UMEÅ UNIVERSITY
Institution of Computer Science

8th March 2005

Master's Thesis Project D, 20 credits

Implementing Parallel Recursive Blocked Algorithm for Solving The Standard Triangular Sylvester Matrix Equation

Author: Jonas Nyström
E-mail: c00jnm@cs.umu.se
Supervisor: Robert Granat

Abstract

The fastest known algorithm for solving the continuous-time Sylvester equation $AX - XB = C$ is the recursive blocked algorithm that can be found in the RECSY library, with results like 10-times shorter execution time compared to the solver in the LAPACK library. The approach taken in RECSY solves the problem in an interesting way. Many implementations of parallel solvers has been proposed for the Sylvester equation, but no one that solves the problem like RECSY using recursive blocking on a distributed memory machine. This Master's Thesis project is a first attempt in creating such a solver.

Contents

1 Introduction **9**
1.1 Background 9
1.2 Outline 9

2 The Task **11**

3 Fundamentals and "in-depth" study **13**
3.1 Sylvester Matrix Equations 13
3.2 GEMM-operations 14
3.3 Parallel Computers 14
3.3.1 Parallel Platforms 15
3.3.2 Design Principles of Parallel Algorithms 15
3.3.3 Basic Communication Operations 15
3.4 Analysis of parallel algorithms 16
3.4.1 Metrics for Parallel Execution 16
3.4.2 Communicational Costs 17
3.5 Fortran 90 17
3.6 MPI 18

4 RECSY **19**

5 The Parallel Algorithms **21**
5.1 4-group D_RECSYCT 21
5.2 3-group D_RECSYCT 23
5.3 2-group D_RECSYCT 23
5.4 Recursive GEMM-solvers 23
5.5 Load Balance 26

6 Implementation **27**
6.1 4-groups 27
6.1.1 D_RECSYCT 27
6.1.2 R_GEMM 28
6.2 3-groups 29
6.2.1 D_RECSYCT 29
6.2.2 R_GEMM 29
6.3 2-groups 29
6.3.1 D_RECSYCT 29
6.3.2 R_GEMM variant 1 30
6.3.3 R_GEMM variant 2 30

7 Algorithm Analysis **31**
7.1 R_GEMM 31
7.2 D_RECSYCT 32

8 Results **33**
8.1 Memory Consumption 33
8.2 Theoretical and Experimental Results 34

9 Adopted limitations and future work **41**

10 Conclusion and summary **43**
10.1 Acknowledgments 43

Bibliography **45**

List of Figures

3.1 2×2 block . . . 13
3.2 Standard Schur form example . . . 14

4.1 Case 1 - $1 \leq n \leq m/2$ . . . 20
4.2 Case 2 - $1 \leq m \leq n/2$ . . . 20
4.3 Case 3 - $n/2 < m < 2n$ . . . 20

5.1 4-group solver template . . . 21
5.2 4-group solver Algorithm . . . 22
5.3 Simplified 4-group solver description . . . 22
5.4 3-group solver template . . . 23
5.5 2-group solver template . . . 23
5.6 4-group R_GEMM algorithm. . . . 24
5.7 4-group R_GEMM template. . . . 25
5.8 3-group R_GEMM template. . . . 25
5.9 3-group R_GEMM template . . . 26
5.10 2-group R_GEMM template . . . 26

6.1 4-group implementation. . . . 28
6.2 GEMM matrix partitions . . . 28
6.3 GEMM variant 1 matrix partitions . . . 30
6.4 GEMM variant 2 matrix partitions . . . 30

8.1 Approximated memory usage. . . . 33
8.2 Actual memory usage. . . . 34
8.3 Theoretical speedup. . . . 35
8.4 Experimental speedups. . . . 37
8.5 Compared speedup with R_GEMM. . . . 37
8.6 Compared speedup without R_GEMM. . . . 38
8.7 Experimental vs. Theoretical R_GEMM results. . . . 39

List of Tables

4.1 The different types of Sylvester-type matrix equations that the RECSY library solves. CT and DT denote the continuous-time and discrete-time variants, respectively. 19

8.1 Results from D_RECSYCT with 2-groups, using RECSYCT as serial solver 36
8.2 Results from R_GEMM with 2-groups variant 1 39
8.3 Results from D_RECSYCT with 2-groups, using DTRSYL as serial solver 40

Chapter 1

Introduction

This Master Thesis Project is an attempt to create a parallel solver that solves the so-called triangular continuous-time Sylvester Equation

$$AX - XB = C, \tag{1.1}$$

where A of size $m \times m$ and B of size $n \times n$ are matrices in real Schur form and C of size $m \times n$ is a real matrix. This will be done by parallelizing a recursive blocked algorithm for solving the equation(1.1) on a distributed memory machine.

1.1 Background

A lot of research has been conducted finding the numerical solution to the equation, and a standard method can be found in an article by R.H. Bartels and G.W. Stewart [1].

This algorithm is implemented in explicitly blocked form in the LAPACK library [2]. The best known algorithm for solving the equation today is a recursive blocked variant by Jonsson-Kågström [7, 8], and can be used through the library RECSY [10, 9].

Some attempts has also been done to parallelize the algorithms in a distributed memory (DM) environment, see [16, 4]. These were based on ScaLAPACK (Scalable LAPACK) which includes a subset of the routines found in LAPACK [2] that has been redesigned to fit on distributed memory MIMD parallel computers [3].

Some research has also been conducted to combine the algorithm used in ScaLAPACK and use the RECSY-library as node solver, see [5].

1.2 Outline

After the introduction, a chapter (Chapter 2) will follow that specifies what the actual task is about. That is, *what* should be done, *how* and *with* what tools. Chapter 3 will give a brief introduction to some topics essential for this project to evolve. We will in this chapter give som information about matrix-equations, Parallel computers, how to analyze parallel algorithms and some

basics about Fortran90 and MPI. The parallel computers section will include some fundamentals associated with parallel computers, and since our target environment is the distributed memory machines, a bit more information is given on these types of parallel computers. In the next chapter, Chapter 4, the RECSY-library is reviewed. That is, how to use it, how it solves equation (1.1) by serial recursion.

This is followed by Chapter 5, where we explain how the parallelized algorithm(s) are intended to solve the problem. During the development several changes to the fundamental algorithm has been necessary. However, all stages in the development will be presented. Next, Chapter 6, presents the different implementations, how they work, how they are implemented and what the differences are between the implementations.

Chapter 7 contains the theoretical analysis of the parallel algorithm adopted. This is later to be compared with the experimental results. This is followed by a chapter, Chapter 8, that shows the result from the experimental test and compares them to the theoretical.

In chapter 9 limitations of the adopted implementations are stated. Future work and extensions to this thesis is also outlined. The final chapter, Chapter 10, presents some conclusions and final remarks about the project.

Chapter 2

The Task

The task is to parallelize an existing recursive blocked algorithm for solving the so-called triangular continuous-time Sylvester Equation (1.1) in a Distributed Memory (DM) environment. This already existing recursive blocked algorithm can be found in the HPC (High Performance Computing) software library RECSY [9, 10]. See Chapter 4 for more details about the RECSY library.

Hence, the task is to, in a DM environment, solve for the unknown matrix X in equation (1.1) using a recursive template which will be described closer in Chapter 5. The solver should be written in Fortran 90 using the MPI (Message Passing Interface)-library. See Chapter 3 for more details about Fortran 90 and MPI.

After the implementation an experimental and possibly a theoretical analysis of the result should be done. The ScaLAPACK-based hybrid mentioned above gives a speedup at about $\sqrt{p}$, where p is the number of processors used in the parallel execution. This degree of speedup is also the goal for this project. More information about what speedup is can be found in section 3.3.

The complete specification can be seen in [6].

Chapter 3

Fundamentals and ”in-depth” study

In order to understand the contents of this report, some basic information about matrices, parallel computers, how to analyze parallel algorithms and the Fortran 90 together with the MPI-library is given in this chapter.

3.1 Sylvester Matrix Equations

Equation (1.1) is a matrix equation, and we assume that the reader knows what a matrix is, and how basic operations on matrices are defined, as addition, subtraction and multiplication. As mentioned in (1.1) the matrix A is of size $m \times m$, the matrix B of size $n \times n$ and the matrix C of size $m \times n$. And if one matrix is to be multiplied with another, the inner dimensions of these matrices must be the same. Therefore, the matrix X must also have the size $m \times n$.

The classical method of solving Equation (1.1) is the Bartels-Stewart method [1], which includes three major steps. First matrices A and B are transformed to their Schur form, then the resulting equation is solved, and finally the result is transformed back to the original coordinate system [7].

A, B and C have real entries and after the first step they are in real Schur form (upper quasi-triangular). When a matrix is in standard Schur form all of its eigenvalues are on the diagonal, and if some eigenvalues are complex they appear in 2×2 blocks on the diagonal corresponding to conjugate pairs $a \pm bi$. Such a block is shown in Figure 3.1. Figure 3.2 shows an example of a random

$$\begin{pmatrix} a & b \\ -b & a \end{pmatrix}$$

Figure 3.1: 2×2 block

matrix A that is transformed into standard Schur form (*schur(A)*), and the eigenvalues of this new matrix are shown in *eig(A)*. When the matrices A and B are in standard Schur form the Equation (1.1) is a reduced triangular matrix equation [7] which is solved in the second step.

$$
\begin{aligned}
\mathbf{A} &= \begin{pmatrix} 0.4387 & 0.6435 & 0.7266 \\ 0.4983 & 0.3200 & 0.4120 \\ 0.2140 & 0.9601 & 0.7446 \end{pmatrix} \\
\mathbf{schur}(\mathbf{A}) &= \begin{pmatrix} 1.6235 & 0.1053 & -0.5853 \\ 0 & -0.0601 & 0.1589 \\ 0 & -0.1589 & -0.0601 \end{pmatrix} \\
\mathbf{eig}(\mathbf{A}) &= \begin{pmatrix} 1.6235 \\ -0.0601 + 0.1589i \\ -0.0601 - 0.1589i \end{pmatrix}
\end{aligned}
$$

Figure 3.2: Standard Schur form example

It is well known that Equation (1.1) has a unique solution if and only if the result from adding any eigenvalue from A with the corresponding one (same place on the diagonal) from B is greater then zero [1].

3.2 GEMM-operations

In order to solve this Sylvester equation in blocked form, matrix-matrix operations must be performed when updating a sub-matrix with a partial solution (see Chapter 4). Given three matrices (A, B and C) we must be able to do a GEMM-operation (GEneral Matrix Multiply and add),

$$C = \beta C + \alpha AB, \tag{3.1}$$

where α and β are scalars. Which adds the result from the matrix-matrix-multiply operation between A and B to the original matrix C. The sizes of the matrices A, B and C in Equation (3.1) can not have the same dimensions as in Equation (1.1). Matrix A must be of size $m \times k$, matrix B of size $k \times n$ and matrix C of size $m \times n$.

In the RECSY library there is a function called RECSY_GEMM that do this multiply-add operation (see [10] for more information), which is based on the DGEMM-routine in the BLAS library [13].

3.3 Parallel Computers

The need for a computer system to be faster then currently possible is continual [11]. Problems that need great computational speed often do huge repetitive calculations on large amounts of data. Numerical modeling and simulation of scientific and engineering problems are areas in which parallel computers are often used. One way to increase the computational speed is to use more processors (CPU:s) to solve the problem. This is the way a parallel computer is intended to work, and the parallel computers can be one computer with several processors or several computers connected with some kind of network. This network can in principle be a fast local area network (LAN), optimized to be used on such a parallel system, or the Internet. There are three main motivations for using parallel computers [11]. Maybe the most obvious is the

Computational Power argument and is that a parallel computer can execute more operations per second. The second argument is the *Memory/Disk Speed Argument* [11]. Even if the clock rates of the processors has increased about 40% per year over the last decade the access time to the DRAM memory has only increased with about 10%, so the memory is usually a huge bottleneck in computer systems. When using a parallel computer, some valuable advantages can yield better performance. They often have larger aggregate caches and higher aggregate bandwidth to the memory. The third argument for using parallelism is *data communication*, and is easily explained by mentioning the so called SETI (Search for Extra Terrestrial Intelligence)-project. This project uses various computers around the world to analyze electro-magnetic signals from outer space in the search for aliens using the Internet as a network [11].

3.3.1 Parallel Platforms

Often, the logical view of a computer consist of a memory connected to a processor via a data-path [11]. And all these present bottlenecks to the overall performance of the system. Fortunately several innovations has been presented that deals with these bottlenecks. For instance, using pipelines in the processors to do several instructions in parallel (like a conveyor belt), and using memory caches to reduce the memory latency.

There are two primary ways of parallel tasks to exchange data [11]. One is to access a shared data space like in the Shared-Address-Space platforms. The other is to send messages between the task, and is used on various Message-Passing platforms.

In order to connect the parallel computers several different types of network can be used. Some examples are bus-based networks, multistage networks, linear arrays and tree-based networks [11].

3.3.2 Design Principles of Parallel Algorithms

The two main things to do when writing a parallel algorithm is to divide the computation into smaller parts and to assign them to different processors that can execute them concurrently [11]. Several techniques exist for doing this decomposition. The more general ways are *recursive*- and *data decomposition*. The first uses the divide-and-conquer technique for solving the problem. The data-decomposition divides the data into several chunks and creates tasks accordingly to these. These are then solved by different processors.

3.3.3 Basic Communication Operations

In most parallel algorithms, processes need to exchange data with other processes. This communication can significantly slow down the parallel algorithm by introducing interaction delays [11]. Some often used communication methods beside ordinary *point-to-point send* and *point-to-point receive* are *one-to-all broadcast*, *all-to-one reduction*, *all-to-all broadcast and reduction*, *scatter* and *gather*. The *point-to-point send* and *point-to-point receive* pretty much explains them self, but broadcast is when you send to all and reduction is when all send

to you. The *scatter* operation is used to spread the message in some way, so that different process gets different parts. *Gather* is the opposite to *scatter*. The only one of these that is used in this thesis is *one-to-all broadcast*.

3.4 Analysis of parallel algorithms

So, if we have a parallel algorithm on a parallel computer we have a parallel system. And when we have a parallel algorithm or program, it is often good to compare the experimental results with a theoretical analysis. In order to do this we need some measurements of the theoretical result that we can compare to the experimental. There are several sources of overhead in parallel programs [11], such as the interprocess interaction, idling and excess computation. The first overhead is often the biggest in nontrivial parallel systems, idling can occur when processors have to wait for some other to reach a certain point, and excess computation denotes that the parallel algorithm solves the problem in a more expensive way then the serial solver.

3.4.1 Metrics for Parallel Execution

The most obvious way of measuring is to measure the **execution time**. This is denoted T_s for the serial run and T_p for the parallel.

The **total overhead** equation, Equation (3.2), is defined as the difference between the serial time and the parallel execution time multiplied with the number of processors [11].

$$T_o = pT_p - T_s \tag{3.2}$$

Speedup is a metric that captures the relative benefit of solving a problem in parallel compared to sequential [11]. It is defined as

$$S_p = \frac{T_s}{T_p}, \tag{3.3}$$

where T_s is the sequential execution time and T_p is the parallel execution time for solving the same problem on p processors. The speedup value can theoretically never be more then p, but if it becomes greater, the speedup is said to be super-linear. This happens mostly for extremely parallel applications by the introduction of p cache memories in the parallel execution. If we assume that there will be some parts that are executed on one processor, this execution time can be expressed as fT_s, where f is the fraction of the program that will be serial [12]. In an ideal situation, the other part that can be parallelized is $(1-f)T_s$. Thus, Equation (3.3) can be written as

$$S_p = \frac{T_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}, \tag{3.4}$$

and this equation is known as *Amdahl's law*. This means that even if we have infinitely many processors, the maximum speedup is limited to $\frac{1}{f}$.

Efficiency is a measure of the fraction of time for which a processing element is usefully employed [11]. For a parallel program, efficiency is defined as

$$E_p = \frac{S_p}{p}. \tag{3.5}$$

Theoretically, E_p can not exceed 1, since S_p can not exceed p.

The **scalability** is a measure of a parallel systems capacity to increase the speedup in proportion to the number of processing elements so that the efficiency is constant. **Efficiency** has a tendency to drop when increasing the number of processors, and increase when the size of the problem grows. If it is possible to modify the number of processors and the problem size simultaneously so that the efficiency is held constant the system is said to be *scalable*. The efficiency can also be rewritten as

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}, \tag{3.6}$$

and using the equation for parallel overhead it can be rewritten as

$$E_p = \frac{1}{1 + \frac{T_o}{T_s}}, \tag{3.7}$$

which means that the efficiency is directly related to the parallel overhead.

3.4.2 Communicational Costs

As mentioned before, the communication is a big source of overhead in nontrivial parallel algorithms. There are different ways for a system to transfer a message through the network, and the communication cost differs between them. Two communication schemas are *store-forward routing* and *packet routing*, see [11] for more details. We use a simplified cost model for sending a message in a message passing system and this will be used in the analysis later. This simplified model says that the total communication cost for sending m words from node A to node B takes

$$t_{comm} = t_s + t_w m \tag{3.8}$$

seconds, where t_s is the start-up time for the message (also known as *node latency*), t_w is the time for transmitting one word over one link in the network (which is directly proportional to the inverse of the bandwidth of the network).

3.5 Fortran 90

The programming language *FORTRAN* is the dominating language for technical scientific calculations and was originally developed by IBM in 1954 [14]. In 1958 the language became *FORTRAN II* and eventually even other manufactures released compilers for their architectures. In 1962 IBM released *FORTRAN IV* which was the foundation of the ANSI X3.9-1966 standard, and in 1978 an altered standard which got the nickname *Fortran 77* emerged. In 1991 *Fortran 90* was standardized and in 1997 it was followed by *Fortran 95*. The newest versions of the language includes news for vector- and matrix operations which performs well on vector-machines and parallel computers. Some other new features in the new versions are that they include several new methods of specifying the data precision, lets the user dynamically allocate memory and create user-defined data structures. But the most important new feature in the *Fortran* language for this thesis, is the ability to do recursion.

3.6 MPI

MPI (Message Passing Interface) [15] is one of many libraries for the message-passing programming paradigm. The programs on these types of systems are often SPMD (single program multiple data) [11], that is, every processor runs its own copy of the same program, but on different data.

The most fundamental operations that exist in a message-passing library are the *send* and *receive* operations. These can be blocking or non-blocking. When blocking, it does not return until the message is received, while when using non-blocking variant computation can be done while sending or receiving.

The MPI library contains over 125 routines, but the basic set is a group of six routines. Namely *MPI_Init*, *MPI_Finalize*, *MPI_Comm_size*, *MPI_Comm_rank*, *MPI_Send* and *MPI_Recv*. The first two is for starting and closing the MPI environment. The *comm*-operations are used to get a processors rank in a group (communicator), and see how many processors there are in the communicator. So a communicator can be seen as a group of processors, and when MPI_Init has been runned on all processors all processors are part of a group (communicator) called MPI_COMM_WORLD.

When using the various routines in the library, they often take a communicator as a parameter plus some other parameters. This means that for one processor to communicate with another, they need to be in the same group. MPI implements the basic communication operations previously mentioned, such as the *one-to-all broadcast*.

For more details see documentation of MPI [15]. Another important operation, especially when we want to divide the processors into groups is the *MPI_Comm_split*-operation. It takes an old communicator (group) and divides it into new ones. Every processor in the communicator must call it, and the processors will get a new communicator depending on some color and key given to the function.

Chapter 4

RECSY

RECSY [9] is a library for solving Sylvester-type matrix equations in a recursive blocked manner. The small-sized leaf problems in the recursion tree are solved on new high-performance kernels [7, 8]. This recursive approach leads, in contrast to standard blocking techniques, to automatic variable blocking that has the potential of matching the memory of todays HPC (High Performance Computer)-systems [10]. RECSY consists of a set of Fortran 90 routines that are equipped with Fortran 77 interfaces and LAPACK/SLICOT wrappers. These routines recursively solves eight different Sylvester-type matrix equations, and these standard variants are shown in Table 4.1.

Name	Matrix Equation
Standard Sylvester (CT)	$AX - XB = C$
Standard Lyapunov (CT)	$AX + XA^T = C$
Standard Sylvester (DT)	$AXB^T - X = C$
Standard Lyapunov (DT)	$AXA^T - X = C$
Generalized Coupled Sylvester	$(AX - YB, DX - YE) = (C, F)$
Generalized Sylvester	$AXB^T - CXD^T = E$
Generalized Lyapunov (CT)	$AXE^T + EXA^T = C$
Generalized Lyapunov (DT)	$AXA^T - EXE^T = C$

Table 4.1: The different types of Sylvester-type matrix equations that the RECSY library solves. CT and DT denote the continuous-time and discrete-time variants, respectively.

Depending on the sizes of m and n, three alternatives for doing a *recursive splitting* are considered [7, 8]. In Case 1 ($1 \leq n \leq m/2$), A is split by rows and columns, and C by rows only. Figure 4.1 shows the splitting and the resulting equations. We first solve for X_1, then update and solve the first equation. Similarly, in Case 2 ($1 \leq m \leq n/2$), B is split by rows and columns, and C by columns only (split and equations are shown in Figure 4.2).

$$\begin{pmatrix} A_{11} & A_{12} \\ & A_{22} \end{pmatrix}\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} - \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} B = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}.$$

$$\begin{aligned} A_{11}X_1 - X_1B &= C_1 - A_{12}X_2, \\ A_{22}X_2 - X_2B &= C_2 \end{aligned}$$

Figure 4.1: Case 1 - $1 \leq n \leq m/2$

In this case we first solve for X_1, then update and solve the second equation. Finally, in Case 3 ($n/2 < m < 2n$) both rows and columns of the matrices A,

$$A\begin{pmatrix} X_1 & X_2 \end{pmatrix} - \begin{pmatrix} X_1 & X_2 \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ & B_{22} \end{pmatrix} = \begin{pmatrix} C_1 & C_2 \end{pmatrix}.$$

$$\begin{aligned} AX_1 - X_1B_{11} &= C_1, \\ AX_2 - X_2B_{22} &= C_2 + X_1B_{12}. \end{aligned}$$

Figure 4.2: Case 2 - $1 \leq m \leq n/2$

B and C are split. Figure 4.3 shows the split and the resulting four equations. First, X_{21} is solved for in the third equation. After updating C_{11} and C_{22} with

$$\begin{pmatrix} A_{11} & A_{12} \\ & A_{22} \end{pmatrix}\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} - \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

$$\begin{aligned} A_{11}X_{11} - X_{11}B_{11} &= C_{11} - A_{12}X_{21}, \\ A_{11}X_{12} - X_{12}B_{22} &= C_{12} - A_{12}X_{22} + X_{11}B_{12}, \\ A_{22}X_{21} - X_{21}B_{11} &= C_{21}, \\ A_{22}X_{22} - X_{22}B_{22} &= C_{22} + X_{21}B_{12}. \end{aligned}$$

Figure 4.3: Case 3 - $n/2 < m < 2n$

respect to X_{21}, one can solve for X_{11} and X_{22}. Both updates and the triangular Sylvester solves are independent operations and can be executed concurrently. Finally, one updates C_{12} with respect to X_{11} and X_{22}, and solves for X_{12}. Observe that all subsystems are solved using the recursive blocked algorithm. If a splitting point ($m/2$ or $n/2$) appears at a 2×2 diagonal block (Figure 3.1) of A or B, the matrices are split just below this diagonal block, so that these blocks are not splitted. The recursion stops when the sub-matrices are sufficiently small. The recursive approach is natural to SMP-parallelize, which is implemented in RECSY using OpenMP. When using these SMP-parallelized versions of the routines, performance gain is remarkable, including 10-fold speedups. The signatures of these routines differs from the standard ones, so that "_P" extends the name and an extra parameter PROCS tells how many processors will be used. The software and documentation concerning RECSY is available for download [10]. For details see the papers by Jonsson and Kågström [9, 7, 8].

Chapter 5

The Parallel Algorithms

In this chapter, the initial proposed algorithm for solving Equation (1.1) will first be presented, followed by a second idea for the recursive division, followed by a third and final algorithm. Our recursive solver goes by the name D_RECSYCT and stands for "Distributed" RECSYCT. The logical processor topology is assumed to be a 2-dimensional processor mesh, this is the case in all following algorithms.

5.1 4-group D_RECSYCT

The algorithm proposed by the specification [6] is based on how the RECSY library performs the recursive division. The proposed template for case $n/2 < m < 2n$ divides the matrices like RECSY (see Chapter 4) and the processor grid in a similar way. That is, the processor grid is divided in four sub-grids on which the subsystems are solved. As mentioned in the previous chapter

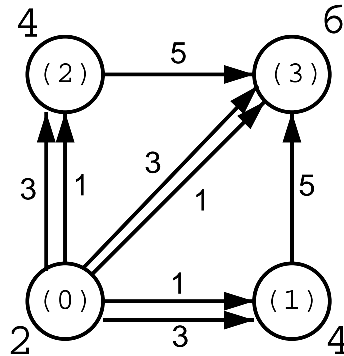


Figure 5.1: 4-group solver template

when splitting the equation in both rows and columns (Figure 4.3) the equation can be rewritten into four triangular equations (Figure 4.3). The proposed algorithm was intended to work as shown in Figure 5.2 and the steps of the algorithm are shown (Figure 5.1) on a 2×2 mesh where the nodes (0, 1, 2 and 3) can be single processors or a groups of processors (this holds for all figures

and algorithms throughout the chapter). In this chapter, we use numbers to describe the operations (computation and communication). These numbers are in Figures 5.1, 5.4 and 5.5 based on the algorithm shown in Figure 5.2. Figures 5.7, 5.8, 5.9 and 5.10 are on the other hand based on the algorithm shown in Figure 5.6.

- All data in 0 - if this is a single processor, the serial solver (RECSYCT) is used and we are done.
- **1** Start non-blocking communication
  - **1** Send A_{22}, B_{22}, C_{22} and B_{11} to (1).
  - **1** Send A_{11}, B_{11}, C_{11} and A_{12} to (2).
  - **1** Send $A_{11}, B_{22}, C_{12}, A_{12}$ and B_{12} to (3)
- **2** Solve X_{21} recursively on (0).
- **3** Broadcast solution (X_{21}) to all ((1),(2) and (3)).
- **4** Update C_{22} and solve X_{22} on (1) concurrently updating C_{11} and solving X_{11} on (2).
- **5** Send X_{11} and X_{22} to (3).
- **6** Update C_{12} and solve X_{12} at (3).

Figure 5.2: 4-group solver Algorithm

The initial thought was to divide the processor grid conformally to the way RECSY divides the matrices in the all three cases (including $1 \leq n \leq m/2$ and $1 \leq m \leq n/2$). But if such a division is made, the resulting equations has no concurrent parts that can be solved in parallel. Therefore, we decided to always divide the matrices according to case 3 (both row-wise and column-wise), and take advantage of that two of the subsystems can be solved concurrently. Therefore, the processor grid is always divided into four new groups if possible (the number of processors at a depth in the recursion is greater then or equal to four), thereof the name 4-group.

A simplified description of how the partial solutions are solved over the processor grid is shown below (Figure 5.3), which can help understanding the transition from 4-groups to 3-groups presented in the next section:

- Processor (0) solves and send the solution to (1) and (2).
- Processors (1) and (2) concurrently updates, solves its subsystem and sends the result to (3).
- Processor (3) updates with result from (1) and (2) and solves the final equation.

Figure 5.3: Simplified 4-group solver description

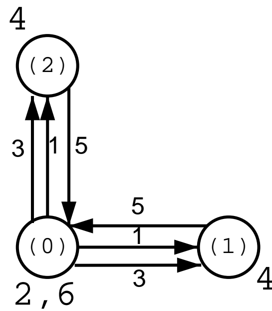


Figure 5.4: 3-group solver template

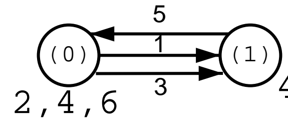


Figure 5.5: 2-group solver template

5.2 3-group D_RECSYCT

Since only case-3 division of the matrices are performed, the number of resulting equations in every recursion step is always four. If we take a look at the description in Figure 5.3 we see that when processor (3) starts updating and solving the final solution the three other processors becomes idle.

To reduce the number of idle processors, we send the results from (1) and (2) back to (0). This template is shown in figure 5.4. In this case the parallel algorithm uses the existing serial RECSYCT when the number of processors in a group is less then three.

5.3 2-group D_RECSYCT

Rather late in the development phase a new idea emerged which instead uses 2-groups. If we return to the simplified description from Section 5.1, we see that processor (0) is finished when the first solution is sent to (1) and (2). What if we let processor (0) do processor (1):s work as well?

When processor (0) has solved the first part, it can update and solve (1):s problem directly after sending the first solution to processor (2). Compared to 4-group division, processor (0) now does all the work that (0), (1) and (3) used to do. And this without losing any concurrency, and doing less communication and allocation. This will be further described in Chapter 6. Figure 5.5 shows the template for the 2-group solver. And this means that RECSYCT is used when the recursion over the processor-grid is down to two or less computers in the group.

5.4 Recursive GEMM-solvers

When the initial algorithm (4-groups) did its updates it used the RECSY_GEMM-function from the RECSY-library. This means that this operation is performed by a single processor, and becomes a large bottleneck. This according to Amdahl's law, that states that the level of speedup is limited by the serial fraction

of a program (Equation 3.4). A small example may illustrate this fact:

Say that the processor (2) in Figure 6.1 has received the first solution from the lower left group (that used processors (0), (4), (8) and (12) to solve the first equation), and should now update the matrices received earlier. Then this update will be done by processor (2) only and not the entire group (including processors (2), (6), (10) and (14)).

It would be better if all the processors in that group participated in that update. This leads to our recursive parallel GEMM-function that we call R_GEMM. The first variant intended to work together with the 4-group solver divides the matrix equation (Equation 3.1) into four sub-equations (which will be shown in section 6.1.2). These are then distributed to the three other groups and that solves and returns the result to the sender. Figure 5.7 shows how the communication is done between the groups and the algorithm is shown in Figure 5.6.

- If single processor and processor (0) - solve and return result.
- If processor (0).
  - **{1, 1.1}** Start non-blocking communication to (1), (2) and (3) in group.
  - Update own part matrix recursively on (0):s new group.
  - **{2, 2.2}** Receive part result from (1), (2) and (3).
  - Merge results and return.
- If processor (1), (2) or (3).
  - **{1, 1.1}** Receive matrices from (0) in group.
  - Update own part matrix recursively on own new group.
  - **{2, 2.2}** Send back the result to (0).

Figure 5.6: 4-group R_GEMM algorithm.

In other words, when processor (0) in figure 5.7 shall update its part it uses the group with the processors (0), (4), (8) and (12) to do this. This part is then (recursively) divided into four parts and sent (step 1.1) to (4), (8) and (12) in Figure 5.7. When (4), (8) and (12) has solved their parts they send the result back to (0) and the lower leftmost group is ready. The same recursive procedure is done by all four groups, and just like the case with the actual solver, the R_GEMM reaches the base-case when there are less then four processors in a group in which a part result is to be calculated.

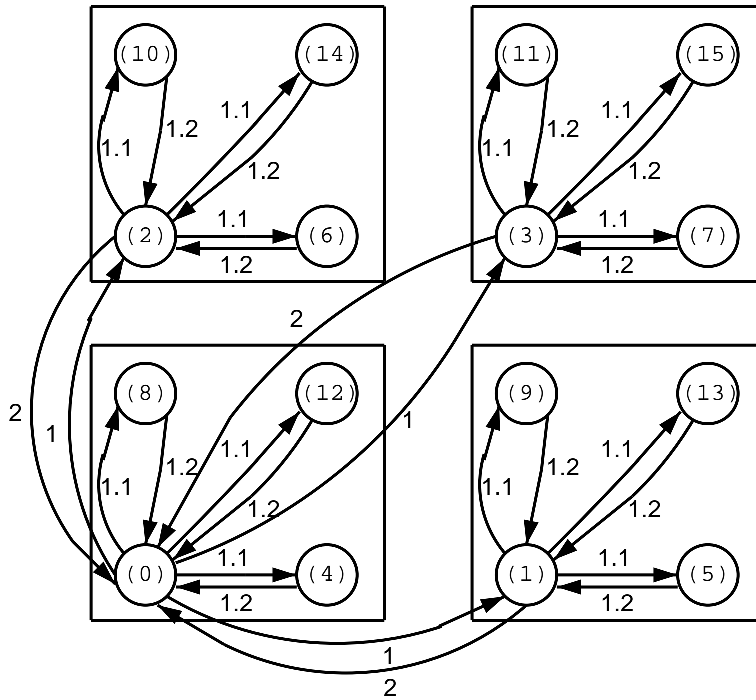


Figure 5.7: 4-group R_GEMM template.

When the solver went from 4-groups to 3-groups, even the R_GEMM had to be altered in a similar way. The result is that instead of using 4-groups in the R_GEMM processor (0) does processor (3):s work, and we get 3-groups in the R_GEMM routine as well (see Figure 5.8). This means that when a GEMM operation is done on a processor grid the (0) group/processor does twice the work compared to (1) and (2). When the 2-group solver was implemented,

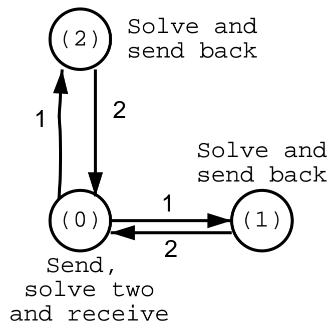


Figure 5.8: 3-group R_GEMM template.

R_GEMM was also altered to fit the same processor-grid division. This was initially done by letting processor/group (0) update both (0) and (1):s parts while (1) updates (2) and (3):s parts (4-group id:s), see Figure 5.9. So the

matrices are still divided into four parts and computed separately. But a better idea (at least in theory) emerged that divides the matrices to be updated into two parts instead of four, and one part is solved on (0) and the other on (1), see Figure 5.10. The different implementations of R_GEMM uses the serial solver

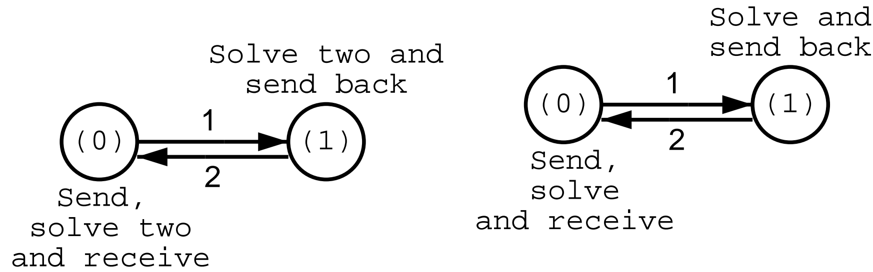


Figure 5.9: 3-group R_GEMM template

Figure 5.10: 2-group R_GEMM template

RECSY_GEMM when the group has less then four, three or two processors, respectively. In Chapter 6 we will show how the matrices are divided and how the R_GEMM routine performs the operation in the different implementations.

5.5 Load Balance

In the 4-group algorithm we have good load balance, all four groups solves one forth of the problem. This, both in the solver and in R_GEMM. But there is a trade-off between load balance and efficiency. Since a lot of idle time is present, the efficiency could be increased, but not without affecting the load balance. With the 3-group algorithm the load balance has decreased since processor (0) solves (3):s part as well. The same unbalance is found in the R_GEMM since even here processor (0) does twice the amount of work as the others. This, on the other hand, leads to fewer idle processors. And in the final 2-group solver we have even worse load balance, since here processor (0) solves three parts and processor (1) only one. However, both variants of R_GEMM are well balanced. Even if the load balance for the solver is even worse, the number of idle processors are less.

Chapter 6

Implementation

As mentioned in Chapter 5 several changes has been done to the basic algorithms during the implementation phase. But the matrices in the solver is always divided into four sub-matrices (RECSY case-3). In this chapter, we will explain in a more detailed manner how the different implementations works. But first we give a general description of how the different processor groups are created in the different implementations. This is made by a communicator split operation mentioned in section 3.3. In order to decide in which group a processor should go, the modulus-operation[1] is performed on the rank of the processor with four, three or two (depending on implementation). The processors with the same result will belong to the same group after the split.

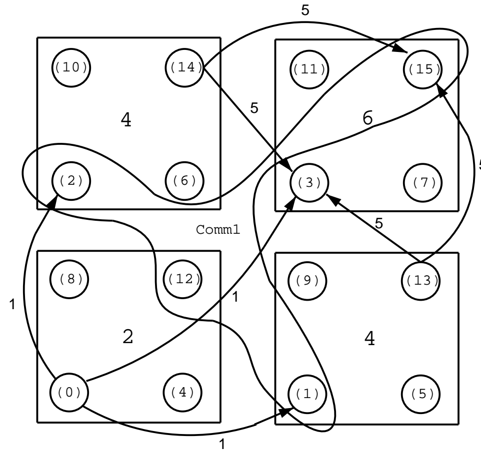
6.1 4-groups

The proposed way of solving the problem was also the first one to be implemented. We will briefly describe how the solver is implemented followed by the R_GEMM variant which was based on 4-groups.

6.1.1 D_RECSYCT

Nothing has been said about where the data is at start. But processor (0) is the one that initiates the matrices and start spreading them. But where will the data be when finished? Typically we don't want the data to be spread around the grid when the execution finish. It is better that when all partial solutions has been calculated they are merged at one processor. If we look at the algorithm in Figure 5.2 one might notice that we send the first partial solution X_{21} to (3) without this using it to update and solve its own part. This is one step in trying to get all the data into one processor, namely (3) (this is for a 2×2 mesh). This is implemented by a broadcast to those three processors. If we now look at Figure 6.1, we see that in a larger grid, this broadcast is sent from the up-most right node in a group to (1), (2) and the up-most right node (15) in group (3). This is because none of the processors in group (3) uses this subsolution and (15) should have the final data when the solver is finished. The same holds for

[1] The modulus operation takes two integers and returns the rest after the division if both integers have the same sign (will always be positive in our case). Ex. mod(10,4)=2. 10/4=2 and the rest is 2.



The processors are labeled between ():s and groups are represented with squares (observe that the numbers in the middle of each square are not the group nr. but the steps of the solve algorithm). The communication steps in the figure are numbered as in Figure 5.1, apart from that Comm1 (broadcast communicator) replaces the third step.

Figure 6.1: 4-group implementation.

.

step 5 in the same algorithm. If the group to send to from group (1) and (2) contains more then three processors, the up-most right processor should have the subsolutions from group (1) and (2). These must also be sent to the down-most left processor (3) in group (3) because it needs them to update its own matrices before solving the last part recursively on group (3). This requires that processors must check whether they are down-most left or up-most right or not so that they know how to communicate. For instance, if a processor is a up-most right node like (15) it must receive from groups (1) and (2) before joining its own group to solve for the final subsolution.

6.1.2 R_GEMM

In order to remove the bottleneck of a call to a serial GEMM routine, the first implementation design for the 4-group solver works as follows. The $C = \beta C + \alpha A * B$ is partitioned like showed in figure(6.2).

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \beta \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} + \alpha \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Figure 6.2: GEMM matrix partitions

And the result of the AB-multiplication can be written as

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

so that

$C_{11} = \beta C_{11} + \alpha A_{11}B_{11} + \alpha A_{12}B_{21}$ is computed by (2),
$C_{12} = \beta C_{12} + \alpha A_{11}B_{12} + \alpha A_{12}B_{22}$ by (3),
$C_{21} = \beta C_{21} + \alpha A_{21}B_{11} + \alpha A_{22}B_{21}$ by (0) and
$C_{22} = \beta C_{22} + \alpha A_{21}B_{12} + \alpha A_{22}B_{22}$ by (1).

In order for processors (1), (2) and (3) to compute their sub-equations, processor (0) will send α and five matrices to each of them. But, since no scaling of the C matrix is done, β is always 1 and will therefore never be sent in the implementations.

6.2 3-groups

The processor grid is divided in three groups in ever step of the recursion (until we reach the base case), which was described in Figure 5.4.

6.2.1 D_RECSYCT

This way of solving the problem has several advantages that by far outweighs its disadvantages. We have one less communication, since processor (1) already has the matrices that was previously sent to (3). Processor (0) solves the first sub problem, and sends them to (1) and (2). They update and solve their subproblems concurrently and sends the solution back to (0). Processor (0) now updates (two GEMM-operations) and solves for the final subsolution, and the solution is in (0). The two last updates can utilize the whole group since (1) and (2) are idle. Since the processors always send back the result to (0) in a group there is no need to do some extra communication to get the matrices elsewhere (like step 5 in the 4-group solver). The only drawback is that the load balance is not optimal, since processor (0) solves two parts while (1) and (2) only solves one. But a faster result is sometimes better then good load balance, which holds in this case. Experiments also confirms this fact.

6.2.2 R_GEMM

The second version of the R_GEMM implemented to be used together with the 3-group solver divides the matrices like the previous version (Figure 6.2). But here we lack the (3) so processor/group (0) does (3):s part of the job.

6.3 2-groups

The final implementation uses as said before 2-groups to reduce the number of idle processors to a minimum.

6.3.1 D_RECSYCT

This implementation does not differ very much from the 3-group variant. The initial communication becomes "smaller", since we only send to (1) since (0) will do both (2):s and (3):s parts of the job (in 4-group notation). After sending the matrices that is needed by (1), (0) solves the first subsystem, sends it to (1), updates and solves the part that (2) did earlier. When (1) has sent back the result of its problem, processor (0) does two updates as for the 3-group variant.

Even here the last two updates are done on the whole group ((0) and (1)) using R_GEMM.

6.3.2 R_GEMM variant 1

This implementation also divides the matrices like the 4-group variant, but since the number of subsolutions to be found is even and the processor-group is divided by two, group (0) can do half the work and (1) the other.

6.3.3 R_GEMM variant 2

An idea that emerged which in theory should be better suited for the 2-group R_GEMM is to split the matrices in half. The splitting can be done in two ways, and are done depending on the dimensions of the C-matrix. If the C-matrix has the dimensions $m \times n$ and $m \geq n$, A and C are splitted in half row-wise, while B is not decomposed, see Figure 6.3.

$$\underset{C}{\left(\frac{0}{1}\right)} = \underset{C}{\left(\frac{0}{1}\right)} + \underset{A}{\left(\frac{0}{1}\right)} \underset{B}{\left(\ 0/1\ \right)}$$

Figure 6.3: GEMM variant 1 matrix partitions

Otherwise ($m \leq n$), B and C are splitted in half column-wise while A is intact, see Figure 6.4.

$$\underset{C}{\left(\ 0 \ \middle|\ 1\ \right)} = \underset{C}{\left(\ 0 \ \middle|\ 1\ \right)} + \underset{A}{\left(\ 0/1\ \right)} \underset{B}{\left(\ 0 \ \middle|\ 1\ \right)}$$

Figure 6.4: GEMM variant 2 matrix partitions

The indexation (0 and 1) used in the matrices shows what parts of the matrices the processors/groups uses. Processor/group (0) sends the parts that has index 1 to processor/group (1) in this variant of R_GEMM.

Chapter 7

Algorithm Analysis

Since the distributed parallel solver uses the parallel distributed R_GEMM, the theoretical parallel execution time must consist of R_GEMM parts. The serial solver ($T_0(n)$) takes $2n^3t_a$ seconds to solve a problem with size n, where t_a is the time for one flop.

7.1 R_GEMM

The experimental difference in performance of the solver is small for the different implementations of R_GEMM (see Chapter 8). Therefore, we have chosen to do the analysis of R_GEMM with the first 2-group variant. This because the division of the matrices is always done into four new sub-matrices. We define k to be $log_2(p)$, where p is the number of processors. For example, if k equals 2 we have 4 processors.

The serial time ($R_0(n)$) to compute a GEMM-operation on $n \times n$ matrices (both A, B and C have the size $n \times n$) is $2n^3t_a$.

Processor (0) will start by sending the submatrices to processor (1). If we take a look at the resulting four subsystems in section 6.1.2 and recall that in the first 2-group variant the work is divided in half, we see that in order to send two subsystems we need to send 10 matrices to processor (1). Processor (1) sends two result matrices back to processor (0), and when processor (0) is finished we have the result. Processor (0) solves its part (two equations, which consist of two GEMM-operations each) while processor (1) solves the other half. But since (0) sends the matrices non-blocking and starts solving its part directly, (0) will finish before (1) and must therefore wait until (1) has sent the results back before the operation is complete. For higher k-values it is suitable to express the computational parts with recursion equations, since the computations will be performed on the $k-1$ level in every step.

In summary, we have the following:
k = 1
$R_{commsend} = t_s + t_w * 10 * (\frac{n}{2})^2$
$R_{commrecv} = t_s + t_w * 2 * (\frac{n}{2})^2$
$R_{commtot} = ... = 2 * t_s + 2 * t_w * n^2$
$R_{comp} = 4 * 2 * (\frac{n}{2})^3 * t_a = n^3 * t_a$

$R_1(n) = n^3 * t_a + 2 * ts + 3 * t_w * n^2$

k = 2
$R_{commsend} = t_s + t_w * 10 * (\frac{n}{2})^2$
$R_{commrecv} = t_s + t_w * 2 * (\frac{n}{2})^2$
$R_{commtot} = ... = 2 * t_s + 2 * t_w * n^2$
$R_{comp} = 4 * R_0(\frac{n}{2})$
$R_2(n) = 4 * R_0(\frac{n}{2}) + 2 * t_s + 3 * t_w * n^2$

k = k
$R_k(n) = 4 * R_{k-1}(\frac{n}{2}) + 2 * t_s + 3 * t_w * n^2$

7.2 D_RECSYCT

The first communication to (1) is assumed to be hidden under computations. Processor (0) computes its first part (T_{comp}), then sends the solution to (1) ($T_{commsend}$), updates and solves its second part ($T_{comp(0)}$) a bit before (1) starts its update and solve. So in the analysis, processor (0) has to wait the time it takes to send the first matrices plus the time to send the solution back ($T_{commsend}$ and $T_{commrecv}$). After this, (0) can do its last two updates ($T_{2\times GEMM}$) and solve the last part ($T_{complast}$). Like in the R_GEMM analysis recursion equations is suitable to use since the solver uses level $k - 1$ to solve the subproblems.

k = 1
$T_{comp} = T_0(\frac{n}{2})$
$T_{commsend} = t_s + t_w * (\frac{n}{2})^2$
$T_{comp(0)} = R_0(\frac{n}{2}) + T_0(\frac{n}{2})$
$T_{commrecv} = t_s + t_w * (\frac{n}{2})^2$
$T_{2\times GEMM} = 2 * R_1(\frac{n}{2})$
$T_{complast} = T_0(\frac{n}{2})$
$T_1(n) = 3 * T_0(\frac{n}{2}) + R_0(\frac{n}{2})) + t_s + t_w * (\frac{n}{2})^2 + 2 * R_1(\frac{n}{2})$

k = 2
$T_{comp} = T_1(\frac{n}{2})$
$T_{commsend} = t_s + t_w * (\frac{n}{2})^2$
$T_{comp(0)} = R_1(\frac{n}{2}) + T_1(\frac{n}{2})$
$T_{commrecv} = t_s + t_w * (\frac{n}{2})^2$
$T_{2\times GEMM} = 2 * R_2(\frac{n}{2})$
$T_{complast} = T_1(\frac{n}{2})$
$T_2(n) = 3 * T_1(\frac{n}{2}) + R_1(\frac{n}{2})) + t_s + t_w * (\frac{n}{2})^2 + 2 * R_2(\frac{n}{2})$

k = k
$T_k(n) = 3 * T_{k-1}(\frac{n}{2}) + R_{k-1}(\frac{n}{2}) + 2 * R_k(\frac{n}{2}) + 2 * t_s + 3 * t_w * n^2$

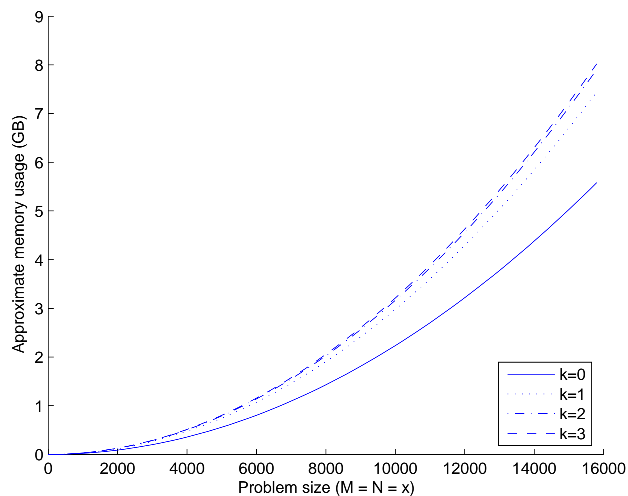
Chapter 8

Results

This is indeed a hard problem to solve by recursion in a distributed memory environment, therefore one should not have to high expectations on the prototype implementations.

8.1 Memory Consumption

In order to see how large problems that could be tested, a graph over the approximated memory usage was created. And since all the initial data is created in (0) and all the other has a little amount of this sent to them, the memory usage at (0) sets the limit of the problem size. Figure 8.1 shows this graph, and the different lines are for different values of k (recall that the number of processors utilized is 2^k).



Memory usage for 2^k processors (that is 1, 2, 4, 8 ,...).

Figure 8.1: Approximated memory usage.

The largest problem solved was 14000 on 32 processors and if we look in the graph we see that the estimated memory consumption should be somewhere around 6.2 GB, but it becomes a bit larger then that in practice. When using four processors ($k = 2$) the consumption of memory will approximately be around 5.8 GB. But the graph only considers some of the solver's used memory, for instance the memory used by R_GEMM is not included. Therefore, the graph is a little optimistic compared to the actual result shown in Figure 8.2.

The figure (Figure 8.2) shows an experimental graph over the memory usage

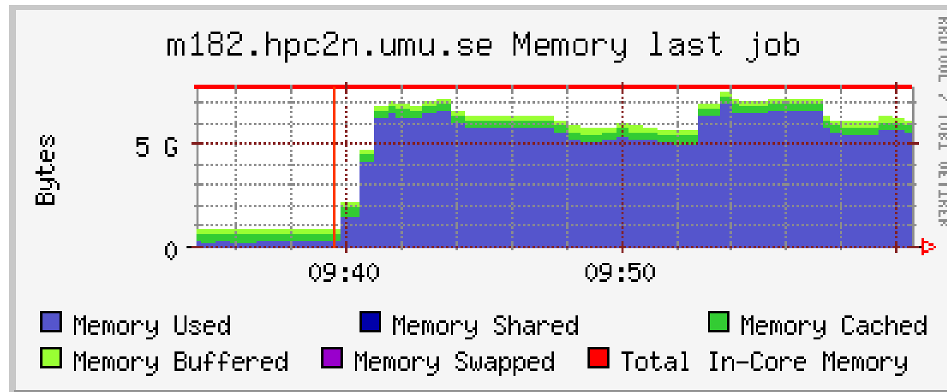


Figure 8.2: Actual memory usage.

on processor (0) when running a job with size 14000 on four processors. We got this experimental system information during runtime from HPC2N:s portal [17].

8.2 Theoretical and Experimental Results

Some of the values are not to be taken for 100% correct, since only one test run has been made on some of them. But some hints about how the algorithm solves the equation can be seen. All test are performed on problems with $M = N$, this means that matrices A, B and C have the same sizes. Before the actual results from the test runs are presented, a graph of the theoretical result, based on the analysis from Chapter 7 is shown in Figure 8.3.

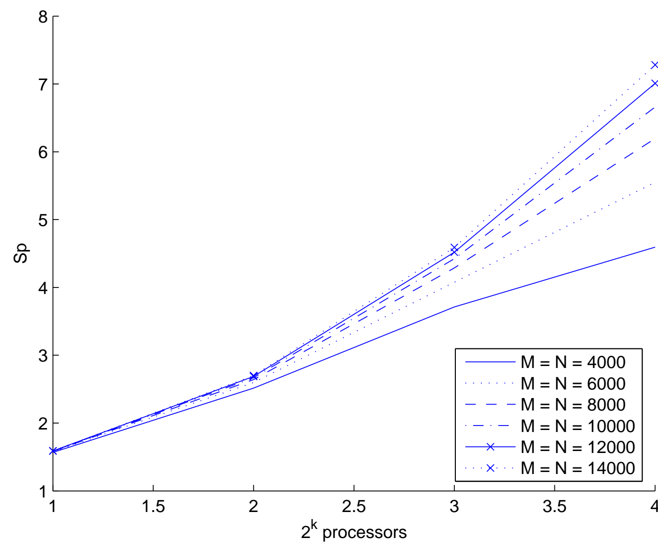


Figure 8.3: Theoretical speedup.

Since the two first implementations (4-group and 3-group solvers) has no (theoretical) possibility of being better then the 2-group implementation, no data from the early test runs with these are available and therefore they will not be presented. Table 8.1 shows the result from the tests.

$M = N$	p	v1 time (sec.)	S_p	E_p	v2 time (sec.)	S_p	E_p
2000	1	4.68	1.00	1.00	4.68	1.00	1.00
	2	5.76	0.81	0.41	5.65	0.83	0.41
	4	5.55	0.84	0.21	5.2	0.90	0.45
	8	6.2	0.76	0.09	5.31	0.88	0.11
4000	1	34.7	1.00	1.00	34.7	1.00	1.00
	2	39.4	0.88	0.44	38.9	0.89	0.45
	4	33.45	1.03	0.26	32.1	1.08	0.27
	8	32.5	1.06	0.13	30.3	1.15	0.14
6000	1	115.3	1.00	1.00	115.3	1.00	1.00
	2	125.7	0.92	0.46	125.3	0.92	0.46
	4	107.6	1.07	0.27	106.6	1.08	0.27
	8	91.3	1.26	0.16	91.6	1.26	0.16
8000	1	266.9	1.00	1.00	266.9	1.00	1.00
	2	288.0	0.93	0.46	287.0	0.93	0.47
	4	234.7	1.14	0.28	238.6	1.12	0.28
	8	185.6	1.44	0.18	199.2	1.34	0.17
10000	1	520.0	1.00	1.00	520.0	1.00	1.00
	2	553.0	0.94	0.47	550.0	0.95	0.47
	4	444.0	1.17	0.29	455.0	1.14	0.29
	8	351.3	1.48	0.19	375.0	1.39	0.17
12000	1	890	1.0	1.00	N/A	1.00	1.00
	2	938	0.94	0.47	N/A	N/A	N/A
	4	739	1.20	0.30	N/A	N/A	N/A
	8	561	1.59	0.20	N/A	N/A	N/A
14000	1	N/A	1.00	1.00	1410	1.00	1.00
	2	N/A	N/A	N/A	1469	0.96	0.48
	4	N/A	N/A	N/A	1185.3	1.19	0.30
	8	N/A	N/A	N/A	960	1.47	0.18
	16	N/A	N/A	N/A	771.5	1.83	0.11
	32	N/A	N/A	N/A	648	2.17	0.07

Table 8.1: Results from D_RECSYCT with 2-groups, using RECSYCT as serial solver

The meaning of the columns are as follows:

- $M = N$ - The problem sizes, M and N have the same value.
- p - The number of processors utilized.
- v1 time - Time taken by variant 1 (in seconds).
- v2 time - Time taken by variant 2 (in seconds).
- S_p - Speedup.
- E_p - Efficiency.

Figure 8.4 shows the speedup plotted for the different problem sizes.

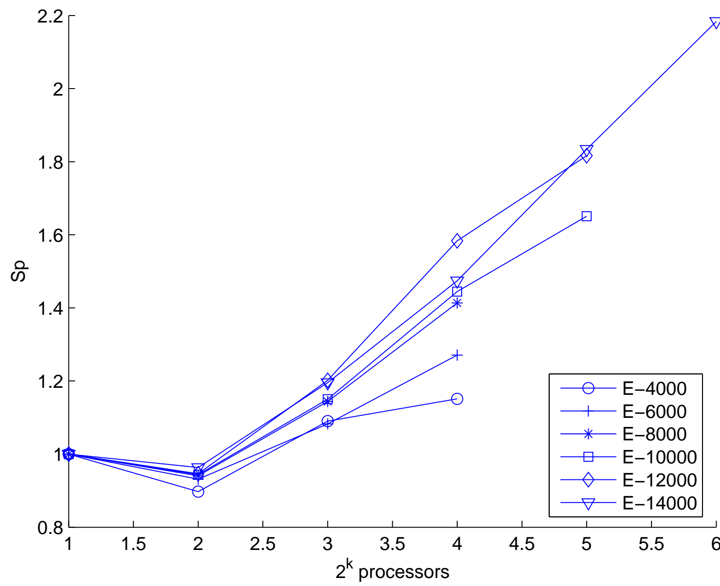


Figure 8.4: Experimental speedups.

In the figures E stands for experimental and T for theoretical. E.g. $E - 4000$ stands for the experimental speedup with problem size 4000. The speedup values are not high, which is confirmed when plotting them together with the theoretical values as in Figure 8.5.

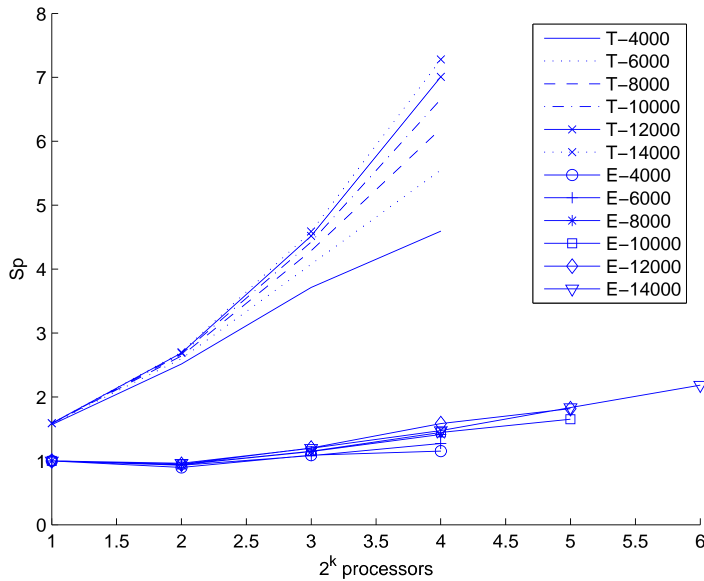


Figure 8.5: Compared speedup with R_GEMM.

Let us try to show how the algorithm works compared to if the theoretical analysis were done with no parallel GEMM-operations. This is shown in Figure 8.6.

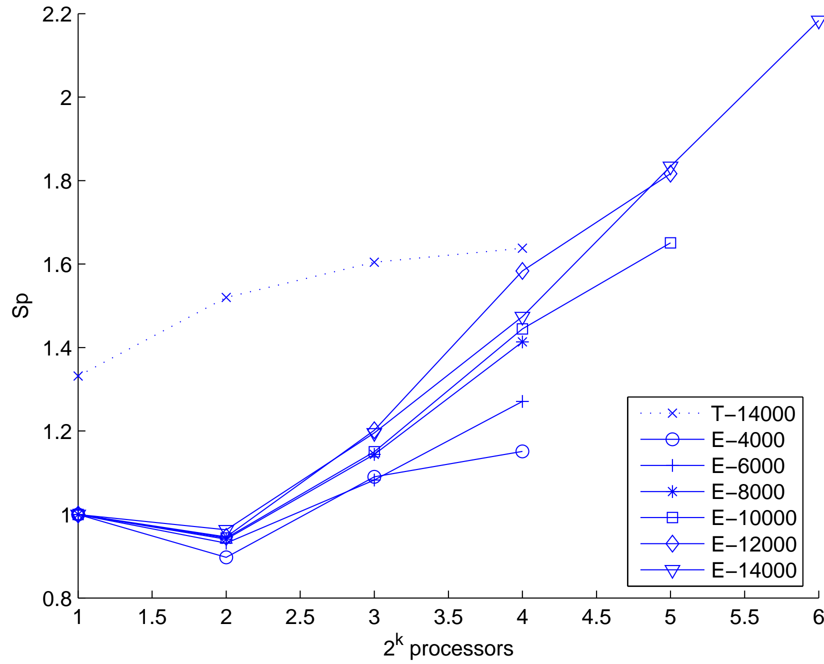


Figure 8.6: Compared speedup without R_GEMM.

Here we see that if no parallel GEMM is used in the theoretical analysis the results from the experimental testing would overcome the theoretical speedup for a large enough problem on enough many processors. Looks like this could happen for a problem size of 8000 when the number of processors is about 64. It is interesting to see that the speedup with different problem sizes only varies a little. In practice it may be R_GEMM that sets a limit on the speedup. Therefore, R_GEMM was tested separately and the results are shown in Table 8.2.

$M = N$	p	time (sec.)	S_p	E_p
1000	1	0.52	1.00	1.00
	2	0.80	0.65	0.32
	4	0.84	0.62	0.15
2000	1	4.06	1.00	1.00
	2	5.20	0.78	0.39
	4	4.52	0.90	0.22
	8	4.67	0.87	0.11
4000	1	32.35	1.00	1.00
	2	37.10	0.87	0.44
	4	27.80	1.16	0.29
	8	24.00	1.35	0.17
6000	1	109.2	1.00	1.00
	2	120.0	0.91	0.46
	4	83.5	1.30	0.33
	8	67.0	1.63	0.20
	16	63.0	1.73	0.11

Table 8.2: Results from R_GEMM with 2-groups variant 1

In Figure 8.7 the values in Table 8.2 are plotted together with the results from the analysis in Chapter 7. Here we can clearly see that R_GEMM do not perform

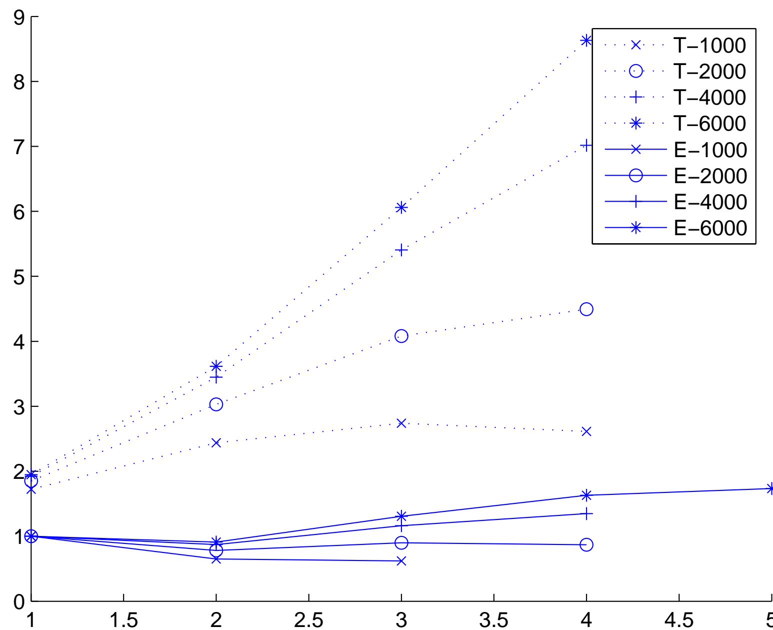


Figure 8.7: Experimental vs. Theoretical R_GEMM results.

as well as we had hoped for. It actually performs worse then the serial solver for smaller sizes. So in order to take advantage of R_GEMM, really large problems

must be solved.

So, why does the experimental results differ so much from the theoretical? First, the theoretical analysis for the solver and for R_GEMM are to optimistic. Work including copying matrices, building one-dimensional arrays of the matrices to be sent, rebuilding the matrices from the one-dimensional arrays when receiving are work that is not included in the analysis. Second, the implemented code may not work optimally, therefore, further code analysis has to be done in order to optimize the code if possible. In general, the data to be solved on distributed memory machines is already distributed among the processors. This approach can be one way of significantly improve the results.

The last test is a test where the ∼10 times slower node-solver DTRSYL from the LAPACK library is used instead of the fast RECSYCT from RECSY. Table 8.3 shows the result from this test.

$M = N$	P	dtrsyl time (sec.)	S_p	E_p
64	1	0.009	1.00	1.00
	2	0.016	0.56	0.28
	4	0.043	0.21	0.05
128	1	0.009	1.00	1.00
	2	0.007	1.29	0.64
	4	0.028	0.32	0.08
256	1	0.10	1.00	1.00
	2	0.063	1.59	0.79
	4	0.047	2.13	0.53
	8	0.11	0.91	0.11
512	1	3.26	1.00	1.00
	2	0.63	5.18	2.6
	4	0.36	9.10	2.26
	8	0.27	12.07	1.51
1024	1	32.94	1.00	1.00
	2	11.39	2.89	1.45
	4	2.88	11.43	2.86
	8	1.69	19.49	2.43

Table 8.3: Results from D_RECSYCT with 2-groups, using DTRSYL as serial solver

Here the resulting speedup is super-linear, which probably has to do with that the solver does not use the caches as effectively as the RECSYCT solver. And when the problems are divided into smaller sub-problems, they can be solved faster, and therefore we get such high level of speedup. These results are comparable (and slightly better) with the results from the ScaLAPACK-based implementations that also uses DTRSYL as node solver.

Chapter 9

Adopted limitations and future work

The limitations of the solver designed and implemented in this thesis are pretty obvious. The result is not as good as one could hope for. The goal was $\sqrt{p}$ in speedup, but the result is not even close to this. One rule of thumb in parallel computing is that it is much easier to get good speedup from a bad algorithm (e.g. DTRSYL) than from a good one (like RECSYCT).

Another limitation is that when matrices are splitted, the sizes are not checked. Thus, if some matrix is to small to split the solver will probably crash.

There is an INFO-result that tells how the serial solver finished. This information is returned by every processor that calls the serial RECSYCT, so when the parallel solver is finished this should be gathered to processor (0) where the result matrices are. This can be done by a global reduction (all-to-one), with the maximum-operation (the maximum value of INFO is received in processor (0)).

In the R_GEMM implementations, only α is sent. In order to use the β-scalar in R_GEMM, this has to be implemented.

Future work can be to solve these limitations. The easiest limitations to solve are to gather the INFO-result from the processors to processor (0) and check the sizes of the matrices before splitting. The first limitation, that is the actual result, is a bit harder to fix. Since only two of the four parts solved can be parallelized in every recursion step, the speedup can not be huge. In order to get larger speedups, maybe a different algorithm is necessary. This is a matter of future investigation. So is further analysis of the code.

Chapter 10

Conclusion and summary

Even if the result has not been exactly as intended, it has been a very learn-full process. Since there has been several modifications of the algorithms during the development phase, many difficulties when implementing parallel programs has been understood. And the whole Fortran bit was also new, and it is always good to have seen an extra programming language. Since the D-level course in parallel computing has not been read, some analytical concepts were new as well, but now it is a possibility that this course will be attended.

10.1 Acknowledgments

I would really like to thank my supervisor Robert Granat for providing this Master's Thesis Project, and for supporting and encouraging me throughout the entire project. I would also like to thank HPC2N for support and facilitation of the parallel computers.

Bibliography

[1] R.H. Bartels and G.W. Stewart Algorithm 432: Solution of the Equation $AX + XB = C$, *Comm. ACM*, 15(9):820–826, 1972.

[2] E. Anderson, Z. Bai, C. Bischof. J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov and D. Sorensen. *LAPACK User's Guide*. Third Edition. SIAM Publications, 1999.

[3] S. Blackford, J. Choi, A. Clearly, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM Publications, Philadelphia, 1997.

[4] R. Granat, B. Kågström, P. Poromaa. Parallel ScaLAPACK-style Algorithms for Solving Continuous-Time Sylvester Matrix Equations, In H. Kosch et.al. (editors), *Euro-Par 2003 Parallel Processing*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 2790, pages 800–809, 2003.

[5] R. Granat, I. Jonsson and B. Kågström. Combining Explicit and Recursive Blocking for Solving Triangular Sylvester-Type Matrix Equations in Distributed Memory Platforms. In M. Danelutto, D. Laforenza, M. Vanneschi (EDS.): Euro-Par 2004, *Lecture Notes in Computer Science*, Vol. 3149, pp. 742-750. 2004

[6] R. Granat. Master Thesis Proposal (D-Level, 20 credits): Implementing Parallel Recursive Blocked Algorithms for Solving The Standard Triangular Sylvester Matrix Equation, 9th September 2004.

[7] I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part I: One-Sided and Coupled Sylvester-Type Equations, *ACM Trans. Math. Software*, Vol. 28, No. 4, pp 393–415, 2002.

[8] I. Jonsson and B. Kågström. Recursive Blocked Algorithms for Solving Triangular Matrix Equations—Part II: Two-Sided and Generalized Sylvester and Lyapunov Equations, *ACM Trans. Math. Software*, Vol. 28, No. 4, pp 416–435, 2002.

[9] I. Jonsson and B. Kågström. RECSY - A High Performance Library for Solving Sylvester-Type Matrix Equations, In H. Kosch et.al. (editors), *Euro-Par 2003 Parallel Processing*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 2790, pages 810–819, 2003.

[10] I. Jonsson and B. Kågström. RECSY — A High Performance Library for Sylvester-Type Matrix Equations. `www.cs.umu.se/research/parallel/recsy/`, 2003.

[11] A. Grama, A. Gupta, G. Karypis and V. Kumar Introduction to Parallel Computing Second Edition Addison-Wesley, 2003.

[12] B. Wilkinson and M. Allen Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers Prentice Hall, 1999.

[13] BLAS – Basic Linear Algebra Subprograms. `http://www.netlib.org/blas/`.

[14] B. Einarsson Lärobok i Fortran 90/95 Version 2.8, Maj 2004 Matematiska institutionen, Linköpings universitet, 2004

[15] MPI – Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[16] P. Poromaa. Parallel Algorithms for Triangular Sylvester Equations: Design, Scheduling and Scalability Issues. In Kågström et al. (eds), *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*, Lecture Notes in Computer Science, Vol. 1541, pp 438–446, Springer-Verlag, 1998.

[17] HPC2N – High Performance Computing Center North. `http://www.hpc2n.umu.se`.