7E-11	374
5760	2.9E-11	422
6000	3.2E-11	469

Table 4.4: Algorithm 2 solving on one processor.

m=n	res	Time(s)
960	8.5E-13	3
1200	1.4E-12	5
1440	2.1E-12	8
1680	2.8E-12	12
1920	3.7E-12	18
2160	4.6E-12	24
2400	5.6E-12	32
2640	6.7E-12	41
2880	7.9E-12	53
3120	9.1E-11	71
3360	1.0E-11	81
3600	1.2E-11	98
3840	1.3E-11	120
4080	1.5E-11	139
4320	1.7E-11	163
4560	1.9E-11	190
4800	2.0E-11	222
5040	2.3E-11	299
5280	2.5E-11	288
5520	2.7E-11	326
5760	2.9E-11	373
6000	3.2E-11	413

and algorithm 2 when they run on one processor is most likely due to the extra overhead that the recursion cause for SMP_RECSY.

The 70 second difference in solving time that we can see in Figure 4.8 between SMP_RECSY and algorithm 2 is not likely to decrease if we make the problem-size bigger. This due to the fact that the curves increase at different rates. This makes a strong case for explicitly blocked algorithms.

Table 4.5: Algorithm 1 solving on two processors.

m=n	res	Time(s)	Eff. %
960	8.4E-13	3	66.7
1200	1.4E-12	6	58.3
1440	2.1E-12	9	61.1
1680	2.8E-12	13	65.4
1920	3.6E-12	18	63.9
2160	4.6E-12	24	64.6
2400	5.6E-12	31	67.7
2640	6.7E-12	39	67.9
2880	7.8E-12	49	67.3
3120	9.1E-12	61	70.5
3360	1.0E-11	70	70.7
3600	1.2E-11	83	71.1
3840	1.3E-11	99	71.7
4080	1.5E-11	114	72.0
4320	1.7E-11	131	73.7
4560	1.9E-11	150	74.3
4800	2.1E-11	172	74.7
5040	2.3E-11	217	76.5
5280	2.5E-11	218	76.4
5520	2.7E-11	244	76.6
5760	2.9E-11	273	77.8
6000	3.2E-11	301	77.0

Table 4.6: Algorithm 2 solving on two processors.

m=n	res	Time(s)	Eff. %
960	8.5E-13	2	75.0
1200	1.4E-12	4	62.5
1440	2.1E-12	6	66.7
1680	2.8E-12	9	66.7
1920	3.7E-12	13	69.2
2160	4.6E-12	17	70.6
2400	5.6E-12	22	72.7
2640	6.7E-12	28	73.2
2880	7.9E-12	36	73.6
3120	9.1E-12	47	75.5
3360	1.0E-11	52	77.9
3600	1.2E-11	63	77.8
3840	1.3E-11	77	77.9
4080	1.5E-11	88	79.0
4320	1.7E-11	102	79.9
4560	1.9E-11	117	81.2
4800	2.0E-11	136	81.6
5040	2.3E-11	184	81.3
5280	2.5E-11	174	82.8
5520	2.7E-11	195	83.6
5760	2.9E-11	222	84.6
6000	3.2E-11	245	84.3

Table 4.7: Algorithm 1 solving on four processors.

m=n	res	Time(s)	Eff. %
960	8.4E-13	5	20.0
1200	1.4E-12	9	19.4
1440	2.1E-12	13	21.2
1680	2.8E-12	17	25.0
1920	3.6E-12	25	23.0
2160	4.6E-12	31	25.0
2400	5.6E-12	38	27.6
2640	6.7E-12	49	27.0
2880	7.8E-12	57	28.9
3120	9.1E-12	69	31.2
3360	1.0E-11	81	30.6
3600	1.2E-11	94	31.4
3840	1.3E-11	107	33.2
4080	1.5E-11	124	33.3
4320	1.7E-11	135	35.7
4560	1.9E-11	134	41.6
4800	2.1E-11	152	42.3
5040	2.3E-11	183	45.4
5280	2.5E-11	189	44.0
5520	2.7E-11	210	44.5
5760	2.9E-11	235	44.9
6000	3.2E-11	265	49.9

Table 4.8: Algorithm 2 solving on four processors.

m=n	res	Time(s)	Eff. %
960	8.5E-13	2	37.5
1200	1.4E-12	4	31.3
1440	2.1E-12	5	40.0
1680	2.8E-12	8	37.5
1920	3.7E-12	10	45.0
2160	4.6E-12	14	42.9
2400	5.6E-12	18	44.4
2640	6.7E-12	22	46.6
2880	7.9E-12	27	49.1
3120	9.1E-12	35	50.7
3360	1.0E-11	39	51.9
3600	1.2E-11	46	53.3
3840	1.3E-11	56	53.6
4080	1.5E-11	63	55.2
4320	1.7E-11	72	56.6
4560	1.9E-11	82	57.9
4800	2.1E-11	94	59.0
5040	2.3E-11	127	58.9
5280	2.5E-11	119	60.5
5520	2.7E-11	133	61.3
5760	2.9E-11	150	62.2
6000	3.2E-11	165	62.3

Table 4.9: SMP_RECSY on one processor.

m=n	Time(s)
960	3
1200	6
1440	10
1680	15
1920	23
2160	31
2400	43
2640	58
2880	74
3120	94
3360	114
3600	137
3840	172
4080	201
4320	235
4560	280
4800	324
5040	396
5280	437
5520	489
5760	565
6000	625

Table 4.10: SMP_RECSY on four processors.

m=n	Time(s)	Eff. %
960	1	75
1200	3	50
1440	4	62.5
1680	7	53.5
1920	10	57.5
2160	13	59.6
2400	17	63.2
2640	22	65.9
2880	30	61.7
3120	37	63.5
3360	46	62.0
3600	55	62.3
3840	68	63.2
4080	85	59.1
4320	94	62.5
4560	110	63.6
4800	126	64.3
5040	156	63.5
5280	169	64.6
5520	192	63.7
5760	221	63.9
6000	235	66.5

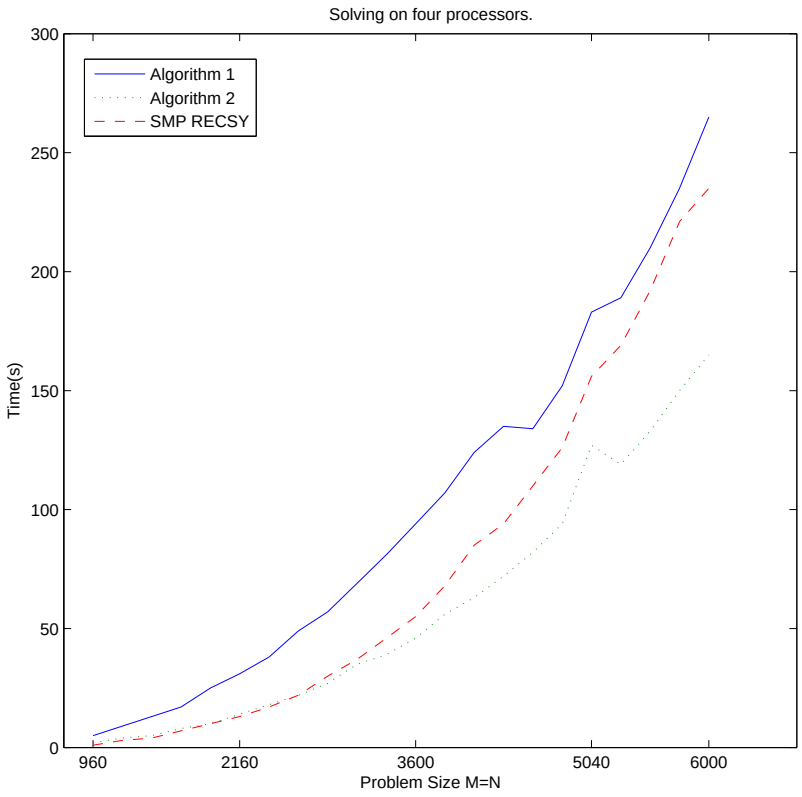

Figure 4.8: Comparison between Algorithm 1, Algorithm 2 and SMP RECSY

Chapter 5

Future Work

This work can be made more general. Right now the A and B matrices in Equation (1.1) are expected to be square and the blocksize is expected to be such that N is an integer multiple of the blocksize.

It would also be very interesting to see how false sharing effects the performance. From reading IBM documents on Processor optimization for AIX we have seen that there are many areas to explore on the subject [13, 18]. Of interest would also be to see how a more course grained approach to parallelism would behave.

To see if the efficiency goes as high for four processors as they do for two if the problemsize could be increased a lot would also be very interesting.

Chapter 6

Summary and conclusion

That the difference between the two algorithms was so big on one processor was a bit odd. That the difference is so big is probably due to the difference in coding style.

The cache-hit effect that can be seen in Figure 4.8 was also a bit of a surprise. This effect is most likely due to the blocksize fitting well with the cache and the hardware.

6.1 Final thoughts on coding the algorithms.

Algorithm 1 is done with a object oriented approach where the algorithm is divided up in modules. This caused a bit of a problem for the compiler to optimize the code. This due to the fact that in FORTRAN the different modules is first separately compiled and then linked together. This separate compilation makes it hard to optimize the code with the other modules. That the parallelization was made in the appropriate modules also caused problem for the optimization routine for the compiler. This had the effect that thread creation and destruction adds to the execution time. During code design we thought that this would not cause noticeable effect.

Algorithm 2 is all done in one module. This had the effect that the compiler had a easier time optimizing the code. That the algorithm inherently give the programmer advantages for parallelization had the effect that there is only one point of thread creation and destruction. This lowers the overhead for the algorithm and helps making algorithm 2 faster.

6.2 Thoughts on parallelization and FORTRAN

I have not read the advance course on parallelization, but now i am considering it.

My thoughts on FORTRAN have changed quite a bit. I did not know it when I started the master thesis but I had heard quite a bit of horror stories about the language.

But now when I have learned a bit and worked with it, i think it is quite nice. That OpenMP and FORTRAN fits very well together was also very nice and helped things along very much.

Chapter 7

Acknowledgments

I would like to thank Robert Granat for providing an interesting Master Thesis and for his tireless support, as the almost 100 email is evidence of. I would also like to thank Niklas Edmundsson at HPC2N for his support.

Finally I also would like to thank my mother for the homecooked meals when the morale was low.

Bibliography

[1] A. Grama, A. Gupta G. Karypis, V. K. *Introduction to Parallel Computing, Second Edition*. Addison Wesley, 2003.

[2] Adams, J. Controversy, compromise, modernization: From fortran to fortran 90. Web, 7 Dec. 2004. http://www.scd.ucar.edu/tcg/consweb/Fortran90/scnhist.html.

[3] Amd demonstrates world's first x86 dual-core processor. Web, 7 Dec. 2004. http://www.amd.com/us-en/0,,3715_11787,00.html .

[4] Openmp. Web, 7 Dec. 2004. http://www.openmp.org .

[5] Bartels, R. H., and Stewart, G. W. Solution of the matrix equation ax + xb = c. *ACM Communications 15*, 9 (1972), 820–826.

[6] A brief history of fortran. Web, 7 Dec. 2004. http://www-h.eng.cam.ac.uk/help/mjg17/f90/history.htm .

[7] David L. Mills, P. The anatomy of a hollerith card. Web, 17 Dec. 2004. http://www.eecis.udel.edu/ mills/ibmcard.html.

[8] Advanced parallelizing compiler. Web, 17 Dec. 2004. http://www.apc.waseda.ac.jp/.

[9] Blas (basic linear algebra subprograms). Web, 17 Dec. 2004. http:www.netlib.org/blas/.

[10] A brief history of fortran. Web, 14 Dec. 2004. http://www.ibiblio.org/pub/languages/fortran/ch1-1.html .

[11] Did you know. Web, 14 Dec. 2004. http://www.ualr.edu/ lasmoller/matrices.html .

[12] Hpc2n hardware resources. Web, 17 Dec. 2004. www.hpc2n.umu.se/resources/hardware.html.

[13] Ibm xl fortran compiler. Web, 17 Dec. 2004. https://support.iap.ac.cn/hpc/ibm/xlf.html .

[14] James joseph sylvester. Web, 14 Dec. 2004. http://www-history.mcs.st-and.ac.uk/history/Mathematicians/Sylvester.html .

[15] Lapack – linear algebra package. Web, 17 Dec. 2004. http:www.netlib.org/lapack/.

[16] Mathworld. Web, 14 Dec. 2004. http://mail.mcjh.kl.edu.tw/ chenkwn/mathword/m.html .

[17] Matrix algebra. Web, 14 Dec. 2004. http://www.sosmath.com/matrix/matrix.html .

[18] Performance management guide. Web, 17 Dec. 2004. http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixbman/prftungd/2365cf2.htm.

[19] Frobenius norm. Web, 18 Feb. 2005. http://mathworld.wolfram.com/FrobeniusNorm.html .

[20] History and philosophy of fortran. Web, 7 Dec. 2004. http://www.isi.edu/ iko/pl/hw3_fortran.html.

[21] Systems of linear equations. Web, 19 Jan. 2005. http://www.roma.unisa.edu.au/07305/syseqn.htm .

[22] Gene H. Golub, C. F. V. L. *Matrix Computations (Johns Hopkins Series in the Mathematical Sciences)*. Johns Hopkins University Press; 3rd edition, 1 Nov. 1996.

[23] Granat, R., and Kågström, B. Parallel scalapack-style solvers for the continuous-time and discrete-time sylvester/lyapunov matrix equations: algorithms, software and condition estimation. *In preparation, Department of Computing Science,Umeå University* (2004).

[24] Jonsson, I., and Kågström, B. Recursive blocked algorithms for solving triangular systems: Part I: one-sided and coupled Sylvester-type matrix equations. *ACM Transactions on Mathematical Software 28*, 4 (Dec. 2002), 392–415.

[25] Jonsson, I., and Kågström, B. 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 (Dec. 2002), 416–435.

[26] Jonsson, I., and Kågström, B. Recsy - high performance library for sylvester-type matrix equations. Web, 17 Dec. 2004. http://www.cs.umu.se/ isak/recsy/.

[27] Pfleeger, S. L. *Software Enginering: Theory and Practice (2nd Edition)*. Prentice Hall, 7 Feb. 2001.

[28] Poromaa, P. *Algorithms and Library Software for Sylvester Equations and Certain Eigenvalue Problems with Applications in Condition Estimation*. PhD thesis, Umeå University Sweden, 1997.

Appendix A

Algorithm 1

This is the code used for algorithm 1. It is divided into three modules. The division of modules are mapped after the three major parts of the algorithm. The first module is called gralg and it contains the solver part.

```
MODULE gralg

  PRIVATE i,j
CONTAINS

  SUBROUTINE grsolv(A, B, C, matrisSize, blockSize, LDA, LDB, LDC)
    USE bigupdate
    USE updaterow
    IMPLICIT NONE
    EXTERNAL recsydt


    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE,intent(in)  :: A
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE,intent(in)  :: B
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE  :: C
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE  :: E

    INTEGER,INTENT(in) :: matrisSize,blocksize
    INTEGER   :: jstart,jstop,istart,istop,blockrad,blockkolumn
INTEGER   :: antalblock,tmpJs,tmpje,jend
    !parametrar för recsydt
    INTEGER :: T = 3
    DOUBLE PRECISION :: scale
    INTEGER :: M
    INTEGER :: N
    INTEGER :: LDA  ! = matrisSize
    INTEGER :: LDB  ! = matrisSize
    INTEGER :: LDC  ! = matrisSize
    INTEGER :: info
   ALLOCATE(E(blocksize,matrissize))
```

```
      M   = blocksize
      N   = blocksize
      !
      scale = 0
      info  = 0
      antalblock = CEILING(REAL(matrisSize)/REAL(blockSize))
      !från nedrehögra hörnet i matrisSize X matrisSize stora matrisen
      DO blockkolumn = antalblock, 1, -1
         DO blockrad = antalblock, 1, -1
            Jstart   = (blockrad    -1)  * blockSize+ 1
            Istart   = (blockkolumn -1 ) * blocksize+ 1
            tmpJs    =  jstart - blocksize
            jend     =  blockrad !jend = j

            CALL recsydt(T,scale,M,N, A(istart,istart),LDA , &
                 B(jstart,jstart),LDB , &
                 C(istart,jstart),LDC ,info)

            IF(info .NE. 0) THEN
               WRITE(*,*)"FEL!! i recsy"
            END IF
            !om första på ny rad så töm
            if (blockkolumn == 1) then ! if(i=1  ) jend = j-1
               jend = blockrad - 1
            end if

            IF (blockrad == antalblock) THEN  !if j= Db Ek =0
               E(:,:) = 0.0D+00
            END IF
            !lilla updateringen
            CALL  updater(E,C,B,blocksize,jend,blockkolumn,blockrad,matrissize)

            IF(blockrad == 1) THEN
               !stora updateringen
               CALL  bigup(C, A, E, blocksize, antalblock, blockkolumn,matrissize)
            ELSE
               call dgemm('n','n',blocksize,blocksize,blocksize,-1.0D+00, &
                    A(istart,istart),matrissize, &
                    E(:,tmpjs)      ,blocksize, 1.0D+00, &
                    C(istart,tmpjs) ,matrissize )

            END IF
         END DO
      END DO
      deallocate(E)
      RETURN
   END SUBROUTINE grsolv
END MODULE gralg
```

The second module is called bigupdate and it is here that the C matrix is updated with the result.

```
module bigupdate

private i,j,Istop,Istart,Jstart,Jstop
contains
 SUBROUTINE  bigup(C,A,E,blocksize,antalblock,blockkolumn,matrissize)
    implicit none

    double precision, dimension(:,:),allocatable :: C(:,:)
    double precision, dimension(:,:),allocatable :: A(:,:)
    double precision, dimension(:,:),allocatable :: E(:,:)

    integer, intent(in) :: blocksize,antalblock,blockkolumn
    integer :: i,j,Istop,Istart,Jstart,Jstop,matrissize,astart,astop

    !OMP GÅR HÄR
!$omp parallel do default(shared),private(i,j,istart,istop,jstart,jstop)
    do i = (blockkolumn-1),1,-1
       do j =1, antalblock,1
          Istart = (i-1) * blocksize +1
          istop  = i     * blocksize
          Jstart = (j-1) * blocksize +1
          Jstop  =  j    * blocksize
          call dgemm('n','n',blocksize,blocksize,blocksize,        - 1.0D+00, &
               A(istart:istop,((blockkolumn-1) * blocksize+1)) ,matrissize,   &
               E(:,jstart:jstop)                           ,blocksize,1.0D+00, &
               C(istart:istop,jstart)                      ,matrissize)
       end do
    end do
!$omp end parallel do

  end SUBROUTINE bigup
end module bigupdate
```

The third and last module is named updaterow and it is used to update the temporary E matrix.

```
MODULE updaterow


CONTAINS
    subroutine updater(E,C,B,blocksize,startkolumn,blockkolumn,blockrad,matrissize)
   IMPLICIT NONE
   DOUBLE PRECISION, dimension(:,:), allocatable :: E
   DOUBLE PRECISION, dimension(:,:), allocatable :: C
   DOUBLE PRECISION, dimension(:,:), allocatable :: B
   external omp_get_num_threads,omp_get_thread_num,omp_num_procs
   INTEGER :: blocksize,startkolumn,matrissize,k,kstart,blockrad,blockkolumn
   integer :: i,j,id,Nthread,idstart,idstop,Nproc,kstop,jstop
```

```
   integer :: omp_get_num_threads,omp_get_thread_num,omp_num_procs


   i      = (blockkolumn -1) * blocksize +1
   j      = (blockrad    -1) * blocksize +1
   jstop  = j+blocksize -1

   !$omp parallel do private(kstart,kstop) default(shared)
   DO k = 1,startkolumn,1
      kstart = (k-1) * blocksize +1
      kstop = kstart +  blocksize
      call dgemm('n','t' ,blocksize,blocksize,blocksize,1.0D+00,   &
           C(i,j)        ,matrissize,              &
           B(kstart      ,j),matrissize, 1.0D+00,  &
           E(: ,kstart)  ,blocksize                )
   END DO
   !$omp end parallel do

   return
 END subroutine updater

END MODULE updaterow
```

Appendix B

Algorithm 2

This is the code used for algorithm 2. A different style of coding was used for this algorithm. There is only one module and the parallelization is at the bottom layer of the algorithm and then OMP control directives are used to control the parallelization.

```
MODULE wave

CONTAINS

  SUBROUTINE wavesolve(A, B, C, matrisSize, blockSize, LDA, LDB, LDC)
    IMPLICIT NONE
    EXTERNAL recsydt

    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE,INTENT(in)  :: A
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE,INTENT(in)  :: B
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE  :: C
    DOUBLE PRECISION,DIMENSION(:,:),ALLOCATABLE  :: E
    INTEGER,INTENT(in) :: matrisSize,blocksize
    DOUBLE PRECISION :: scale
    INTEGER   :: antalblock
    INTEGER :: T = 3
    INTEGER :: M,N
    INTEGER :: LDA,LDB,LDC
    INTEGER :: info,diag
    INTEGER :: JS,Db,IS,IE,Da,Dc
    INTEGER :: k,Iindex,jend,Jindex,i,j,d,kstart,kstop
    INTEGER :: Jtemp,itemp,W
    ALLOCATE(E(matrissize,matrissize))
    antalblock  = CEILING(REAL(matrisSize)/REAL(blockSize))
    M           = blocksize
    N           = blocksize
    JS          = antalblock
    Db          = antalblock
    IS          = antalblock
    IE          = antalblock
```

```
Da          = antalblock
Dc          = Da + Db -1
scale       = 0
info        = 0
E(:,:) = 0.0D+00

!$omp parallel default(shared)
DO d = 1,Dc,1
   !$omp barrier
   !$omp master
   IF (d > Db) THEN
      IS = IS -1
   END IF
   j = JS
   !$omp end master
   !$omp barrier
   DO i = IS, IE ,-1
      !$omp master
      Jindex   = (j-1)  * blockSize  +1
      Iindex   = (i-1)  * blocksize  +1
      itemp    =  iindex + blocksize -1
      !$omp end master

      !$omp barrier

      !$omp sections
      !$omp section
      CALL recsydt(T,scale,M,N  , &
           A(Iindex,Iindex),LDA , &
           B(Jindex,Jindex),LDB , &
           C(Iindex,Jindex),LDC ,info)
      IF(info .NE. 0) THEN
         WRITE(*,*)"\t\t ---- FEL!! i recsy ----"
      END IF
      !$omp end sections

      !$omp master
      jend = j
      IF(i == 1) THEN
         jend = j - 1
      END IF
      !$omp end master
      !$omp barrier
      !$omp do private(kstart,kstop) ordered
      DO k = 1,jend,1
         kstart = (k-1)  * blocksize +1
         kstop  = kstart + blocksize -1
         if(j == Db) Then
            E(Iindex:itemp,kstart:kstop) = 0.0D+00
         end if
```

```
            CALL dgemm('N','T' ,blocksize,blocksize,blocksize,1.0D+00,   &
                 C(Iindex ,jindex)  ,matrissize,  &
                 B(kstart ,Jindex)  ,matrissize, 1.0D+00  ,&
                 E(Iindex ,kstart)  ,matrissize   )
         END DO
         !$omp end do

         !update Column j
         !$omp do private(kstart)
         DO W = 1,(i - 1),1
            kstart = (W-1)  * blocksize +1
            CALL dgemm('N','N',blocksize,blocksize,blocksize, -1.0D+00, &
                 A(kstart ,Iindex)  ,matrissize,   &
                 E(Iindex ,Jindex)  ,matrissize,1.0D+00, &
                 C(Kstart ,Jindex)  ,matrissize)
         END DO
         !$omp end do
         !$omp barrier
         !$omp master
         IF(j > 1) THEN
            Jtemp = Jindex - blocksize
            CALL dgemm('N','N',blocksize,blocksize,blocksize,-1.0D+00, &
                 A(Iindex ,Iindex)  ,matrissize, &
                 E(Iindex ,Jtemp)   ,matrissize, 1.0D+00, &
                 C(Iindex ,Jtemp)   ,matrissize )
         END IF
         j = j + 1
         !$omp end master
       END DO

       !$omp master
       JS = MAX((JS-1),1)
       IE = MAX((IE-1),1)
       !$omp end master

    END DO
    !$omp end parallel

  END SUBROUTINE wavesolve
END MODULE wave
```