Increasing solver performance for circuit simulation problems

Vollebregt, A.J.

Award date:
2007

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

INCREASING SOLVER PERFORMANCE FOR CIRCUIT SIMULATION PROBLEMS

by
A.J. Vollebregt

Supervisors:
Prof. Dr. W.H.A. Schilders (TU/e)
Dr. ir. S.H.M.J. Houben (Magma)

Eindhoven, December 2005

Abstract

Since computer chips have become smaller and smaller over the years, their interior is getting more complex. These chips suffer both dynamic and static dissipation, which result in power leakage and signal delay. The magnitude of these effects (and thus the performance of a chip) can be determined by calculating the voltage drop, for which a linear system has to be solved. A common used technique to solve increasingly growing SPD-matrices with origin in circuit simulation is the Preconditioned Conjugate Gradient method with an incomplete Cholesky decomposition as preconditioner. A consequence of Cholesky decomposition of sparse matrices is the occurrence of fill-in. Fill-in elements are additional nonzero matrix elements, which increase the amount of memory. Reordering of the matrix is a widely used method to reduce the appearance of fill-in. In addition to some methods to improve the iteration speed of the Preconditioned Conjugate Gradient method, this thesis will also discuss several local and global ordering methods such as the Minimum Degree ordering and the Nested Dissection ordering. These orderings are designed to reduce the number of fill-in elements. We present the software package MADAND that reduces the number of fill-in elements and the number of iteration steps of the PCG-method by approximately 50 percent in comparison with the current ordering MINOLD, and can partition the matrix so it can be solved in parallel, which may lead to a speedup factor of 4.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In the demanding age we live in, people do not only want new features for their electronic devices, they also expect them to become smaller and more durable. For example, a mobile phone can have features like a camera, internet, and games, while it has the size of a thumb and has a battery that lasts in standby mode for a week without charging. So new technologies result not only in smaller designs of computer chips (also called integrated circuits or IC's), but the designs get also more complex. Therefore it becomes an increasingly bigger challenge to design fast IC's using low power.

Power Dissipation

We have to zoom to approximately 100nm to understand the difficulties of chip design. Computer chips are build up out of millions of transistors. Assume a design has complementary metal oxide semiconductor (CMOS) devices, the most common used semiconductor. Figure 1.1 shows a simple inverter design with two transistors, T_1 and T_2, connected by a gate. T_1 is connected to the power rail and T_2 to the ground rail. The transistors have a certain switching threshold V_t.

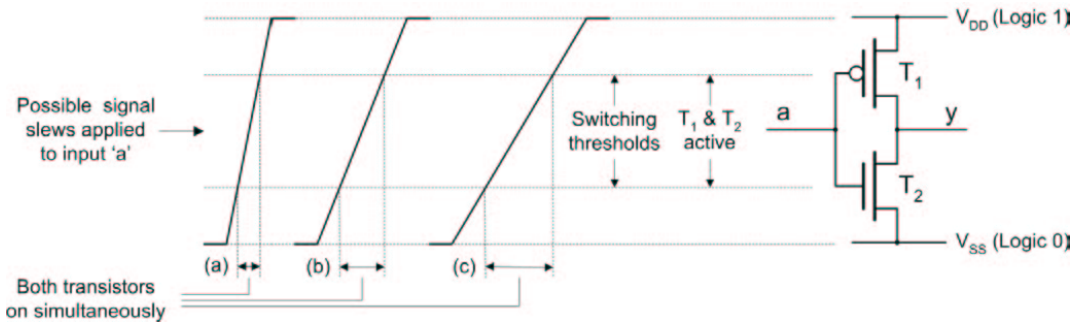

Figure 1.1: Simple power design.

If a input signal reached the inverter, the gate switches from one state to another, and both transistors are turned on for a fraction of a second. This results into a short circuit between the power net V_{DD} and the ground net V_{SS}. The time this circuit is active depends on the thresholds and the slope of the input signal. The size of the transistors is very important for the effects that will occur. If the transistors are too large, the gate is overpowered, which means that it consumes a great deal of power (case a). On the other hand, if the transistors are too small, the circuit will be on for too long, therefore the inverter will consume a considerable amount of power (case c). This is called *Dynamic Power Dissipation*. It can be calculated by the following equation

$$P_{dyn} \approx \alpha f \cdot C \cdot V^2, \tag{1.1}$$

with P_{dyn} is the dynamic power, f is the operating frequency, α is the switching activity factor of the gate, C is the amount of capacitance being switched, and V is the supply voltage.

In addition to Dynamic Power Dissipation, a design may suffer *Static Power Dissipation*. This happens when the gates are not active. There is a certain leakage current ($I_{Leakage}$) going through the transistors.

$$I_{Leakage} \approx e^{-qV_t/kT}, \tag{1.2}$$

with q is charge of an electron, k is the Boltzman constant, and T is the temperature. Also there is a certain delay (t_{delay}), the switching time, described by

$$t_{delay} \approx V_{DD} \cdot (V_{DD} - V_t)^{-\alpha}. \tag{1.3}$$

The loss of power and especially the delay must be minimized to get an optimal working chip.

Voltage Drop

If a chip is designed the V_t is known for every transistor. To acknowledge the dissipation effects the other voltages V_{DD} and V_{SS} must be calculated. These voltages suffer a phenomena that is called *voltage drop*. Voltage drops are reduction of the voltage if a current goes through a resistance. Because both the dynamic power and the delay depend on the supply voltage, it is important to calculate the voltage drop at each segment of the design.

Current Situation

One of the products the company of Magma Design Automation develops is software to simulate IC-designs.

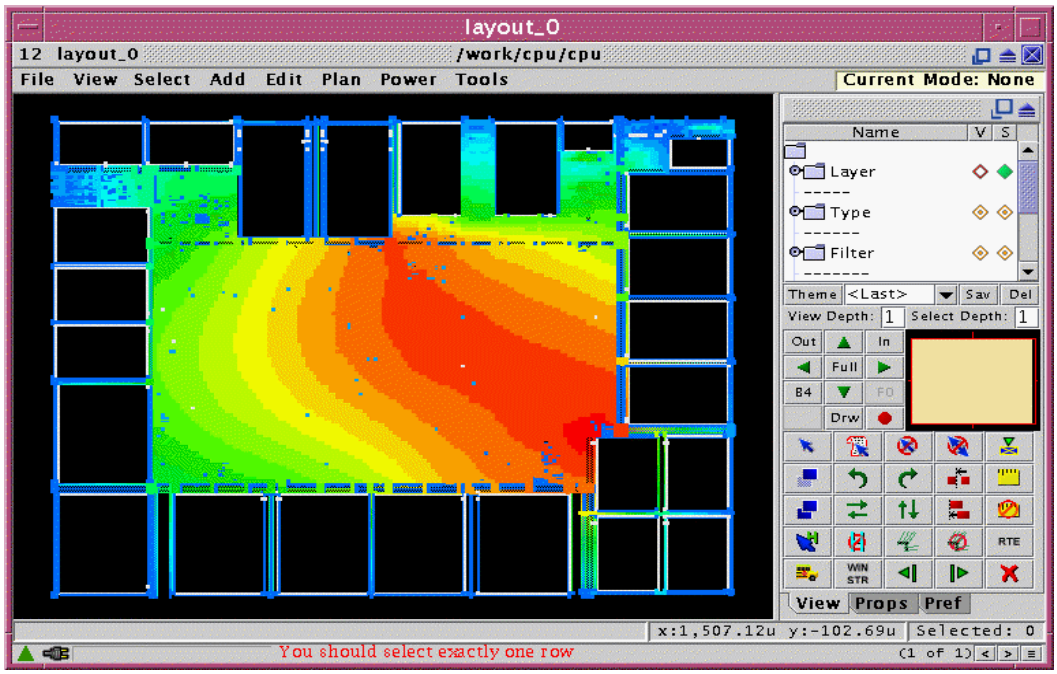

Figure 1.2: Magma's voltage drop map.

Figure 1.2 shows the voltage drop map of a chip design, and points out the problem ares. To calculate the voltage drop some very basic - but enormously large - mathematical systems have to be solved, which

can be done using linear algebra. These systems are solved using the preconditioned conjugate gradient method with an incomplete Cholesky factorization using a threshold as preconditioner.

Problem

The problem with this preconditioner is the occurrence of fill-in, new nonzero matrix elements which cost extra memory. If the number of fill-in elements is to large, the memory will get swamped, which is lethal for the performance of the solver. The fill-in can be reduced by tightening the threshold, but, as a consequence, the convergence speed of the solver drops.

Goal

The goal of the thesis is:

- Increase the performance of the solver by either reduction of the fill-in or reduction of the number of iteration steps
- Find a parallel solving method so multiple processors can be used to calculate the solution.

In this thesis we study several methods that may either reduce the occurrence of fill-in, reduce the number of iteration steps of the solving method or parallelize the problem. In chapter 2 the basics of circuit simulation is described, and the mathematical system is modelled. In chapter 3 we discuss several solving methods and preconditioner matrices, and the origin of the problem is formulated.

In chapters 4, 5, and 6 we describe several matrix ordering methods that may reduce the number of fill-in elements and may partition the matrix. Chapter 5 contains several local ordering methods such as the Minimum Degree ordering and Minimum Fill ordering, and several improvements. In chapter 6 we describe some global ordering methods such as the Nested Dissection ordering and Multisection ordering that partition a matrix, and in chapter 7 is described how the system is solved for a partitioned matrix. Finally, the results of all methods are given in chapter 8 .

I would like to finish this introduction with some acknowledgements. First of all, I would like to thank my supervisors prof. dr. Wil Schilders and dr. ir. Stephan Houben for valuable suggestions and guidance throughout the project. I am also very grateful for mr. Houbens help in mastering the necessary programming skills. In addition, I would like to thank dr. Jos Maubach and dr. Rudi Pendavingh for participating in my graduation committee. My gratitude also goes to the company of Magma Netherlands for giving me the opportunity to fulfill my graduation project and giving me a great experience. Finally, I would like to thank all my friends who gave me advice, especially regarding the C++ language.

Sander Vollebregt, December 2005

Chapter 2

Electronic circuits

2.1 Physical laws

Electrical systems consist of certain electrical devices, such as transistors, resistors, and capacitors, all connected by a network. If a mathematical model of these devices and their interaction is made, computers can be used to predict their behaviour. This procedure is called *circuit simulation*. A simple electrical circuit has two main variables: the *current* $\mathbf{i}$, and the *voltage difference* $\mathbf{v}$. The circuit consists of *nodes*, connected by *branches*, which represent the electrical devices. However, in our model we only use resistors, so all branches only have a certain resistivity ($\mathbf{r}$).

There are two types of equations that can be used to describe an electrical circuit: *branch equations* and *topological equations*. A branch equation (BE) depends on the type of branch used (resistor), and describes a relation between the circuit variables (in this case the current and the voltage difference). A topological equation (TE) depends on the topology of the circuit, which means that it only depends on the manner the nodes are connected. For our simple circuit there are three important physical laws that describe the circuit:

1. Ohm's Law (BE)

$$v_j = i_j r_j, \tag{2.1}$$

   for every branch j.

2. Kirchhoff's Current Law (TE)

$$\sum_{a_k \in \mathbf{a}} i_{a_k} = 0, \tag{2.2}$$

   for every cutset $\mathbf{a} = \{a_1, \ldots, a_n\}$.

3. Kirchhoff's Voltage Law (TE)

$$\sum_{b_k \in \mathbf{b}} v_{b_k} = 0, \tag{2.3}$$

   for every loop $\mathbf{b} = \{b_1, \ldots, b_n\}$.

A *cutset* is a minimal set of branches that divides the circuit into two separate parts if one would remove them. A *loop* is a path that starts and finishes in the same node. The circuit is directed, which means that the sign of the variable depends on the direction. If branch k connects node p with node q, then i_k and v_k are positive in the $p - q$ direction and negative in the $q - p$ direction.

Ohm's law (2.1) can also be written in terms of conductances

$$i_j = \gamma_j v_j, \tag{2.4}$$

with the conductance $\gamma_j = \frac{1}{r_j}$. This relation can be written in matrix form

$$\Gamma \mathbf{v} = \mathbf{i}, \tag{2.5}$$

with $\Gamma = diag(\gamma_1, \ldots, \gamma_b), \Gamma \in \mathbb{R}^{b \times b}$, $\mathbf{i} = \{i_1, \ldots, i_b\}$, $\mathbf{i} \in \mathbb{R}^b$ and $\mathbf{v} = \{v_1, \ldots, v_b\}, \mathbf{v} \in \mathbb{R}^b$.

2.2 Nodal Analysis

A classic method to construct electrical circuit equations is *Nodal Analysis*. For this method, the electrical circuit is represented as a simple, directed graph. The information found in the graph is used in the *nodal incidence matrix* $A \in \mathbb{R}^{n \times b}$, with n the number of nodes and b the number of branches in the circuit. A is defined by

$$A(i,j) := \begin{cases} 1 & \text{if branch } j \text{ finishes in node } i; \\ -1 & \text{if branch } j \text{ starts in node } i; \\ 0 & \text{if branch } j \text{ has no connection with node } i. \end{cases} \tag{2.6}$$

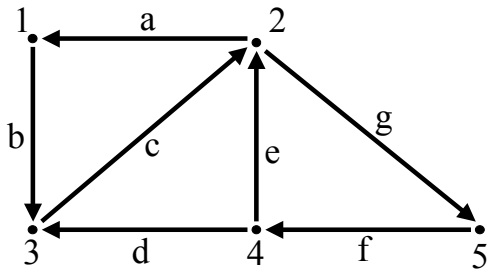

Figure 2.1: A graph of a simple electrical circuit.

An example of an electrical circuit graph can be found in Figure 2.1. This graph has the following incidence matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \end{array}.$$

Using Nodal Analysis as a framework we can make a connection between the physical laws from section 2.1. The Kirchhoff's Current Law (KCL) says that the sum of the current over every cutset $\mathbf{a}$ should be equal to zero. For every node the set of adjacent edges is a cutset, which means that for every node the incoming current should equal the outgoing current. If, for example, we look at node 2 in Figure 2.1, we find using the KCL

$$-i_a + i_c + i_e - i_g = 0.$$

This equation is also found by calculating $A(2,j)\mathbf{i} = 0$. We can generalize this for every node, which results into

$$A\mathbf{i} = \mathbf{0}. \tag{2.7}$$

This system is only valid if there are no current sources. More generally, we have

$$A\mathbf{i} = \mathbf{s}, \tag{2.8}$$

with $\mathbf{s}$ the vector of the source terms.

It is also possible to find a similar relation in matrix form for the KVL. The Kirchhoff's Voltage Law says that the sum of the voltage difference over a loop should be equal to zero. This means that if we take two arbitrary points the voltage difference between those two points is always the same, independent of the path we choose to get from one point to the other. Take two arbitrary nodes q_1 and q_2, and two disjoint paths from q_1 to q_2, p_1 and p_2. Then the KVL says that $v_{p_1} + (-v_{p_2}) = 0$, so $v_{p_1} = v_{p_2}$. Now give q_1 a certain potential w_1. Then the potential of q_2 , w_2, can be calculated, since every path to q_2 has the same voltage difference. And because q_1 and q_2 are arbitrary, all nodes have a potential, which we will refer to as w_i. The connection between the node potential and the voltage difference is

$$v_j = w_{i+} - w_{i-},$$

with v_j the voltage difference over branch j, w_{i+} the potential of the node in which branch j finishes and w_{i-} the potential of the node in which branch j starts. Looking at Figure 2.1 gives for branch a

$$v_a = w_1 - w_2.$$

This can be generalized for the entire circuit, which results into the following relation

$$A^T\mathbf{w} = \mathbf{v}, \tag{2.9}$$

with $\mathbf{w} = \{w_1, \ldots, w_b\}, \mathbf{w} \in \mathbb{R}^b$. It is important to realize that $\mathbf{w}$ is a potential, i.e. for given $\mathbf{w}$ we will need a reference value to solve this system uniquely.

2.3 Linear systems

In the previous subsections three linear systems were derived

$$\Gamma\mathbf{v} = \mathbf{i}, \tag{2.10}$$
$$A\mathbf{i} = \mathbf{s}, \tag{2.11}$$
$$A^T\mathbf{w} = \mathbf{v}. \tag{2.12}$$

Our aim is to solve these systems and determine the value of the variables $\mathbf{i}, \mathbf{v}$ and $\mathbf{w}$. Many of the circuits Magma analyzes are extremely large, so the matrices in these linear systems are similarly large. Therefore numerical methods must be used for solving the systems. One way of simplifying these equations is the substitution of (2.12) into (2.10)

$$\Gamma A^T\mathbf{w} = \mathbf{i}. \tag{2.13}$$

Multiplying both sides of (2.13) with A makes it possible to substitute (2.11) into (2.13)

$$A\Gamma A^T\mathbf{w} = \mathbf{s} \qquad \Longrightarrow \qquad G\mathbf{w} = \mathbf{s}, \tag{2.14}$$

with $G = A\Gamma A^T$. This matrix G can also be determined in a direct way.

Theorem 2.3.1. *Every matrix G of the form $G = A\Gamma A^T$, with A an incidence matrix and Γ a positive diagonal matrix, can be constructed as*

$$G(i,i) = \sum_{j_q} \gamma_{j_q} \qquad \text{for } j_q \in \{j_p \mid \exists k, k \neq i, j_p \in J(i,k)\}, \tag{2.15}$$
$$G(i,k) = \sum_{j_q} -\gamma_{j_q} \qquad \text{for } j_q \in J(i,k), i \neq k, \tag{2.16}$$

with J(i,k) the set of branches connecting node i and node k.

Proof. G is an $n \times n$ matrix, so that suggests that its information in the i-th row and the k-th column is related to the interaction between node i and node k. We can calculate $G(i,k)$

$$\begin{aligned} G(i,k) &= \sum_{p=1}^{b}\sum_{q=1}^{b} a_{ip}\gamma_{pq}a_q k^T \\ &= \sum_{p=1}^{b} a_{ip}\gamma_{pp}a_{kp}. \end{aligned}$$

If node i is not directly connected to node k, i.e. if $J(i,k) = \emptyset$, we know $a_{ip}a_{kp} = 0$, because if $a_{ip}a_{kp} \neq 0$ would imply $a_{ip} = \pm 1$ and $a_{kp} = \pm 1$. By definition, node i would then be connected to node k. So we have $G(i,k) = 0$ if node i and node k are not directly connected. Next, assume node i is connected to node k in the graph by branches $j_q \in J(i,k)$. Then $a_{ip}a_{kp} = -1$ if $j_p \in J(i,k)$ and $a_{ip}a_{kp} = 0$ otherwise. Hence, $G(i,k) = \sum_{j_q} -\gamma_j$ for $j_q \in J(i,k)$, $i \neq k$. We know that $G(i,i) = \sum_{p=1}^{b} a_{ip}^2\gamma_{pp}$, and a_{ip}^2 is either equal to 0 or equal to 1, since A only contains the numbers 0, 1 and -1. So $a_{iq}^2\gamma_{qq} = \gamma_{qq}$ if $a_{iq} = \pm 1$, or in other words, if there exists a k such that $j_q \in J(i,k)$. So $G(i,i) = \sum_{j_q} \gamma_{j_q}$ for $j_q \in \{j_p \mid \exists k, k \neq i, j_p \in J(i,k)\}$. □

As an example, we show matrix G for the circuit in Figure 2.1

$$G = \begin{pmatrix} \gamma_a + \gamma_b & -\gamma_a & -\gamma_b & 0 & 0 \\ -\gamma_a & \gamma_a + \gamma_c + \gamma_e + \gamma_g & -\gamma_c & -\gamma_e & -\gamma_g \\ -\gamma_b & -\gamma_c & \gamma_b + \gamma_c + \gamma_d & -\gamma_d & 0 \\ 0 & -\gamma_e & -\gamma_d & \gamma_d + \gamma_e + \gamma_f & -\gamma_f \\ 0 & -\gamma_g & 0 & -\gamma_f & \gamma_f + \gamma_g \end{pmatrix}.$$

2.4 Matrix properties

In this section some properties of matrix G will be discussed. Matrix properties such as being symmetric, positive definite and diagonally dominant are important for choosing the appropriate solving method. First, we will give some definitions.

Definition 2.4.1. An $n \times n$ matrix B is *symmetric* if and only if

$$B = B^T. \tag{2.17}$$

Definition 2.4.2. A real $n \times n$ symmetric matrix B is *positive semi-definite* if for all nonzero $\mathbf{x} \in \mathbb{R}^n$ holds that $\mathbf{x}^T B\mathbf{x} \geq 0$. B is *positive definite* if the inequality strictly holds.

Clearly, matrix G is symmetric. If we take $\mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq \mathbf{0}$ we see that

$$\mathbf{x}^T G\mathbf{x} = (G\mathbf{x}, \mathbf{x}) = (A\Gamma A^T\mathbf{x}, \mathbf{x}) = (\Gamma A^T\mathbf{x}, A^T\mathbf{x}) = (\Gamma\mathbf{y}, \mathbf{y}) = \mathbf{y}^T\Gamma\mathbf{y} \geq 0, \tag{2.18}$$

because Γ is a diagonal matrix with positive nonzero entries on the diagonal. So G is positive semi-definite. However, since $Ker(A) \neq \emptyset$, G is not positive definite ($Ge = 0$ for $e = \{1, 1, \ldots, 1\}$) . The rank of A is $n - p$, with p the number of disjoint parts of the graph. We assume that the graph is connected, i.e. the rank of A equals $n - 1$. If we remove one row, say row i, from A and call the new matrix $\tilde{A}$, then the rank of $\tilde{A}$ is also $n - 1$, so $\tilde{A}$ is non-singular. This means that $A\mathbf{x} \neq 0$ for all $\mathbf{x} \in \mathbb{R}^{n-1}, \mathbf{x} \neq 0$. Clearly, the i-th row and i-th column of Γ have to be removed to keep consistency with A. Take $\tilde{\Gamma}$ the modified matrix Γ, then $\tilde{G} = \tilde{A}\tilde{\Gamma}\tilde{A}^T$ and we have for $\mathbf{x} \in \mathbb{R}^{n-1}, \mathbf{x} \neq 0$

$$\mathbf{x}^T\tilde{G}\mathbf{x} = (\tilde{\Gamma}\tilde{A}^T\mathbf{x}, \tilde{A}^T\mathbf{x}) = \mathbf{y}^T\tilde{\Gamma}\mathbf{y} > 0,$$

since $\tilde{\Gamma}$ is positive definite and $\mathbf{y} = \tilde{A}\mathbf{x} \neq 0$. The new system is

$$\tilde{G}\tilde{\mathbf{w}} = \tilde{\mathbf{s}}. \tag{2.19}$$

A physical interpretation of removing the i-th row and the i-th column is grounding node i, so $w_i = s_i = 0$.

A general matrix B can be represented by a *graph*. A connection from point i to j is a nonzero entry for $B(i, j)$. We call a graph *connected* if there is a path from an arbitrary point to every other point.

Definition 2.4.3. A matrix B is *reducible* if its graph is not connected, and *irreducible* otherwise [29].

Definition 2.4.4. A matrix B is *diagonally dominant* if

$$|B(i,i)| \geq \sum_{j=1, j\neq i}^{n} |B(i,j)|, \tag{2.20}$$

B is *strictly diagonally dominant* if the inequality is strict, i.e. if the $\geq$ is replaced by $>$. B is *irreducibly diagonally dominant* if B is irreducible and diagonally dominant, with a strict inequality for at least one i.

We will now define some special types of matrices.

Definition 2.4.5. An $n \times n$ matrix B is an *L-matrix* if $b_{ii} > 0, i = 1, \ldots, n$, and $b_{ij} \leq 0, i, j = 1, \ldots, n, i \neq j$.

Definition 2.4.6. An $n \times n$ matrix B is an *M-matrix* if B is a non-singular L-matrix and all elements of B^{-1} are non-negative.

The fact that $\tilde{G}$ is an L-matrix is quite clear, but just by using Definition 2.4.6 we will not be able to show that $\tilde{G}$ is an M-matrix. We will need the following Theorem

Theorem 2.4.7. *An irreducibly, diagonally dominant L-matrix is an M-matrix [29].*

Using this theorem, we can now conclude that $\tilde{G}$ is an M-matrix. This property will be of importance in the next section. In the remaining part of this thesis we will omit the superscript of (2.19).

Chapter 3

Solution methods

The linear system that has to be solved (2.19) was constructed in the previous section, now we will discuss some solution methods. The matrix G is symmetric positive definite and the most common used method for these matrices is the Conjugate Gradient method or a variant of it. This method and its variants will be discussed in this section.

3.1 Conjugate Gradient method

3.1.1 Steepest Descent method

The Conjugate Gradient method (CGM) is actually a "smart" way of formulating *the Steepest Descent method* (SDM), but before we will discuss these methods we take another look at the problem

$$G\mathbf{w} = \mathbf{s}. \tag{3.1}$$

This is equivalent to finding the minimum of $\varphi(\mathbf{w})$ for

$$\varphi(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T G\mathbf{w} - \mathbf{w}^T\mathbf{s}, \tag{3.2}$$

since φ is a convex functional and $\nabla\varphi(\mathbf{w}) = G\mathbf{w} - \mathbf{s}$. Hence, solving $\nabla\varphi(\mathbf{w}) = 0$ is equivalent to solving (3.1). SDM uses the fact that φ changes fastest in a direction that is opposite to its gradient. This suggests the following iteration

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \nabla\varphi(\mathbf{w}_k),$$

for a smart chosen α_k. We can define the residual $\mathbf{r}_k$ of approximation $\mathbf{w}_k$ by

$$\mathbf{r}_k = \mathbf{s} - G\mathbf{w}.$$

By taking

$$\alpha_k = \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_k^T G\mathbf{r}_k},$$

we minimize $\varphi(\mathbf{w}_k + \alpha_k\mathbf{r}_k)$.

Lemma 3.1.1.

$$\mathbf{r}_k^T\mathbf{r}_{k+1} = 0 \qquad \textit{for all } k \geq 0. \tag{3.3}$$

Proof. Proof: Take $k \geq 0$. Then

$$\begin{aligned}
\mathbf{r}_k^T \mathbf{r}_{k+1} &= \mathbf{r}_k^T \mathbf{s} - \mathbf{r}_k^T G \mathbf{w}_k - \alpha_k \mathbf{r}_k A \mathbf{r}_k \\
&= \mathbf{r}_k^T \mathbf{s} - \mathbf{r}_k^T G \mathbf{w}_k - \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_k^T G \mathbf{r}_k} \mathbf{r}_k G \mathbf{r}_k \\
&= \mathbf{r}_k^T \mathbf{s} - \mathbf{r}_k^T G \mathbf{w}_k - \mathbf{r}_k^T (\mathbf{s} - G \mathbf{w}_k) \\
&= 0.
\end{aligned}$$

□

Algorithm 1 Steepest Descent

$w_0 = 0$
$r_0 = s$
for $k = 0, 1, \ldots$ **do**
  $\alpha_k = (r_k, r_k)/(r_k, Gr_k)$
  $w_{k+1} = w_k + \alpha_k r_k$
  $r_{k+1} = s - Gw_{k+1}$
  if $r_{k+1} < \varepsilon$ **then**
    break
  end if
end for

We can now formulate the Steepest Descent method in Algorithm 1. The SDM leads to convergence for any non-singular symmetric positive definite matrix, but the convergence speed can be rather slow. The search direction is equal to the direction of the negative gradient. The algorithm doesn't use all previous search directions, only the last one, so it is possible that two different search direction are taken over and over again. If the information of all preceding search directions is stored the efficiency of the algorithm is improved. This is the basis of the CGM.

3.1.2 Search Directions

To store all previous search directions we introduce the vector $\mathbf{p}_k$, defined by

$$\mathbf{p}_k = \mathbf{r}_k + \sum_{i=0}^{k-1} \eta_{k,i} \mathbf{r}_i. \tag{3.4}$$

This gives a new iterative relation for $\mathbf{w}_k$

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k.$$

We can find α by minimizing the search direction in one dimension, giving

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T G \mathbf{p}_k}.$$

We find for the residual

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k G \mathbf{p}_k.$$

If we now use the property of (3.3) for (3.4) we get

$$\mathbf{p}_k^T \mathbf{r}_{k+1} = \mathbf{p}_k \mathbf{r}_k - \alpha_k \mathbf{p}_k^T G \mathbf{p}_k = 0. \tag{3.5}$$

If now $\mathbf{r}_i\mathbf{r}_{k+1} = 0$ for all $0 \leq i \leq k-1$ we see that

$$\mathbf{r}_k^T\mathbf{r}_{k+1} = \mathbf{p}_k^T\mathbf{r}_{k+1} - \sum_{i=0}^{k-1} \eta_{k,i}\mathbf{r}_i^T\mathbf{r}_{k+1} = 0.$$

So all residuals are mutually orthogonal if we find a set $\{\eta_{k,0}, \ldots, \eta_{k,k-1}\}$ such that $\mathbf{r}_i^T\mathbf{r}_k + 1 = 0$ for all $0 \leq i \leq k-1$. Now we will prove that there is such a set. Take

$$\mathbf{r}_j^T\mathbf{r}_{k+1} = \mathbf{r}_j^T\mathbf{r}_k^T - \alpha_k\mathbf{r}_j^TG\mathbf{p}_k = -\alpha_k\mathbf{r}_j^TG\mathbf{r}_k - \alpha_k\sum_{i=0}^{k-1}\eta_{k,i}\mathbf{r}_j^TG\mathbf{r}_i.$$

Now compose the matrix R_{k-1} by using r_0 to r_{k-1} as its columns. Then

$$\mathbf{r}_j^T\mathbf{r}_{k+1} = 0, \qquad 0 \leq j \leq k-1,$$

is equivalent to

$$R_{k-1}^TGR_{k-1}(\eta_{k,0}, \ldots, \eta_{k,k-1}, 1)^T = -R_{k-1}^TG\mathbf{r}_k. \tag{3.6}$$

This last condition requires α_k to be nonzero. We know G is positive definite and non-singular, so $R_{k-1}^TGR_{k-1}$ has the same properties. Therefore, (3.6) has an unique solution. Now we have to find the coefficients $\eta_{k,i}$. To accomplish this we are going to need some lemma's.

Lemma 3.1.2. *If G is symmetric, then*

$$\mathbf{r}_j^TG\mathbf{r}_k = 0. \tag{3.7}$$

Proof.

$$\begin{aligned}\mathbf{r}_j^TG\mathbf{r}_k = \mathbf{r}_k^TG\mathbf{r}_j &= \mathbf{r}_k^TG(\mathbf{p}_j - \sum_{i=0}^{j-1}\eta_{j,i}\mathbf{r}_i)\\ &= \frac{1}{\alpha_j}\mathbf{r}_k^T(\mathbf{r}_j - \mathbf{r}_{j+1}) - \sum_{i=0}^{j-1}\eta_{j,i}\mathbf{r}_k^TG\mathbf{r}_i\\ &= -\sum_{i=0}^{j-1}\eta_{j,i}\mathbf{r}_i^TG\mathbf{r}_k.\end{aligned}$$

Induction completes the proof. □

Using this Lemma we can formulate a recurrent relation for $\eta_{k,i}$. Take $k \geq 2$ and $0 \leq j \leq k-2$, then we get

$$\sum_{i=0}^{k-2}\eta_{k-1,i}\mathbf{r}_j^TG\mathbf{r}_i + \mathbf{r}_j^TG\mathbf{r}_{k-1} = 0.$$

Now multiplication of this equation by any nonzero δ and substitution of (3.7) results into

$$\sum_{i=0}^{k-2}\delta\eta_{k-1,i}\mathbf{r}_j^TG\mathbf{r}_i + \delta\mathbf{r}_j^TG\mathbf{r}_{k-1} + \mathbf{r}_j^TG\mathbf{r}_k = 0.$$

If we take $\delta = \eta_{k,k-1}$ we find the recurrence relation

$$\eta_{k,i} = \eta_{k,k-1}\eta_{k-1,i} \qquad 0 \leq i \leq k-2.$$

Lemma 3.1.3. *If G is symmetric, the search directions satisfy*

$$\mathbf{p}_k = \mathbf{r}_k + \eta_{k,k-1}\mathbf{p}_{k-1}. \tag{3.8}$$

Proof.

$$\mathbf{p}_k = \mathbf{r}_k + \sum_{i=0}^{k-1} \eta_{k,i}\mathbf{r}_i = \mathbf{r}_k + \eta_{k,k-1}(\mathbf{r}_{k-1} + \sum_{i=0}^{k-2} \eta_{k-1,i}\mathbf{r}_i) = \mathbf{r}_k + \eta_{k,k-1}\mathbf{p}_{k-1}.$$

□

So the recent search directions can be found using only the previous search directions. This saves a lot of computing time. We can determine parameter $\eta_{k,k-1}$ by requiring that $\mathbf{p}_{k-1}^T\mathbf{r}_{k+1} = 0$. This gives

$$\gamma_{k,k-1} = -\frac{\mathbf{p}_{k-1}^T G\mathbf{r}_k}{\mathbf{p}_{k-1}^T G\mathbf{p}_{k-1}}.$$

Lemma 3.1.4. *The search directions satisfy*

$$\mathbf{p}_j^T G\mathbf{p}_i = 0, \qquad i \neq j. \tag{3.9}$$

Proof.

$$\mathbf{p}_j^T\mathbf{r}_{i+1} = \mathbf{p}_j^T\mathbf{r}_i - \alpha_k\mathbf{p}_j^T G\mathbf{p}_i.$$

We see that $\mathbf{p}_j^T\mathbf{r}_{i+1} = \mathbf{p}_j^T\mathbf{r}_i = 0$. This completes the proof. □

This lemma explains why this method is called the Conjugate Gradient method: the search directions are perpendicular with respect to the inner product

$$[\mathbf{p}_j, \mathbf{p}_i]_G = \mathbf{p}_j^T G\mathbf{p}_i. \tag{3.10}$$

To improve the computation speed, we can reformulate the expressions for α_k and $\eta_{k,k-1}$ by

$$\begin{aligned} \alpha_k &= \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{p}_k^T G\mathbf{p}_k}, \\ \eta_{k,k-1} &= \frac{\mathbf{r}_k^T\mathbf{r}_k}{\mathbf{r}_{k-1}^T\mathbf{r}_{k-1}}. \end{aligned}$$

Since the algorithm doesn't use $\eta_{k,0}, \ldots, \eta_{k,k-2}$ we can define $\beta_{k-1} = \eta_{k,k-1}$. Algorithm 2 uses the theory above to solve (3.1)

Algorithm 2 Conjugate Gradient

```
w_0 = 0
r_0 = s
for k = 1, 2, ... do
  ρ_{k-1} = (r_{k-1}, r_{k-1})
  if k = 1 then
    p_k = r_0
  else
    β_{k-1} = ρ_{k-1}/ρ_{k-2}
    p_k = r_{k-1} + β_{k-1} p_{k-1}
  end if
  q_k = G p_k
  α_k = ρ_{k-1}/(p_k, q_k)
  w_k = w_{k-1} + α_k p_k
  r_k = r_{k-1} − α_k q_k
  if ||r_k||_2 < ε then
    break
  end if
end for
```

3.2 Cholesky Factorization

Another way of solving the system (3.1) is to solve it directly using a *Cholesky decomposition*, which is the symmetric version of the LU-decomposition. Because G is a symmetric M-matrix the decomposition

$$G = LL^T, \tag{3.11}$$

with L a lower triangular matrix, is always possible and unique with positive diagonal elements ([6]). The system can then be solved by calculating consecutively

$$\begin{aligned} Ly &= s, \\ L^T z &= y. \end{aligned}$$

The algorithm to calculate the elements of L is described in Algorithm 3.

Algorithm 3 Cholesky factorization (CF)

for $j = 1, \ldots, n$ **do**

$$\ell_{jj} = \sqrt{g_{jj} - \sum_{k=1}^{j-1} (l_{jk})^2}$$

for $i = j+1, \ldots, n$ **do**

$$\ell_{ij} = \tfrac{1}{\ell_{jj}} \sqrt{g_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}}$$

end for

end for

For complete Cholesky factorization the sparsity pattern of the L matrix is generally not equal to that of the lower diagonal matrix of G. We see that $\ell_{ij} \neq 0$ if $g_{ij} \neq 0$ or $\ell_{ik}\ell_{jk} \neq 0$ for some $k < i, j$. The nonzero elements ℓ_{ij} are called *fill-in* if $g_{ij} = 0$.

The physical dimension of G is $siemens$, so the dimension of L is $\sqrt{siemens}$, which is unsuitable for we want a dimensionless L (the reason is discussed later in this chapter). A good alternative is to use the factorization $G = LDL^T$, which can be derived quite simply. If we demand L to be a lower triangular matrix with the unity vector on the diagonal and D to be a diagonal matrix we get

$$\begin{aligned} G &= LDL^T. \\ g_{ij} &= \sum_{k=1}^{n} \sum_{p=1}^{n} \ell_{ik} d_{kp} \ell_{pj}^T \\ &= \sum_{k=1}^{n} \ell_{ik} d_{kk} \ell_{kj}^T \\ &= \sum_{k=1}^{j} \ell_{ik} d_{kk} \ell_{jk}, \end{aligned}$$

for $i \geq j$, because $\ell_{ij} = d_{ij} = 0$ by definition if $j > i$. We get

$$\begin{aligned} g_{jj} &= \sum_{k=1}^{j} \ell_{jk} d_{kk} \ell_{jk} \\ &= \ell_{jj}^2 d_{jj} \sum_{k=1}^{j-1} \ell_{jk}^2 d_{kk} \\ &= d_{jj} \sum_{k=1}^{j-1} \ell_{jk}^2 d_{kk} \\ \Longrightarrow d_{jj} &= g_{jj} - \sum_{k=1}^{j-1} (\ell_{jk})^2 d_{kk}, \end{aligned}$$

and

$$\begin{aligned} g_{ij} &= \sum_{k=1}^{j} \ell_{ik} d_{kk} \ell_{jk} \\ &= \sum_{k=1}^{j-1} (\ell_{ik} d_{kk} \ell_{jk}) + \ell_{ij} d_{jj} \ell_{jj} \\ &= \sum_{k=1}^{j-1} (\ell_{ik} d_{kk} \ell_{jk}) + \ell_{ij} d_{jj} \\ \Longrightarrow \ell_{ij} &= \frac{1}{d_{jj}} (g_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk}). \end{aligned}$$

Algorithm 4 Left-looking diagonal Cholesky factorization (LLDCF)

for $j = 1, \ldots, n$ **do**
  $\ell_{jj} = 1$
  $d_{jj} = g_{jj} - \sum_{k=1}^{j-1} (\ell_{jk})^2 d_{kk}$
  for $i = j + 1, \ldots, n$ **do**
    $\ell_{ij} = \frac{1}{d_{jj}} (g_{ij} - \sum_{k=1}^{j-1} \ell_{ik} d_{kk} \ell_{jk})$
  end for
end for

Algorithm 4 is a backward factorization, because it uses elements in the matrix that are already processed. This algorithm is correct, but the construction of the matrices L and D can be done using much less computer time. Consider Algorithm 4; we need to do a summation of some previous found entries for every new entry. If we look, for example, at the off-diagonal elements, then for every new element we need to look back at the values of ℓ_{1i} and ℓ_{1j}. So it is more efficient to put the information of ℓ_{1i} and ℓ_{1j} inside the entries of $\ell_{ij}, i, j > 1$ at the moment ℓ_{1i} and ℓ_{1j} are determined. Algorithm 5 is based on this, it stores almost all the information needed to determine a certain entry before the entry is in fact determined. Take $a[i, j]$ to be an array, with initially $a[i, j] = g_{ij}$ for all $i, j = 1 \ldots n$.

Theorem 3.2.1. *The output of Algorithm 5 is $a[j, j] = d_{jj}$ and $a[i, j] = \ell_{ij}$ for all $i, j = 1 \ldots n, i > j$.*

Proof. We use the invariant $P(p) \equiv P_1(p, p) \wedge P_2(p, p) \wedge Q_1(p) \wedge Q_2(p)$, with

- $P_1(p,q) \equiv \{\forall i \forall j, p \leq j < i : \ a[i,j] = g_{ij} - \sum_{k=1}^{q-1} \ell_{ik} d_{kk} \ell_{jk}\}$.
- $P_2(p,q) \equiv \{\forall i, i \geq p : \ a[j,j] = g_{jj} - \sum_{k=1}^{q-1} \ell_{jk}^2 d_{kk}\}$.
- $Q_1(p) \equiv \{\forall i \forall j, j < p, j < i : \ a[i,j] = \ell_{ij}\}$.
- $Q_2(p) \equiv \{\forall j, j < p : \ a[j,j] = d_{ij}\}$.

If we now prove that $P(1)$ holds and that if $P(p)$ holds also $P(p+1)$ holds we can use induction to proof that $P(n)$ holds.

$P(1)$:

- $P_1(1,1) : \{\forall i \forall j, 1 \leq j < i : \ a[i,j] = g_{ij}\}$ holds.
- $P_2(1,1) : \{\forall j, j \geq 1 : \ a[j,j] = g_{jj} \ \}$ holds.
- $Q_1(1) : \{$ true $\}$
- $Q_2(1) : \{$ true $\}$

So we have $P(1)$. Say we have $P(p)$:

- $P_1(p,p) \equiv \{\forall i \forall j, p \leq j < i : \ a[i,j] = g_{ij} - \sum_{k=1}^{p-1} \ell_{ik} d_{kk} \ell_{jk}\}$.
- $P_2(p,p) \equiv \{\forall i, i \geq p : \ a[j,j] = g_{jj} - \sum_{k=1}^{p-1} \ell_{jk}^2 d_{kk}\}$.
- $Q_1(p) \equiv \{\forall i \forall j, j < p, j < i : \ a[i,j] = \ell_{ij}\}$.
- $Q_2(p) \equiv \{\forall j, j < p : \ a[j,j] = d_{ij}\}$.

Then

- $Q_2(p+1) \equiv \{\forall i \forall j, j < p+1, j < i : \ a[i,j] = \ell_{ij}\}$. Since $Q_2(p)$ before iterating we only need $a[p,p] = d_{pp}$. We have $P_2(p,p)$, so $a[p,p] = g_{pp} - \sum_{k=1}^{p-1} \ell_{pk}^2 d_{kk} = d_{pp}$.
- $Q_1(p+1) \equiv \{\forall i \forall j, j < p+1, j < i : \ a[i,j] = \ell_{ij}\}$. Since $Q_1(p)$ before iterating we only need $a[i,p] = \ell_{ip}$. We have $P_1(p,p)$ so $a[i,p] = \dfrac{a[i,p]}{a[p,p]} = \dfrac{1}{d_{pp}}(g_{ip} - \sum_{k=1}^{p-1} \ell_{ik} d_{kk} \ell_{pk}) = \ell_{ip}$
- $P_1(p+1,p)$ and $P_2(p+1,p)$ follow directly from $P_1(p,p)$ and $P_2(p,p)$, respectively.
- $P_2(p+1,p+1) \equiv \{\forall j, j \geq p+1 : \ a[j,j] = g_{jj} - \sum_{k=1}^{p} \ell_{jk}^2 d_{kk}\}$. We know $P_2(p+1,p)$ halfway, so $\forall j, j \geq p+1 : \ a[j,j] = g_{jj} - \sum_{k=1}^{p-1} \ell_{jk}^2 d_{kk}$. Then $a[j,j] = a[j,j] - a[j,p]^2 a[p,p] = g_{jj} - \sum_{k=1}^{p-1} (\ell_{jk}^2 d_{kk}) - \ell_{jn}^2 d_{pp} = g_{jj} - \sum_{k=1}^{p} \ell_{jk}^2 d_{kk}$. This is true for all $j \geq p+1$, so $P_2(p+1,p+1)$ holds.

- $P_1(p+1,p+1)\{\equiv \forall i \forall j, p+1 \leq j < i : \ a[i,j] = g_{ij} - \sum_{k=1}^{p} \ell_{ik} d_{kk} \ell_{jk}\}$. We know $P_1(p+1,p)$ halfway, so $\forall i \forall j, p+1 \leq j < i : \ a[i,j] = g_{ij} - \sum_{k=1}^{p-1} \ell_{ik} d_{kk} \ell_{jk}$. Then $a[i,j] = a[i,j] - a[i,p]a[p,p]a[j,p] = g_{ij} - \sum_{k=1}^{p-1} (\ell_{ik} d_{kk} \ell_{jk}) - \ell_{ip} d_{pp} \ell_{jp} = g_{ij} - \sum_{k=1}^{p} \ell_{ik} d_{kk} \ell_{jk}$. This is true for all $i, j, p+1 \leq j < i$, so $P_1(p+1,p+1)$ holds.

So given $P(p)$ we have $P(p+1)$. This concludes the proof. □

Algorithm 5 Right-looking Diagonal Cholesky Factorization (RLDCF)

$\{P(0)\}$
for $p = 1, \ldots, n$ **do**
  $\{P(p)\}$
  for $i = p+1, \ldots, n$ **do**
    $a[i,p] = \dfrac{a[i,p]}{a[p,p]}$
  end for
  $\{Q_1(p+1) \wedge Q_2(p+1) \wedge P_1(p+1,p) \wedge P_2(p+1,p)\}$
  for $j = p+1, \ldots, n$ **do**
    $a[j,j] = a[j,j] - a[j,p]^2 a[p,p]$
    for $i = j+1, \ldots, n$ **do**
      $a[i,j] = a[i,j] - a[i,p]a[p,p]a[j,p]$
    end for
  end for
  $\{Q_1(p+1) \wedge Q_2(p+1) \wedge P_1(p+1,p+1) \wedge P_2(p+1,p+1)\} =$
  $\{P(p+1)\}$
end for
$\{P(n)\}$

Now the factorization is forward, since the information is of the current element is put in elements that are not visited yet. We have $G = K = LDL^T$, so solving $Kz = r$ can be done by solving successively

$$\begin{aligned} Lx &= r, \\ Dy &= x, \\ L^T z &= y. \end{aligned}$$

3.3 Preconditioned Conjugate Gradient method

The CGM converges fast for matrices that are well conditioned or have a few distinct eigenvalues. However, this is not generally the case. Using the Cholesky decomposition, the system will be solved directly without iteration steps, but the number of nonzero element might be too big to be used in the memory of a computer. The solution is to combine both methods, by means of transformation of the linear system. The transformed system has the same solution, but is easier to solve using an iterative solver. This process is called *preconditioning*.

Preconditioning techniques for the CGM involve a preconditioner matrix K. What we basically do is find a K such that $K\mathbf{w} = \mathbf{s}$ is easier to solve. K is symmetric positive definite, so we can define the K-inner product

$$[\mathbf{x}, \mathbf{y}]_K \equiv (K\mathbf{x}, \mathbf{y}) = (\mathbf{x}, K\mathbf{y}). \tag{3.12}$$

If we rewrite the algorithm of the CGM for this inner product, and we use the new residual $Kz_j = r_j$ we find the Preconditioned Conjugate Gradient method.

Algorithm 6 Preconditioned Conjugate Gradient

$w_0 = 0$
$r_0 = s$
for $k = 1, 2, \ldots$ **do**
  Solve $Kz_{k-1} = r_{k-1}$
  $\rho_{k-1} = (r_{k-1}, z_{k-1})$
  if $k = 1$ **then**
    $p_k = z_0$
  else
    $\beta_{k-1} = \rho_{k-1}/\rho_{k-2}$
    $p_k = z_{k-1} + \beta_{k-1} p_{k-1}$
  end if
  $q_k = Gp_k$
  $\alpha_k = \rho_{k-1}/(p_k, q_k)$
  $w_k = w_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1} - \alpha_k q_k$
  if $||r_k||_2 < \varepsilon$ **then**
    break
  end if
end for

3.4 Search for the right preconditioner

The Preconditioned Conjugate Gradient algorithm is formulated above, but we still lack a preconditioner matrix. Several suitable matrices will be discussed in this section.

3.4.1 Incomplete Cholesky factorization

In Algorithm 3 the complete Cholesky factorization is shown. We can get an *incomplete Cholesky factorization* by neglecting all values outside the sparsity pattern. This is shown in Algorithm 7.

Algorithm 7 Incomplete Cholesky factorization (ICF)

for $j = 1, \ldots, n$ **do**
  $\ell_{jj} = \sqrt{g_{jj} - \sum_{k=1}^{j-1} (l_{jk})^2}$
  for $i = j + 1, \ldots, n$ **do**
    if $g_{ij} = 0$ **then**
      $\ell_{ij} = 0$
    else
      $\ell_{ij} = \frac{1}{\ell_{jj}} \sqrt{g_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}}$
    end if
  end for
end for

All fill-in is neglected, so the memory needed for the preconditioner matrix is equal to the memory needed for the original matrix.

3.4.2 Standard incomplete Cholesky factorization

Van der Vorst ([23],[30]) suggested the use of the standard incomplete Cholesky factorization. Take $G = L + diag(G) + L^T$, for which L is now a strictly lower triangular matrix. Then we are looking for the preconditioner

$$K = (L + D)D^{-1}(D + L^T). \tag{3.13}$$

The diagonal matrix D can be found using the condition

$$diag(G) = diag(K). \tag{3.14}$$

Take $L + D = C$, then

$$\begin{aligned} k_{ij} &= \sum_{p=1}^{n}\sum_{q=1}^{n} c_{ip}\frac{1}{d_{pq}}c_{qj}^T \\ &= \sum_{p=1}^{i} c_{ip}c_{jp}\frac{1}{d_{pp}}, \\ \Longrightarrow k_{ii} &= \sum_{p=1}^{i} c_{ii}^2\frac{1}{d_{pp}} \\ &= c_{ii}^2\frac{1}{d_{ii}} + \sum_{p=1}^{i-1} c_{ii}^2\frac{1}{d_{pp}}. \end{aligned}$$

Since $c_{ii} = d_{ii}$, $c_{ij} = g_{ij}$ for $i > j$, and $k_{ij} = g_{ij}$ we find the following relation to determine d_{ii}

$$d_{ii} = g_{ii} - \sum_{p=1}^{i-1}\frac{a_{ip}^2}{d_{pp}}. \tag{3.15}$$

If we now use the same strategy we did to find Algorithm 5, we get Algorithm 8.

Algorithm 8 Right-looking standard incomplete Cholesky factorization (RLSICF)

for $p = 1, \ldots, n$ **do**
  for $j = p + 1, \ldots, n$ **do**
    $a[j, j] = a[j, j] - \dfrac{a[j, p]^2}{a[p, p]}$
  end for
end for

Gustafsson ([13]) came up with the modified incomplete Cholesky factorization, which is the standard factorization we used just before, but with the additional condition

$$Rowsum(G) = Rowsum(K). \tag{3.16}$$

Using similar techniques as for (3.14) we find

$$d_{ii} = g_{ii} - \sum_{p=1}^{i-1}\sum_{j=p+1}^{n}\frac{a_{ip}a_{jp}}{d_{pp}}.$$

Using the forward method we get Algorithm 9.

Algorithm 9 Right-looking modified incomplete Cholesky factorization (RLMICF)

for $p = 1, \ldots, n$ **do**
  for $j = p + 1, \ldots, n$ **do**
    for $i = p + 1, \ldots, n$ **do**
      $a[j, j] = a[j, j] - \dfrac{a[j, p]a[i, p]}{a[p, p]}$
    end for
  end for
end for

In response to Gustafssons modified method, Van der Vorst ([30]) saw the coherence between the two previous factorizations and introduced a parameter δ. If $\delta = 0$, we have the standard factorization, and if $\delta = 1$ we have the modified factorization. Basically this means that for Algorithm 9 we multiply the term $\frac{a[j,p]a[i,p]}{a[p,p]}$ with δ if $j \neq i$. This is implemented in Algorithm 10.

Algorithm 10 Right-looking modified incomplete Cholesky factorization (δ) (RLMICFD)

for $p = 1, \ldots, n$ **do**
  for $j = p + 1, \ldots, n$ **do**
    for $i = p + 1, \ldots, n$ **do**
      if $i = j$ **then**
        $a[j, j] = a[j, j] - \dfrac{a[j, p]a[i, p]}{a[p, p]}$
      else
        $a[j, j] = a[j, j] - \delta\dfrac{a[j, p]a[i, p]}{a[p, p]}$
      end if
    end for
  end for
end for

3.4.3 Drop tolerance

Currently, Magma is working with the factorization of G described in Algorithm 5, with a certain adjustment to reduce amount of fill-in, which is called *drop tolerance*. Drop tolerance means that we choose a certain threshold ε, and we only update a value of $a[i, j]$ if $a[i, j] \neq 0$ *or* if the update is larger than ε.

Algorithm 11 Right-looking diagonal Cholesky factorization with drop tolerance (FDCFDT)

for $p = 1, \ldots, n$ **do**
  $a[p, p] = a[p, p]$
  for $i = p + 1, \ldots, n$ **do**
    $a[i, p] = \dfrac{a[i, p]}{a[p, p]}$
  end for
  for $j = p + 1, \ldots, n$ **do**
    $a[j, j] = a[j, j] - a[j, p]^2 a[p, p]$
    for $i = j + 1, \ldots, n$ **do**
      if $a[i, j] \neq 0$ or $a[i, p]a[p, p]a[j, p] > \varepsilon$ **then**
        $a[i, j] = a[i, j] - a[i, p]a[p, p]a[j, p]$
      end if
    end for
  end for
end for

Since we want ε to be dimensionless it is important that the values of L are also dimensionless.

3.5 Solver problems

The Algorithms 6 and 11 form the basis of the current solver. The variable ε represents a connection between the number of iteration steps and the number of fill-in elements. A small ε results in a small amount of iteration steps, but almost complete fill-in, while a large ε results in little fill-in, but many iteration steps. For the current, already quite large threshold the number of fill-in elements can swamp the memory if a very large design is analyzed. This is fatal for the performance of the solver. Therefore Magma needs a method that either reduces the number of iterations (so a large ε can be chosen) or reduces the number of fill-in elements (so a smaller ε can be chosen), or does both. A third option is to exploit parallelism. Parallelism means that the system is solved using multiple processors. This can only be done if the data that is solved by one processor is independent of the data solved by all other processors at that time.

In the next section and coming chapters several methods will be discussed that may have one of the following results:

- Reduction of the number of fill-in elements.
- Reduction of the number of iteration steps.
- Parallelism.

3.6 Solution methods

Inspired by the factorizations of Van der Vorst and Gustafsson, we made a factorization with drop tolerance using a combination of the properties (3.14) and (3.16).

Algorithm 12 Right-looking diagonal Cholesky factorization with drop tolerance (θ) (RLDCFDTT)

for $p = 1, \ldots, n$ **do**
  $a[p,p] = a[p,p]$
  for $i = p+1, \ldots, n$ **do**
    $a[i,p] = \dfrac{a[i,p]}{a[p,p]}$
  end for
  for $j = p+1, \ldots, n$ **do**
    $a[j,j] = a[j,j] - a[j,p]^2 a[p,p]$
    for $i = j+1, \ldots, n$ **do**
      if $a[i,j] \neq 0$ or $a[i,p]a[p,p]a[j,p] > \varepsilon$ **then**
        $a[i,j] = a[i,j] - a[i,p]a[p,p]a[j,p]$
      else
        $a[i,i] = a[i,i] - \theta a[i,p]a[p,p]a[j,p]$
        $a[j,j] = a[j,j] - \theta a[i,p]a[p,p]a[j,p]$
      end if
    end for
  end for
end for

These ideas might also give one of the three results described in the previous section

- Saad's idea to allow only the μ biggest nonzero entries per row. This can be used without or in combination with the drop tolerance.

- Instead of adjusting D so that the $diag(K) = diag(G)$ during the factorization, adjust them after the process.
- Find a permutation of matrix G. This process is called matrix ordering .

The first two ideas gave rather disappointing results. The third however appeared to be a lively and widely studied topic during the last decades. We will discuss matrix ordering during the next three chapters.

Chapter 4

Matrix Ordering

In the previous section we discussed some factorization methods to generate a suitable preconditioner matrix for the PCG-method. Now we look for methods to reduce the numbers of nonzero elements in matrix L. This can be achieved by reordering the matrix G.

Our aim is to find a permutation matrix P such that $PGP = \tilde{L}\tilde{D}\tilde{L}^T$ and the number of nonzero elements of $\tilde{L}$ is minimized. Unfortunately, finding such a permutation matrix P is NP-complete [8]. Therefore we need to use heuristics. The best known heuristics are the Minimum Degree ordering, the indexReverse Cuthill-McKee ordering Reverse Cuthill-McKee ordering, and the Nested Dissection ordering. These ordering methods will be discussed in the following chapters.

4.1 The benefits of ordering

We will underline the importance of ordering by using an example: in Algorithm 5 we see that $a[i,j]$ changes if both $a[i,p]$ and $a[j,p]$ are nonzero elements. So if $a[i,j]$ was initially zero, we now have to store an extra element. We called those new nonzero elements *fill-in*. However, for almost every matrix it is possible to reduce most fill-in by reorganizing the matrix. As an example, look at the following (arrow) matrix (in which the x represent a nonzero matrix value) and its Choleskey factor below.

$$G = \begin{pmatrix} x & x & x & x & x \\ x & x & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 \\ x & 0 & 0 & x & 0 \\ x & 0 & 0 & 0 & x \end{pmatrix}, \qquad L = \begin{pmatrix} x & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ x & x & x & x & 0 \\ x & x & x & x & x \end{pmatrix}.$$

The matrix L has maximum fill-in. If we now use a permutation that switches row and column 1 with row and column 5 and row and column 2 with row and column 4 we see that there is no fill-in created at all.

$$G = \begin{pmatrix} x & 0 & 0 & 0 & x \\ 0 & x & 0 & 0 & x \\ 0 & 0 & x & 0 & x \\ 0 & 0 & 0 & x & x \\ x & x & x & x & x \end{pmatrix}, \qquad L = \begin{pmatrix} x & 0 & 0 & 0 & 0 \\ 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \\ x & x & x & x & x \end{pmatrix}.$$

This example shows that an ordering can reduce the number of nonzero elements that need to be stored significantly. However, since most circuit matrices are very large and quite strongly connected we have to come up with algorithms to order them. The table below gives an idea of the impact of ordering and shows the importance to choose an appropriate ordering algorithm.

Table 4.1: Nonzero elements of the complete Cholesky factor of testcase3 for several orderings.

Ordering	Nonzero elements of L
MINOLD	1,657,581
Matlab AMD	196,738
METIS oemetis	244,233
Matlab RCM	746,314

MINOLD is the current ordering Magma is using. AMD stands for Approximate Minimum Degree and RCM stands for Reverse Cuthill-McGee. METIS ([18]) is a software package of the University of Minnesota. These algorithms are mainly based on the graph representation of the matrix, so this representation will be discussed in the next section.

4.2 Graph representation

Symmetric matrices have perfectly clear graph representations. Assume F is the graph of matrix G, $F = (V, E)$, with node $v_i \in V$ represents the i-th column/row, and edge $e_{ij} \in E$ represents a nonzero value in row i and column j. We define the nodes adjacent to v_i by

$$Adj(v_i) := \{v_j \in V \mid e_{ij} \in E\}, \tag{4.1}$$

and the adjacent set of a set X by

$$Adj(X) := \{v_j \in V \backslash X \mid e_{ij} \in E \; for \; some \; v_i \in X\}. \tag{4.2}$$

The degree of a node, denoted by d_i, is defined by

$$d_i := |Adj(v_i)|. \tag{4.3}$$

Another useful set is the reach of a node v_i through a set X, denoted by $Reach(v_i, X)$

$$Reach(v_i, X) := \{v_j \notin X \mid v_j, v_i \in Adj(X), v_j \neq v_i\}. \tag{4.4}$$

Our goal is to find a permutation $\underline{\pi} = \{\pi_1, \pi_2, \ldots, \pi_n\}$, so first we label node v_{π_1}, then v_{π_2} etc. The creation of fill-in goes analogously to the matrix problem; if node v_i is connected to the nodes v_k and v_j with no connection between the last two nodes, an edge is created between v_k and v_j if v_i is labelled first. We can use *elimination graphs* to describe the nonzero pattern of the submatrix after the labelling of the k-th node. The initial elimination graph $F^0 = (V^0, E^0)$ is the original graph. After step k we have $F^k = (V^k, E^k)$, and to construct F^{k+1} we choose a node v_i to label, and then remove it and all adjacent edges from the graph. Next, all nodes adjacent to the eliminated node are fully connected with each other, so, in other words, they form a *clique*. In Figure 4.1 a small circuit is shown as an elimination graph. The figure also contains the elimination graphs after eliminating the nodes in the current order of labelling.

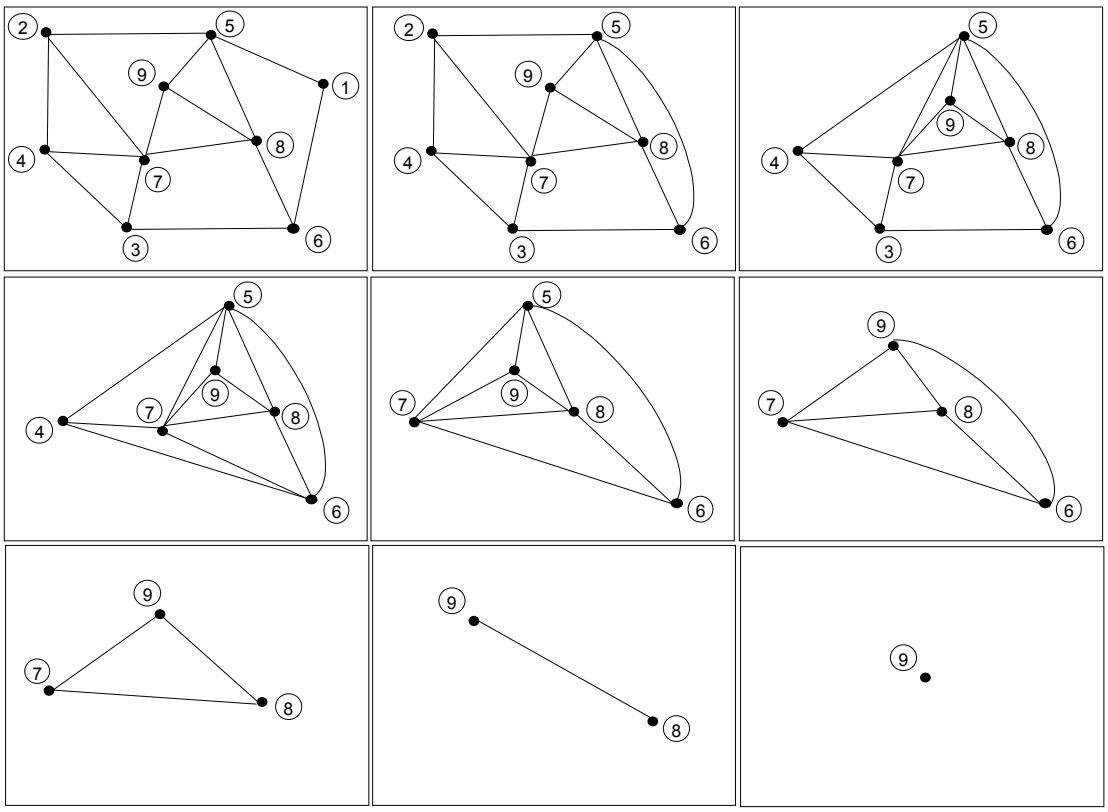

Figure 4.1: The elimination graphs $F^0 \ldots F^8$ of a simple circuit.

The set $Adj(v_i)_{F^k}$ represent the adjacent nodes of v_i in the elimination graph F^k. Each node has a *score function*, a value $score(v_i)$. A possible way to determine the node to be eliminated next is a node with $score(v_i) \leq score(v_j),\ v_j \in V^k$.

There are two types of ordering algorithms: local ordering algorithms and global ordering algorithms. Local ordering algorithms use graphs like elimination graphs, the nodes are chosen one at the time and after each elimination the next node is chosen by some metric, in most cases the score function. Global ordering methods use the structure of the graph and try to minimize the interaction between the nodes by separating them.

Before we discuss the different ordering methods we look at the structure of a matrix before ordering, and the same matrix ordered with the current ordering method MINOLD. Such a matrix, testcase1, and its L matrix are shown in Figures 4.2 and 4.3. All nonzero elements are coloured and all elements that are zero are blank.

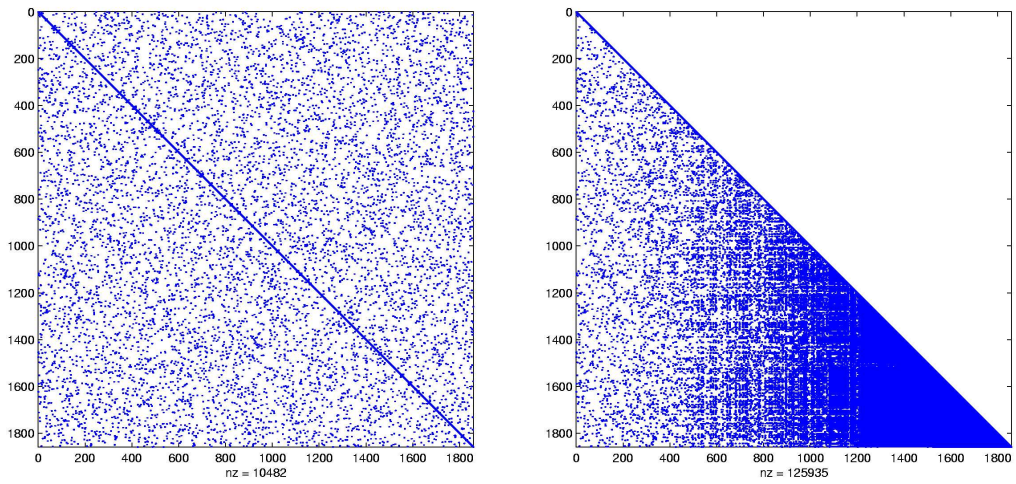

Figure 4.2: Sparsity pattern of testcase1 and its Cholesky factor with a random ordering.

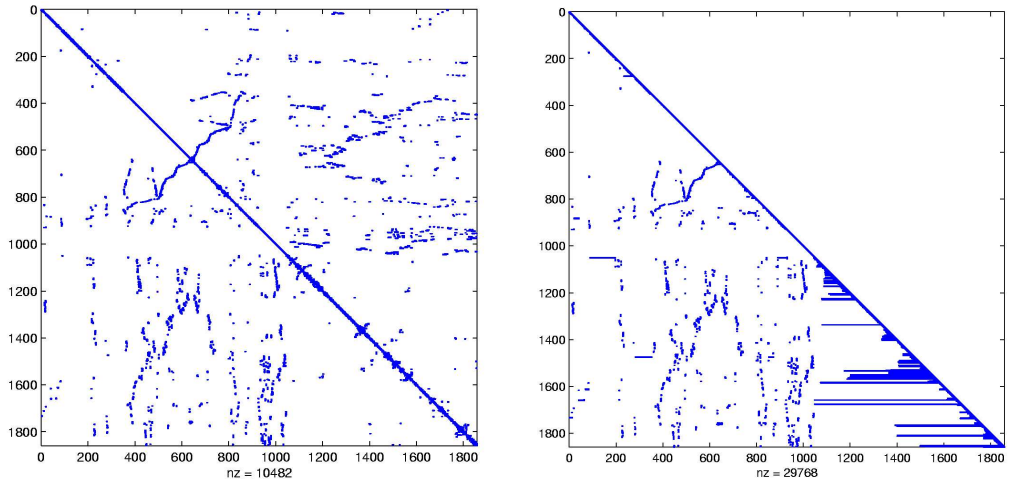

Figure 4.3: Sparsity pattern of testcase1 and its Cholesky factor with the ordering MINOLD.

The problems with fill-in are in the bottom right corner of the matrix. The figures of the MINOLD ordering show the problem with elimination graphs. If a node is eliminated, all his neighbours form a clique. Consequently, removing d_i edges can lead to adding $(d_i^2 - d_i)/2$ edges in the worst case. So handling nodes with large degree is very consuming for both time and memory. Therefore MINOLD only handles nodes below a certain degree, as you can see in Figure 4.3. How the problem with elimination graphs can be solved is described in the next chapter.

Chapter 5

Local ordering algorithms

In this chapter we will discuss several local ordering methods that are known for reducing fill-in. With an historical overview we highlight the most important improvements of the past decades that lead to efficient local orderings.

5.1 Minimizing fill-in; The Minimum Fill Ordering

Our objective is to find a labelling of the graph that results in the least fill-in for the (incomplete) Cholesky factor L. The most intuitive way to get such a labelling is to calculate for every node exactly how much fill-in will appear if the node is eliminated next. This is a local approach, since minimum fill-in per step does not guaranty total minimum fill-in. The amount of fill-in is equal to the number of possible edges between the adjacent nodes of the node we eliminate minus the number of edges already present between the adjacent nodes. Mathematically this means

$$score(v_i) = (d_i^2 - d_i)/2 - |\Upsilon| \tag{5.1}$$

with

$$\Upsilon = \{e_{j\ell} \in E \mid v_\ell \in Adj(v_i) \wedge v_j \in Adj(v_i)\}$$

The method for which the fill-in is explicitly calculated is called *Minimum Deficiency ordering* or *Minimum Fill ordering* (MF). This method is very time-consuming, for comparisons have to be made for every possible pair $v_j, v_\ell \in Adj(v_i)$ during every update. However, if we neglect Υ, we get an upper bound for the amount of fill-in, since . This upper bound (ub_{score}) is

$$score(v_i) = (d_i^2 - d_i)/2 - |\Upsilon| \leq \frac{(d_i^2 - d_i)}{2} = ub_{score}.$$

Since this is a monotonic function for positive integers we can also use just d_i as an upper bound. The method that uses the degree as a score function is called *Minimum Degree* (MD).

5.2 Minimum Degree Ordering

The Minimum Degree ordering algorithm has been a very popular ordering algorithm for over 30 years. There have also been multiple enhancements to speed up the algorithm ([10]), which will be discussed later in this chapter. The algorithm simply chooses a node v_i in the graph for which d_i is the smallest degree, and eliminates it from the elimination graph. Next, the elimination graph is updated and a new node is chosen. This procedure is repeated until all nodes are eliminated. This version of MD is already much faster than MF, but the elimination graphs still have a big drawback. In the previous section we discussed that the upper bound of fill-in is calculated by $(d_i^2 - d_i)/2$, and that eliminating a node could

result (in the worst case scenario) into removing d_i edges and adding $(d_i^2 - d_i)/2$. This means that during the first part of the algorithm the number of edges can increase dramatically, which leads to unacceptable high use of memory. A much more efficient way to model the elimination process is to use *quotient graphs*.

Quotient graphs consist of two kinds of nodes: unlabelled nodes (also referred to as supernodes) and eliminated nodes (also referred to as elements). When a node should be labelled next it becomes an eliminated node, which remains in the graph. Two unlabelled nodes are adjacent if they are connected with an edge or if they can reach each other through their eliminated neighbours. This way, no extra edges will be formed, so there will be no need for extra memory.

The initial quotient graph $F^0 = (V^0, \bar{V}^0, E^0, \bar{E}^0)$ is the same as the initial elimination graph, because $V^0 = V$ and $E^0 = E$, and $\bar{V}^0$ and $\bar{E}^0$ are empty. V^k is the set of supernodes and $\bar{V}^k$ is the set of eliminated nodes after step k. E^k is the set of edges $V^k \times V^k$ and $\bar{E}^k$ is the set of edges $V^k \times \bar{V}^k$. For each node, there are two extra interesting sets we should distinguish: all adjacent supernodes to node v_i ($\mathcal{A}_i$), and all adjacent eliminated nodes to node v_i ($\mathcal{E}_i$). The basic MD algorithm with quotient graphs is given in Algorithm 13.

Algorithm 13 MD with quotient graphs

1: $V = \{v_1, \ldots, v_n\},\ N = |V|$
2: $E = \{e_1, \ldots, e_m\},\ M = |E|$
3: **for** $i = 1 \ldots M$ **do**
4: Pick $e_i = (v_j, v_k).\ \mathcal{A}_j = \mathcal{A}_j \cup v_k,\ \mathcal{A}_k = \mathcal{A}_k \cup v_j.$
5: **end for**
6: **for** $i = 1 \ldots N$ **do**
7: $d_i = |\mathcal{A}_i|$
8: **end for**
9: **while** $V \neq \emptyset$ **do**
10: Pick v_p with $d_p \leq d_j, v_j \in V$
11: $\mathcal{A}_p = (\mathcal{A}_p \cup \bigcup_{v_j \in \mathcal{E}_p} \mathcal{A}_j) \backslash v_p$
12: **for** $v_j \in \mathcal{A}_p$ **do**
13: $\mathcal{A}_j = \mathcal{A}_j \backslash v_p$
14: $\mathcal{E}_j = \mathcal{E}_k \cup v_p \cup \mathcal{E}_p$
15: $d_j = |\mathcal{A}_j \cup \bigcup_{v_k \in \mathcal{E}_j} \mathcal{A}_k|$
16: **end for**
17: $V = V \backslash v_p$
18: **end while**

A visual example of quotient graphs is given later in this section in Figure 5.1. In addition to the use of quotient graphs several methods have been thought of during the years to improve the runtime of MD. These include mass elimination, graph compression, incomplete degree update, remove redundant edges, element absorption, multiple elimination, tie-breaking pre-ordering, external degrees and approximate degrees. We will discuss these enhancements next.

5.2.1 Mass elimination

George and McIntyre [11] observed that when a node is eliminated there is often a subset of nodes that can be eliminated in the same elimination step. If v_i is eliminated in step k and we look at the subset

$$U = \{v_j \in Adj(v_i)_{F^k} | (d_i)_{F^{k-1}} = (d_j)_{F^{k-1}-1}\}, \tag{5.2}$$

then all elements of U can be eliminated right after v_i. These nodes appeared to be *indistinguishable* with respect to v_i. Two nodes v_i and $v_j \in V \backslash X$ are indistinguishable if

$$Reach(v_i, X) \cup \{v_i\} = Reach(v_j, X) \cup \{v_j\}. \tag{5.3}$$

In other words, a set of indistinguishable nodes form a clique. Since removing one node from a clique does not generate any fill-in at all (all adjacent nodes are also adjacent to one another), all these nodes can be eliminated right after each other in any order. If we merge indistinguishable nodes in one node, which we will call a *supernode*, we encounter two advantages: we eliminate multiple nodes in one step, and we have less nodes for which we need to calculate new degrees. Mass elimination means that after eliminating a node we check its adjacency set for indistinguishable nodes, and each couple nodes are merged if they are indistinguishable. The identification of supernodes can be done using the hash function

$$hash_i = \sum \mathcal{A}_i + \sum \mathcal{E}_i \mod N. \tag{5.4}$$

If two nodes have the same hash value their adjacency list should be compared, and if they are the same they should be merged. We choose one node as the source and one as the target. The source node is removed from the graph and becomes a child of the target node, which we will write as $ch_{target} = ch_{target} \cup v_{source}$. We also define the cardinality of a node v_i as

$$|v_i| = 1 + |ch_i|. \tag{5.5}$$

The MD algorithm can be expanded with the supernode detection after the degree update

Algorithm 14 Hash-function initialization(v_i) (in the second for loop)

$hash_i = \sum \mathcal{A}_i + \sum \mathcal{E}_i \mod N$
$H(hash_i) = H(hash_i) \cup v_i$

Algorithm 15 Supernode detection (after the third for loop)

```
for j = 0...N-1 do
  if |H(j)| > 1 then
    for each v_i ∈ H(j) do
      for each v_k ∈ H(j), v_k ≠ v_i do
        if A_i ∪ v_i == A_k ∪ v_k then
          ch_i = ch_i ∪ v_k ∪ ch_k
          V = V\v_k, H(j) = H(j)\v_k
          A_k = ∅, E_k = ∅
        end if
      end for
    end for
  end if
end for
for j = 0...N-1 do
  H(j) = ∅
end for
```

Since N is very large for matrices that cause memory problems, the chance that we have a hash collision for two neighbouring nodes is very small, and since we know they are neighbours and have at least one element in common, the amount of extra fill-in generated by a bad collision is not that big. However, skipping the second-if statement of Algorithm 15 will save quite some computing time.

5.2.2 Graph compression

In addition to mass elimination we look for indistinguishable nodes prior to the elimination process. This addition works especially good for discretized problems with more then one solution component per grid point. Because this is not the case for our problem we will not use compression in our implementation.

5.2.3 Incomplete degree update

Some nodes have high degree, it is not their turn to be eliminated by far, so we would like to skip their degree updating, but how does the algorithm know when they come back in the field? The solution is the *outmatching* of nodes. A node v_j is outmatched by v_i if

$$Reach(v_i) \cup \{v_i\} \subseteqq Reach(v_j) \cup \{v_j\}. \tag{5.6}$$

It is clear that v_j will never have a lower degree then v_i, so we don't have to update the degree of v_j until v_i is eliminated.

5.2.4 Remove redundant edges

Some edges in the quotient graph are redundant, and therefore it would improve the performance of the algorithm if they were removed. Take three nodes, v_i, v_j and v_k, which are fully connected with each other. If v_k is eliminated, v_i and v_j are connected through v_k and with a direct connection. This direct connection is redundant and can be removed. So instead of line 13 of Algorithm 13 we get

$$\mathcal{A}_j = (\mathcal{A}_j \backslash \mathcal{A}_p) \backslash v_p$$

5.2.5 Element absorption

Eliminated nodes are used to represent a clique between some of the nodes in V. But if all elements remain part of the graph it is expensive to find all the reachable sets for those nodes. It is more efficient to absorb adjacent elements, for the new adjacent element span the same adjacency list as the union of the old adjacency list, so no information is lost. This method is called *element absorption*. We can replace line 14 of Algorithm 13 with

$$\mathcal{E}_j = (\mathcal{E}_j \backslash \mathcal{E}_p) \cup v_p$$

An extension of element absorption is *aggressive element absorption*, for which all elements v_e, including the ones not directly adjacent to v_p, for which holds that

$$\mathcal{A}_e \backslash \mathcal{A}_p = \emptyset, \tag{5.7}$$

are merged into v_p.

As an example, look in Figure 5.1 at the simple circuit from the previous chapter, shown as an quotient graph. The figure also contains the quotient graphs after eliminating the nodes in the current order of labelling, using mass elimination, (aggressive) element absorption, and removal of redundant edges. The large open circles represent elements.

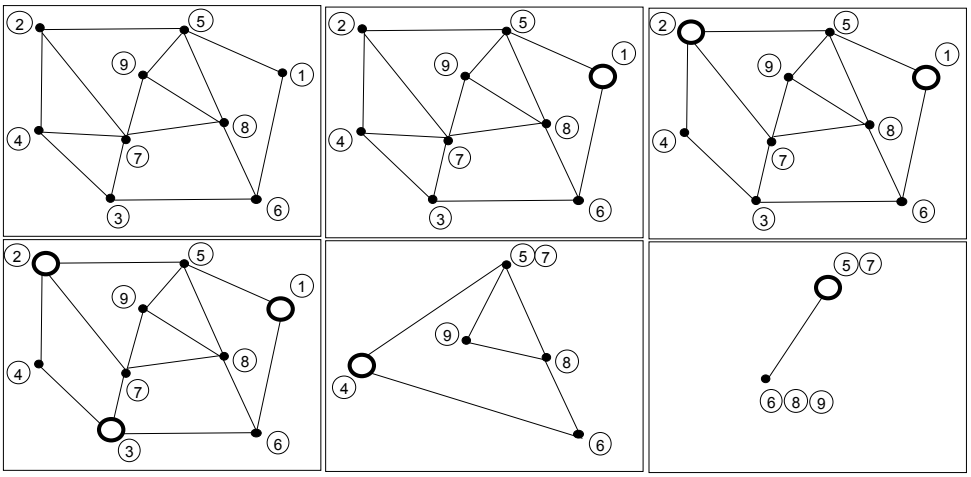

Figure 5.1: The quotient graphs $F^0 \dots F^5$ of a simple circuit.

In the third picture, after the elimination of v_2, the edge between nodes v_4 and v_7 is redundant and should be removed. When v_4 is eliminated the adjacent elements (nodes v_2 and v_3) must be absorbed. However, although v_1 is not adjacent to node v_4, it does not give any additional information and should therefore also be absorbed. This is an example of aggressive element absorption. Nodes v_5, v_7 and v_8 form a clique now, but we only check for supernodes in $\mathcal{A}_4$, so we merge nodes v_5 and v_7. After eliminating node v_5 the remaining quotient graph is a clique, this will become one supernode and is the last node that is eliminated.

5.2.6 Tie-breaking pre-ordering

Large graphs imply many nodes with the same degree. Since the first node to be eliminated is the node of the lowest degree with the lowest initial labelling it might be of importance how the matrix is ordered prior to the elimination process. Two possible pre-ordering are random ordering and the Reversed Cuthill-McKee ordering, which will be discussed later in this chapter.

5.2.7 External degrees

Liu came up with the idea to use external degrees instead of true degrees. External degrees of supernodes are equal to the true degree minus the number of nodes merged into the supernode. Since supernodes form a clique, no internal fill-in will be created, so external degrees form a tighter bound than true degrees. Therefore, the degree update should be

$$d_j = |\mathcal{A}_j \backslash v_p| + |\bigcup_{v_k \in \mathcal{E}_j} \mathcal{A}_k \backslash v_p|. \tag{5.8}$$

and we should add to the supernode routine in Algorithm 15

$$d_j = d_j - |v_k|. \tag{5.9}$$

5.2.8 Multiple elimination

If we eliminate a node v_i we need to calculate the degree of its neighbours. But if we find another node with the same minimal degree to eliminate that is not a neighbour of v_i we can postpone updating the degrees. So, in general we do

$G_i = (V_i, E_i)$
$W = V_i$
Eliminate $v_j \in W$ with $score(v_j) \leq score(v_k),\ v_k \in W$.
$W = W\ (v_j \bigcup Adj(v_j))$
while ($\exists v_k \in W$ with $score(v_k) = score(v_j)$) **do**
  Eliminate v_k
  $W = W\ (v_k \bigcup Adj(v_k))$
end while

This enhancement is known as multiple elimination, and the method, very popular between 1985 and 2000, is known as *Multiple Minimum Degree* (MMD). Figure 5.2 shows the multiple minimum degree ordering of the matrix testcase1 using the Matlab-ordering `symmmd`. This figure shows that using the minimum degree ordering leads to significant reduction of fill-in.

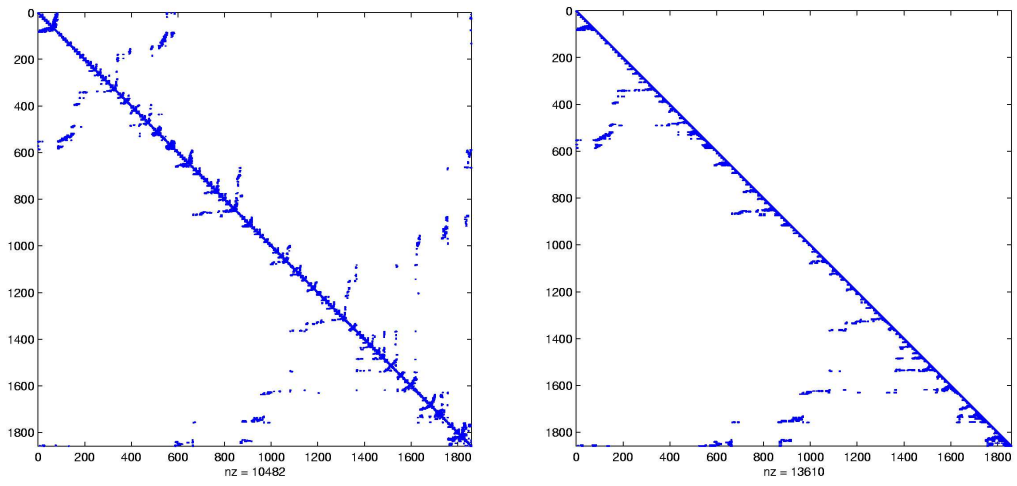

Figure 5.2: Sparsity pattern of testcase1 and its Cholesky factor ordered with symmmd.

5.2.9 Approximate degrees

Since the term degree is already a somehow loose upper bound for the amount of fill-in, it is not unlikely to think that an approximation of the degree will give similar results but a better performance. This is what Amestoy, Davis and Duff thought when they came up with the *Approximate Minimum Degree ordering* algorithm (AMD) [1]. Instead of calculating the exact degree d_i the approximate degree d_i^{-k} is calculated. We take p to be the node that is eliminated next and $|v_i|$ to be the weight of the current supernode. This approximate degree is the minimum of three upper bounds for the degree. The first upper bound, $ub_1 = n - k$, is quite obvious. It is equal to the number of nodes left in the graph. The second bound, $ub_2 = d_i^{k-1} + |\mathcal{A}_p \backslash |v_i||$, is equal to the old degree plus the worst case fill-in. Clearly, if d_i^{-k-1} is an upper bound, ub_2 is also an upper bound. For the third bound, $ub_3 = |\mathcal{A}_i \backslash |v_i|| + |\mathcal{A}_p \backslash |v_i|| + \sum_{e \in \mathcal{E}_i \backslash p} |\mathcal{A}_e \backslash \mathcal{A}_p|$, we calculate the number of directly adjacent nodes, the number of adjacent nodes through the new element, and the number of nodes through other elements that are not present in the current element. All nodes adjacent to the current node are counted, so ub_3 is an upper bound. This gives the approximate degree

$$d_i^{-k} = \min \begin{cases} n - k, \\ d_i^{-k-1} + |\mathcal{A}_p \backslash |v_i||, \\ |\mathcal{A}_i \backslash |v_i|| + |\mathcal{A}_p \backslash |v_i|| + \sum_{e \in \mathcal{E}_i \backslash p} |\mathcal{A}_e \backslash \mathcal{A}_p|. \end{cases} \tag{5.10}$$

The calculation of the term $|\mathcal{A}_e \backslash \mathcal{A}_p|$ can be done in an efficient way. Every element has a certain w-value, that is -1 at the start of each iteration step. After the formation of $\mathcal{A}_i$ all nodes in this set are visited (v_j),

and for all these neighbours all adjacent elements are visited. If such an element e has a w_e of -1, the value is set to $w_e = |\mathcal{A}_e|$. Then $w_e = w_e - |v_j|$. After all neighbouring nodes are visited we have

$$w_e = |\mathcal{A}_e \backslash \mathcal{A}_p| \tag{5.11}$$

Clearly, $w_p = 0$. If $w_e = -1$ the element is not visited, so we will not need it for the degree update.

Theorem 5.2.1. *The approximate degree d_i^{-k} is equal to d_i if —$\mathcal{E}_i| \leq 2$.*

Proof. Initially, if —$\mathcal{E}_i| = 0$ for all i, $d_i^0 = d_i$. Assume we update the degree of a node v_i with $|\mathcal{E}_i| = 1$. Then the only possible adjacent element is the current eliminated node v_p and $\sum_{e \in \mathcal{E}_i \backslash p} |\mathcal{A}_e \backslash \mathcal{A}_p| = 0$, so $d_i^{-k} = d_i$. Assume now we update the degree of a node v_i with $|\mathcal{E}_i| = 2$. Then there is one additional element besides v_p in $\mathcal{E}_i$, say v_q. The degree of v_i is equal to the number of nodes adjacent to it plus the number of nodes that can be reached through v_p plus the number of nodes that can be reached through v_q without the nodes adjacent to v_p. This is exactly d_i^{-k}, so $d_i^{-k} = d_i$. □

The following example (Figure 5.3) shows that the equation $d_i^{-k} = d_i$ is no longer always valid if $|\mathcal{E}_i| > 2$.

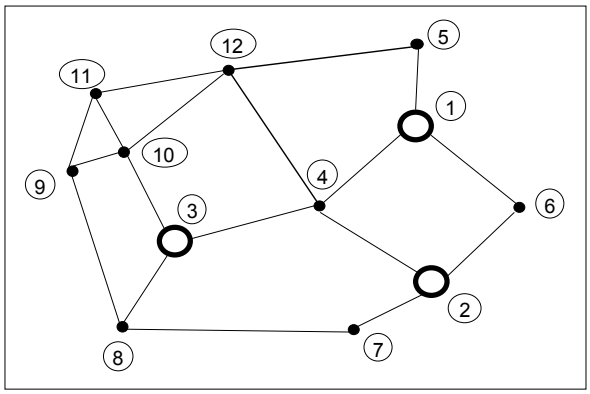

Figure 5.3: Example of a quotient graph in which $d_4^{-k} > d_4$.

Node v_3 is the node that is just eliminated. If we want to calculate d_4^{-3} we need d_4^{-2}, say $d_4^{-2} = 5$ (the minimum value possible). We find that $d_4 = 6$, since the adjacent nodes are v_5, v_6, v_7, v_8, v_{10} and v_{12}. $ub_1 = 12 - 3 = 9$ and $ub_2 = 5 + 2 = 7$. For ub_3 we need all adjacent nodes (v_{12}), all nodes reachable through v_3 (v_8 and v_{10}), all nodes reachable trough v_1 that are not adjacent to v_3 (v_5 and v_6) and all nodes reachable trough v_2 that are not adjacent to v_3 (v_6 and v_7). Note that node v_6 is counted twice. So $ub_3 = 1 + 2 + 2 + 2 = 7$ and $d_4^{-3} = \min(9, 7, 7) = 7 > 6 = d_4$.

5.2.10 MADAND(AMD)

AMD is considerably faster than MMD, and a combination is not very suitable due to the method of calculating the approximate degree ([1]). For the same reason we do not use incomplete degree update. We had already ruled out graph compression. With the use of the other enhancements we get our MADAND(AMD) algorithm. The pseudo code of this algorithm can be found in the appendix.

5.3 Approximate Minimum Deficiency

The AMD algorithm handles the problem of updating degrees very well, but there might still be room for improvement concerning the score function. Rothberg and Eisenstat ([25]) recognized the degree as an

upper bound of fill-in, which could be fit tighter. Take $\mathcal{A}_e$ for some eliminated node e. Then all supernodes in $\mathcal{A}_e$ form a clique, so there will no new edges occur between these nodes. Now take a supernode v_i and assume $\hat{e}$ is the last eliminated neighbour of v_i. Then $c_i = |\mathcal{A}_{\hat{e}} \backslash |v_i||$ is the number of nodes in the clique (without the current node v_i), so $(c_i^2 - c_i)/2$ is the number of edges already present in the clique that we can subtract from the upper bound. This gives a new score function:

$$score(v_i) = (d_i^2 - d_i)/2 - (c_i^2 - c_i)/2. \tag{5.12}$$

The method using this score function is called *Approximate Minimum Local Fill ordering* (AMF). In the same paper they also discussed the *Approximate Minimum Mean Local Fill ordering* (AMMF) with score function

$$score(v_i) = \frac{score_{AMF}}{|v_i|}, \tag{5.13}$$

and *Approximate Minimum Increase in Neighbour Degree ordering* (AMIND) with score function

$$score(v_i) = score_{AMF} - (d_i \times |v_i|). \tag{5.14}$$

The implementations are called MADAND(AMF), MADAND(AMMF) and MADAND(AMIND), respectively.

5.4 Minimum degree with drop tolerance prediction

Eliminating nodes from the graph can result into new edges which represent fill-in in the matrix. However, during the creation of the L matrix we neglect some fill-in using the drop tolerance. So perhaps if we can predict which edge will be neglected we can save some computing time.

Initially, each edge has a value (γ_e), the negative value of the capacitance. Recall that if node v_i is connected to the nodes v_k and v_j with no connection between the last two, an edge is created between v_k and v_j if v_i is labelled first. Assume we know $a[i,k] = -g_{ik},\ a[i,j] = -g_{ij},\ a[k,j] = 0$ and the node to be eliminated is node i. Then $a[i,k] = -g_{ik}/g_{ii}$ and $a[i,j] = -g_{ij}/g_{ii}$, and we have $a[k,j] = -g_{ij}g_{ik}/g_{ii}$.

Now assume i or j is the first node to eliminate. Then if $\gamma_{ij} \ll g_{ii}$ AND $\gamma_{ij} \ll g_{jj}$, the edge between i and j will not likely create fill-in, so removing the edge from the graph may have little influence on the total fill-in of the matrix, while less edges are present in the graph.

This clearly only holds initially, since eliminating nodes not only creates new edges, but also adjusts the values of the old edges. However, if we assume that these adjustments are small and if we only remove edges for which hold

$$|\frac{\gamma_{ij}}{\min(\gamma_{ii}, \gamma_{jj})}| < \eta, \tag{5.15}$$

for some small $\eta > 0$, the amount of fill-in should not increase substantially. The drop tolerance prediction can be used for every MADAND variant.

5.5 Reverse Cuthill-McKee ordering

The reverse Cuthill-McKee ordering turned out to be superior over the original Cuthill-McKee ordering from 1969. The principle of the algorithm is to find an appropriate starting node and number all his neighbours in increasing order of degree, and then reverse the ordering. In [9] is described how the starting node is found. Figure 5.4 shows the Reverse Cuthill-McKee ordering of the matrix testcase1 using the Matlab-ordering `symrcm`. This Figure shows that this method does not really reduce the fill-in for this matrix.

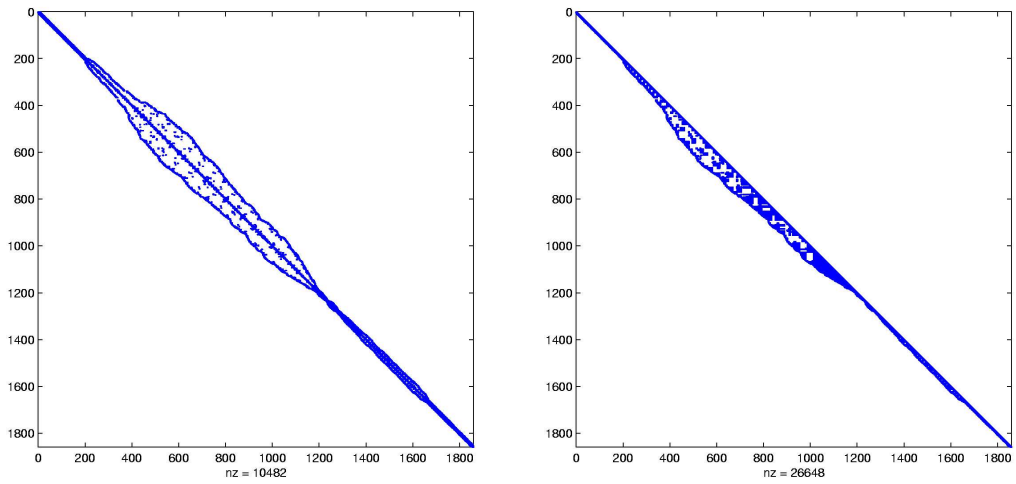

Figure 5.4: Sparsity pattern of testcase1 and its Cholesky factor ordered with symrcm.

Chapter 6

Global ordering algorithms

Global ordering methods try to separate the nodes so their interaction (and thus the occurrence of fill-in) is minimal. The aim is to find a *separator* (also called *border*) that cuts the graph into multiple disjoint partitions. This is usually done either recursive as in the Nested Dissection ordering, or at once as in the Multisection ordering. We will discuss both methods in this chapter.

6.1 Nested Dissection ordering

The aim of the Nested Dissection ordering to find a separator in the form of a bisector, a set of vertices whose removal would cut the graph into two parts, approximately of the same length. These parts can be cut again and again, until the parts have an appropriate length. A matrix ordered with a Nested Dissection can be used for parallel computing, since we can calculate each part on a different processor, and only the columns of the separators depend on the parts they separate. This is a hierarchical structure

$$PGP^T = \begin{pmatrix} A & 0 & S_A^T \\ 0 & B & S_B^T \\ S_A & S_B & S \end{pmatrix} = \begin{pmatrix} AA & 0 & S_{AA}^T & 0 & 0 & 0 & S_A^T \\ 0 & AB & S_{AB}^T & 0 & 0 & 0 & \vdots \\ S_{AA} & S_{AB} & SA & 0 & 0 & 0 & S_A^T \\ 0 & 0 & 0 & BA & 0 & S_{BA}^T & S_B^T \\ 0 & 0 & 0 & 0 & BB & S_{BB}^T & \vdots \\ 0 & 0 & 0 & S_{BA} & S_{BB} & SB & S_B^T \\ S_A & \cdots & S_A & S_B & \cdots & S_B & S \end{pmatrix}. \tag{6.1}$$

Packages such as METIS [18] and CHACO [15] can calculate such orderings, as shown in Figure 6.1. In addition to the parallel structure of the matrix L the ordering method also reduces the fill-in significantly, but not as good as the MD and its variants.

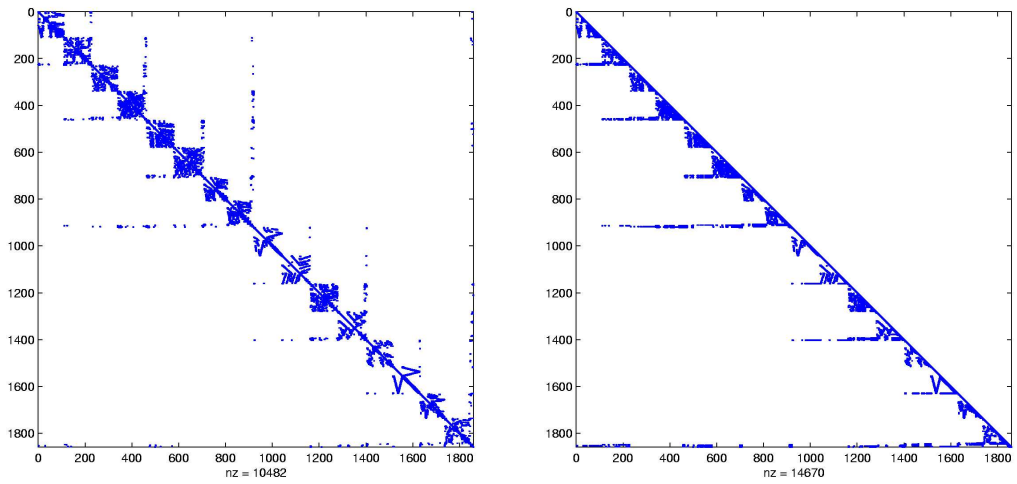

Figure 6.1: Sparsity pattern of testcase1 and its Cholesky factor ordered with METIS (oemetis).

6.2 Multisection ordering

Multisection ordering uses, unlike the Nested Dissection ordering, only one separator. The graph is cut into more than two parts, say M parts. These M parts can be processed parallel, with finally a integral process using the separator.

$$PGP^T = \begin{pmatrix} A & 0 & \cdots & 0 & S_A^T \\ 0 & B & \ddots & \vdots & S_B^T \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & M & S_M^T \\ S_A & S_B & \cdots & S_M & S \end{pmatrix}. \tag{6.2}$$

An example of a Multisection ordering is the ordering of Zecevic and Siljak ([31]), which will be discussed in section 6.4. In Figure 6.2 the results of the algorithm are shown.

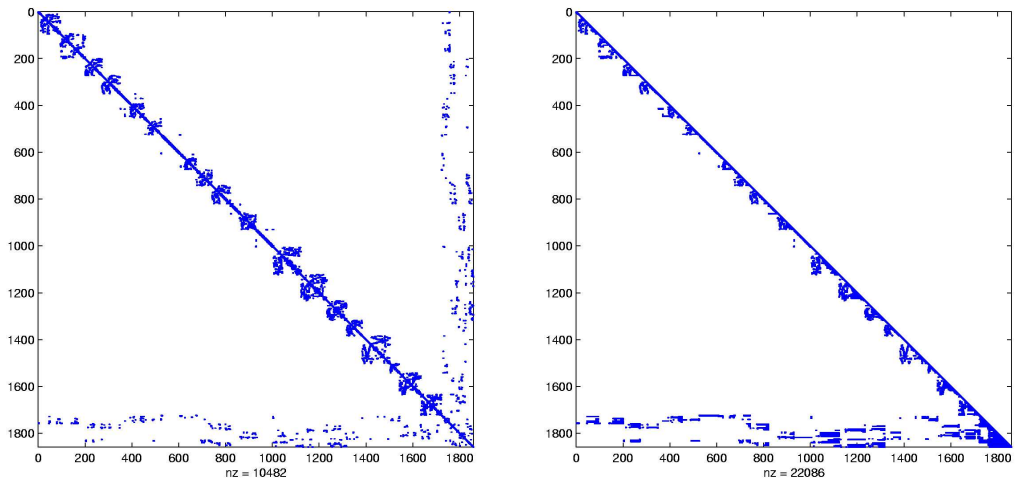

Figure 6.2: Sparsity pattern of testcase1 and its Cholesky factor ordered with siljak8.m.

6.3 Combining local and global ordering methods

For most of our test-matrices the local ordering will lead to less fill-in compared to the global orderings. But we need to use global orderings to parallel solving methods, so it is of course possible to use the local ordering as a pre- or a post-ordering. Which should we use first? When we use the local first we might consider impossible fill-in, because most nodes will be separated, and when we use the global method first we neglect the fill-in produced in the separator. Liu ([20]) suggested to connect the MD and its variants with the different dissection methods. His constrained version of the minimal degree algorithm is simply the MD, but adding the separator to the graph. Those nodes can not be chosen to be eliminated, but they will appear in the adjacency lists. This way the fill-in in the separator will also be minimized. We will refer to the constrained version of MADAND as MADAND(CON).

6.4 Basics of Dissection Orderings

The basics for most Nested Dissection and Multisection ordering algorithms are the same. They consist of three steps, but before we discuss these steps we need to give some definitions.

Definition 6.4.1. A *matching* is a set of edges, no two of which are incident on the same node.

Definition 6.4.2. A matching is *maximal* if the edges that are not in the matching contain at least one node that has been matched.

An efficient greedy algorithm to find a maximal matching is *random matching*, which walks through the nodes and matches all unmatched nodes. A graph can be *coarsened* using a random matching by merging the nodes in the matching and giving the nodes and edges a weight number. The *Kernighan-Lin algorithm* [19] uses a certain initial configuration of the graph, a bisection, and then exchanges two nodes, one from each part, such that the new edge cut (the number of edges connecting both parts) is minimized. The tree basic steps are:

1. Coarsen the graph with a maximal matching, for a coarsened graph needs far less operations with respect to the original graph.

2. Find a certain initial configuration and swap nodes using a Kernighan-Lin alike algorithm.

3. While uncoarsening the graph try to improve the configuration. Then label the nodes by partition, with the separator at the end.

In this thesis we especially focus on the second step.

6.5 Domain decomposition

In this subsection we will highlight two ordering algorithms based on domain decomposition. The first is constructed by Zecevic and Siljak ([31]) and is a Multisection ordering algorithm, and the second is constructed by Ashcraft and Liu ([3]) and is a Nested Dissection algorithm. The main step is to form blocks of adjacent nodes, separated by a small set of nodes, which we already defined as separator. Obviously, the intersection of two blocks is always empty. It is important to keep track of all adjacent blocks, this will be notated with $\mathcal{NB}_i$ for node v_i of $\mathcal{NB}_{b_t}$ for block b_t. The method to form the blocks is described below:

- Choose ω_{max}, ω_{min} and d_{max}.
- Put all nodes v_i with $d_i \geq d_{max}$ in the separator Ψ.
- Pick a random node in $V \backslash \Psi$ and grow it into a block in a breath first fashion.
- If the size of this block is ω_{max} it is full and all adjacent nodes go into Ψ.

6.5.1 Zecevic and Siljak

After the blocks are formed, Z&S continue with:

- Determine for every node v_i the status of its neigbours. q_i is the number of neighbours of v_i that are contained in Ψ and S_i is the total size of all adjacent blocks.
- As long as min(S_i) $\leq |\Psi|$, take v_i out of Ψ. This may lead to several (small) partitions.
- If min(S_i) $> |\Psi|$ we start the algorithm over again, but only using Ψ, and before starting we already create the fill-in that may occur during the factorization process.

This last step turned out to be a bottleneck. Essentially they perform a minimum fill algorithm, which we already pointed out as very slow in the previous chapter. In addition, multisection gives more connection between nodes in the separator, which may lead to more fill-in. Multisection appears to be useful for specific problems like rectangular grids of the form $h \times k$ with $h \gg k$, but not for general unstructured matrices.

Zecevic and Siljak handled their bottleneck by adjusting their algorithm so it became a Nested Dissection ordering [32]. Instead of multiple blocks the growing procedure continued until there are only few blocks left. These blocks can be dissected again, and this results into a nested algorithm, which we will call $impZ\&S$.

6.5.2 Ashcraft and Liu

Ashcraft and Liu came up with a block version of the Kernigan-Linn algorithm:

- For all blocks with a number of nodes less than ω_{min}, put every node in Ψ and destroy the blocks.
- Check for all nodes in Ψ that are adjacent to only one block if there is room for them. If so, put them in the block (lowest degree first).
- Combine all adjacent nodes in Ψ that have no adjacent block in common.
- Combine all adjacent sets of nodes that cut exactly the same block. These two steps make blocks of the separator-nodes in a way such that two different coloured blocks will always be separated.

- We want two parts, W and B, that are balanced and $G \subseteq \Psi$ to be as small as possible. Therefore we use $\gamma(G, W, B) = |G|(1 + \alpha \frac{\max(|B|,|W|)}{\min(|B|,|W|)})$ as a cost function.
- Begin with an initial B, W and G and swap all partitions one by one, the one that leads to the smallest G first. Choose the configuration with the smallest cost en continue, until we find no better configuration.
- All three sets form a separate partition. The partitions of B and W can be dissected again, which will lead to the nested structure.

It is also possible to improve the separator with the use of the *Dulmage-Mendelsohn decomposition*. This method is based on the following scenario: assume there are two partitions, $part1$ and $part2$, separated by sep. Say there are sets of nodes, one in sep , $sep(v_i)$, and one in $part1$, $part1(v_j)$, such that $part1(v_j)$ contains all nodes in $part1$ adjacent to the nodes $sep(v_i)$. If $part1(v_j)$ is put in the separator and $sep(v_i)$ in the other partition, we still have a legal separator. In addition, if $|sep(v_i)| > |part1(v_j)|$, the new separator is smaller then the old one. Clearly, the new configuration should also result into a smaller value of the cost function, otherwise the balance of the blocks could become disturbed. The complete pseudo code of the border-improving method using the Dulmage-Mendelsohn decomposition can be found in the appendix, as well as an example on which the algorithm is applied.

6.5.3 MANDAND(NEST)

The $impZ\&S$-algorithm appears to be a simple version of the $A\&L$-algorithm. However, the blocks are merged with a local search algorithm, and there is no room for improvement after a bad (local) choice. Therefore the implemented nested dissection for the MADAND-software is the algorithm of Ashcraft and Liu, which we will refer to as MADAND(NEST).

Chapter 7

Parallel Solution methods

A Nested Dissection ordering or Multisection ordering gives a structured matrix. In this chapter we discuss how we can benefit from these structures.

7.1 Substitution in an iteration step

For the (incomplete) Cholesky decomposition we can solve our system with the use of Forward-Backward substitution. If we have a structured matrix we are able to do this step on multiple processors. If the matrix G has a block bordered form, a result of a Multisection ordering, parallel processing is possible. The matrix below has a block bordered structure

$$\hat{G} = \begin{pmatrix} G_{11} & 0 & \cdots & 0 & G_{1m} \\ 0 & G_{22} & \ddots & \vdots & G_{2m} \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & G_{m-1m-1} & \vdots \\ G_{m1} & G_{m2} & \cdots & \cdots & G_{mm} \end{pmatrix}. \tag{7.1}$$

One method of solving this system is using the *Schur complement* S with

$$S = G_{mm} - \sum_{i=1}^{m-1} G_{mi} G_{ii}^{-1} G_{im}. \tag{7.2}$$

Using the Schur complement we can construct algorithm 16. With a little adjustment, the Schur complement can also be used for matrices with a nested structure. For two levels, this structure looks like

$$\hat{G} = \begin{pmatrix} G_{11} & 0 & G_{121} & 0 & 0 & 0 & G_{15} \\ 0 & G_{22} & G_{122} & 0 & 0 & 0 & G_{25} \\ G_{112} & G_{212} & G_{12} & 0 & 0 & 0 & G_{125} \\ 0 & 0 & 0 & G_{33} & 0 & G_{343} & G_{35} \\ 0 & 0 & 0 & 0 & G_{44} & G_{344} & G_{45} \\ 0 & 0 & 0 & G_{334} & G_{434} & G_{34} & G_{345} \\ G_{51} & G_{52} & G_{512} & G_{53} & G_{54} & G_{534} & G_{55} \end{pmatrix}. \tag{7.3}$$

We can calculate the Schur complement of the separate blocks 1-2 and 3-4 and then use the results to calculate the Schur complement of the entire system. Clearly, this is possible for multiple levels.

Algorithm 16 Parallel Forward and Backward substitution

for $i = 1, \ldots, m-1$ **do**
  $G_{ii} = L_{ii} D_{ii} L_{ii}^T$
  $L_{mi} = G_{mi} (L_{ii}^T)^{-1} D_{ii}^{-1}$
  $y_i = L_{ii}^{-1} b_i$
  $S^{(i)} = L_{mi} D_{ii} L_{mi}^T$
  $z^{(i)} = L_{mi} y_i$
end for
$S = G_{mm} - \sum_{i=1}^{m-1} S^{(i)}$
$y_m = b_m - \sum_{i=1}^{m-1} z^{(i)}$
Solve $S x_m = y_m$
for $i = 1, \ldots, m-1$ **do**
  $x_i = (L_{ii}^T)^{-1} D_{ii}^{-1} (y_i - D_{ii} L_{mi}^T x_m)$
end for

7.2 Topology of the preconditioner matrix

One of the additional research questions was to think of a manner to recalculate L fast if some nonzero values of G are changed in other nonzero values. Almost all ordering algorithms are based on the topology of the matrix, so changing branch values will not change the order of the nodes. This is important, because we only have to determine the order once, this order is still valid after adjustments of branch-values.

In the construction of L we already noticed that it goes in a forward fashion: Node v_1 adjusts matrix entries for nodes v_2 to v_n, Node v_2 adjusts matrix entries for nodes v_3 to v_n etc. Assume we have calculated L and D and say we adjust branches $e_1 \ldots e_m$ connecting nodes $v_1 \ldots v_k$ for which v_1 has the smallest label. Then for all nodes labelled before v_i nothing changes, so the decomposition of the nodes before v_i remains the same. This way we only need a new decomposition of the matrix entries starting at v_1.

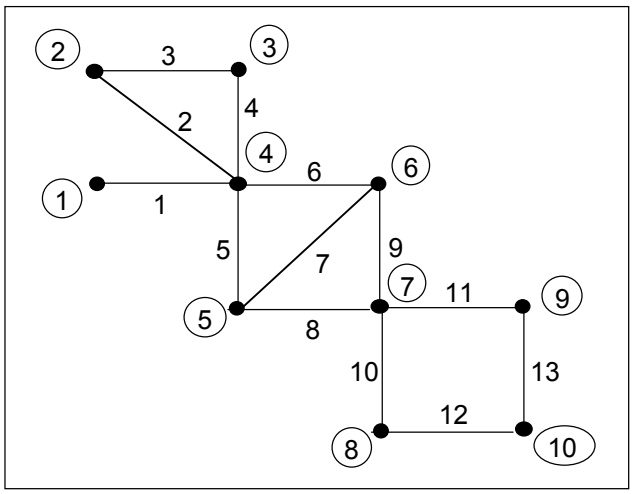

Figure 7.1: Small graph of the example in Section 7.1.

For a matrix that has a nested form, there is more room for improvement. Assume we adjust branch e_1 connecting nodes v_1 and v_2, v_1 labelled first. Then v_1 and v_2 are in the same block, or v_2 or both nodes are in a separator of the block. So we must make a new decomposition of the block v_1 is in, starting at v_1, and of all separators of this block. This is probably less work than in the normal case. As an example, look at Figure 7.1. The T matrix, with

$$T = L + D - I, \tag{7.4}$$

ordered with a minimum degree ordering of this graph is given below.

$$\begin{pmatrix}
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -0.6 & 5.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-0.5 & -0.4 & -1 & 11.5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -0.43478 & 17.826 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -0.52174 & -0.53902 & 13.69 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -0.44878 & -0.97239 & 21.465 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -0.46587 & 17.341 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -0.51246 & -0.29551 & 16.849 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.69199 & -0.98205 & 0.44695
\end{pmatrix}.$$

If the branch between node v_4 and v_5 would be changed in 40 the T matrix becomes

$$\begin{pmatrix}
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -0.6 & 5.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-0.5 & -0.4 & -1 & 46.5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -0.86022 & 20.591 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -0.12903 & -0.5906 & 14.043 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -0.38851 & -0.97732 & 21.478 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -0.46558 & 17.344 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -0.51214 & -0.29528 & 16.854 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.69188 & -0.98156 & 0.45695
\end{pmatrix}.$$

So, in column the columns v_1 to v_3 nothing changes, so we can skip 30% of the factorization. Now assume we use a new (nested) ordering (1,2,3,4,10,9,8,7,6,5). The T matrix is (with the original edges)

$$\begin{pmatrix}
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -0.6 & 5.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-0.5 & -0.4 & -1 & 11.5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 25 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -0.52 & 17.24 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -0.48 & -0.36195 & 13.981 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -0.63805 & -1 & 17 & 0 & 0 \\
0 & 0 & 0 & -0.52174 & 0 & 0 & 0 & -0.52941 & 14.105 & 0 \\
0 & 0 & 0 & -0.43478 & 0 & 0 & 0 & -0.47059 & -0.9815 & 0.47344
\end{pmatrix}.$$

Again, if the (original) branch between v_4 and v_5 would be changed in 40 the T matrix becomes

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.6 & 5.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.5 & -0.4 & -1 & 46.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 25 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.52 & 17.24 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.48 & -0.36195 & 13.981 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.63805 & -1 & 17 & 0 & 0 \\ 0 & 0 & 0 & -0.12903 & 0 & 0 & 0 & -0.52941 & 16.461 & 0 \\ 0 & 0 & 0 & -0.86022 & 0 & 0 & 0 & -0.47059 & -0.99609 & 0.49437 \end{pmatrix}.$$

Now only columns 4, 9 and 10 change, so we can skip 70% of the factorization. Clearly, this is only relevant if the preconditioner matrix is stored.

7.3 Efficient Dissection Verification

A Nested Dissection ordering algorithm can be rather complicated, so an efficient method to test its validity can be very useful. Assume a certain nested dissection is given, and assume all nodes have a certain partition, generated by the tree structure given in Figure 7.2.

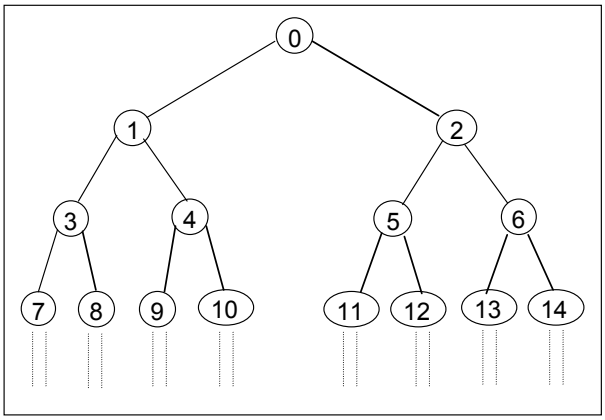

Figure 7.2: Tree structure of the partition numbers.

So the partitions have the following nested structure:

- $M + 1$ is the number of partitions, the first partition is 0.
- For the first $(M/2 - 1)$ partitions holds: partition k has two leaves, partition $2 * k + 1$ and partition $2 * k + 2$.
- Partitions $M/2$ to M have no leaves.
- A *subtree* τ_k is the union of partition k and the subtrees of partitions $2 * k + 1$ and $2 * k + 2$ if $k < M/2$, and the union of partition k and its leaves if $k \geq M/2$.

We traverse the tree depth-first, every time considering the left branch before the right one.

Theorem 7.3.1. *If the partitions of a nested structure are labelled as in Figure 7.2 and traversed as described above, a dissection is valid if and only if during the constrained ordering for every eliminated node v_p holds that*

$$part(v_p) \leq part(v_j), \forall v_j \in \mathcal{A}_p \tag{7.5}$$

Proof. $\Longrightarrow$ is trivial. $\Longleftarrow$ Consider the case that for every eliminated node v_p 7.5 holds. Assume that the dissection is not valid. Then there is a separator for which the two subtrees (τ_1 and τ_2, $\tau_1 < \tau_2$) are separated. Say node v_1 is part of subtree τ_1 and node v_2 is part of subtree τ_2, and take v_1 and v_2 connected. If v_1 is part of partition τ_1 we have a contradiction. So assume v_1 is part of partition τ_3 with $\tau_3 > \tau_1$. But then if all partitions in the subtree τ_1 except τ_1 itself are eliminated, there is a node v_3 with partition τ_1 such that it is connected with v_2, because the connection between v_1 and v_2 can not be removed, only passed on to another node. So we have a contradiction. This concludes the proof. □

With the use of this theorem we can simply check - during the local ordering - whether the dissection is valid or not.

Chapter 8

Results

There are two main steps of the solving method that we can manipulate and are therefore suitable for testing:

- Ordering, matrix decomposition, and fill-in reduction.
- Pre-Conjugate Gradient iterations and forward-backward substitution.

The analyzed testcases used are described in Tables 8.1 and 8.2.

Table 8.1: Description of all testcases processed with MATLAB.

Testcase	N	d
testcase1	1,858	4.64
testcase2	1,964	4.61
testcase3	10,574	5.00

Table 8.2: Description of all testcases processed with C++.

Testcase	N	d	N/H	$\bar{d}_H$
testcase4	291,306	3.40	0.32	5.54
testcase5	14,503,944	2.78	0.28	5.20
testcase6a	3,349,110	3.28	0.28	5.58
testcase6b	2,964,285	3.31	0.25	5.72
testcase7a	3,007,889	3.45	0.42	4.72
testcase7b	3,485,738	3.71	0.40	5.28

We have N to be the number of nodes, and $\bar{d}$ is the average degree. Before the matrix is generated we do a certain preprocessing step. All nodes with degree 3 or less are eliminated with an elimination graph using the local ordering. Note that elimination graphs are no worse than quotient graphs for nodes of degree three or lower. These low degree nodes are solved using a direct method. All other nodes go in a matrix which is solved using the PCG-method. Testcase1 to testcase3 are already pre-processed, the others are not, so for each case the approximate size of the matrix is given, since this depends on the ordering method. The approximate size of the matrix is H, and $\bar{d}_H$ is the average degree of the nodes in this matrix. However, the difference in sizes of the matrix between MINOLD and MADAND are less than 0.5 percent for any testcase.

Other notations that will be used often:

- MNZ is the number of nonzero elements of the Cholesky factor.
- WLNZ is the number of nonzero elements the busiest processor has to process. This is explained in section 8.1.2.
- PART is the current partition.
- NOP is the percentage nodes in the current partition.
- NZP is the percentage nonzero elements in the current part of the rows of Cholesky factor.
- OT is the ordering time.
- CT is the computing time.
- RT is the overall runtime.

8.1 Ordering, Matrix decomposition

8.1.1 One processor

The MADAND local ordering method, discussed in section 5.2.10, is implemented in C++ and the results are shown below.

Table 8.3: MNZ for several orderings.

Ordering	testcase1	testcase2	testcase3
Random Ordering	20,231	21,982	287,211
MINOLD	15,363	14,148	237,278
Matlab AMD	11,725	12,215	144,281
METIS oemetis	13,179	14,048	161,219
MADAND(AMD)	12,058	12,647	146,200
MADAND(AMD) (mass elim.)	11,735	12,221	145,611
MADAND(AMF) (mass elim.)	11,721	13,105	144,588
MADAND(AMMF) (mass elim.)	11,846	13,090	162,267
MADAND(AMIND) (mass elim.)	14,219	18,154	145,611
MADAND(AMD) with DT-pred. (1e-4)	11,891	12,473	147,124
MADAND(AMD) with DT-pred. (1e-2)	16,951	15,715	245,832
MADAND(AMD) with DT-pred. (1e-4) (mass e.)	11,589	12,215	145,823

For the random ordering the mean of three tests is taken. The implemented versions of MADAND(AMF) give similar results as Matlab's AMD-ordering. The ordering of METIS is an improvement compared to the old ordering, but it appears that local orderings give Cholesky factors with lower fill-in then global orderings. The results of the AMMF and AMIND variants seem to be highly matrix dependant. The drop tolerance prediction seems to work for very small values of η. However, because this gives a edge reduction of less then 4 percent it will probably not have a great influence on the performance of the ordering software.

Since the test-results of MADAND(AMF) and MADAND(AMD) are similar, and MADAND(AMF) is a bit more complex, we choose to use MADAND(AMD) in the Magma-software. In Table 8.4 the ordering time, compute time and total runtime of the MADAND(AMD) software is given, scaled with the results of the MINOLD ordering.

- SOT = OT(MADAND(AMD)) / OT(MINOLD).
- SCT = CT(MADAND(AMD)) / CT(MINOLD).
- SRT = RT(MADAND(AMD)) / RT(MINOLD).
- SMNZ = MNZ(MADAND(AMD)) / MNZ(MINOLD).

Table 8.4: Results of the MADAND(AMD) for Magma designs compared with MINOLD.

Testcase	SOT	SCT	SRT	SMNZ
testcase4	1.43	0.82	1.05	0.64
testcase5	1.45	0.63	0.83	0.59
testcase6a	1.59	0.41	0.59	0.38
testcase6b	1.65	0.51	0.72	0.45
testcase7a	2.01	0.76	0.86	0.66
testcase7b	1.88	0.65	0.73	0.64

An important note is that the time values may fluctuate a bit, since the tests are done on a communal server. However, compute time is significantly reduced for all testcases, and the overall runtime is smaller for the MADAND(AMD) software than for the MINOLD software for all large cases. In addition, the fill-in is approximately halved for every case.

Tie-Breaking Pre-ordering

In section 5.2.6 the tie-breaking pre-ordering was discussed. We tested three different pre-orderings for MADAND(AMD). The results are shown in Table 8.5.

Table 8.5: MNZ using MANDAND(AMD) for different pre-orderings.

Testcase	MINOLD	Random ordering	Matlab RCM
testcase1	12,058	12,600	12,092
testcase2	12,647	12,889	12,790

These results show that pre-ordering has some influence on the amount of fill-in. However, none of the orderings is significant better than any other, so we chose not to implement a special pre-ordering for in the Magma-software.

8.1.2 Multiple Processors

In this section we test the MADAND(CON) software for several variable configurations and up to four levels of dissection, which means that we ordered the matrix so we can solve it with 2, 4, 8 or 16 processors.

In the previous subsection we already saw that a nested dissection method like METIS gave considerably more fill-in than the local method AMD. This also holds for MADAND(CON), as shown in Figure 8.1. This is of course only one case, but the results later on in this section will support the fact that the number op partitions should be minimized.

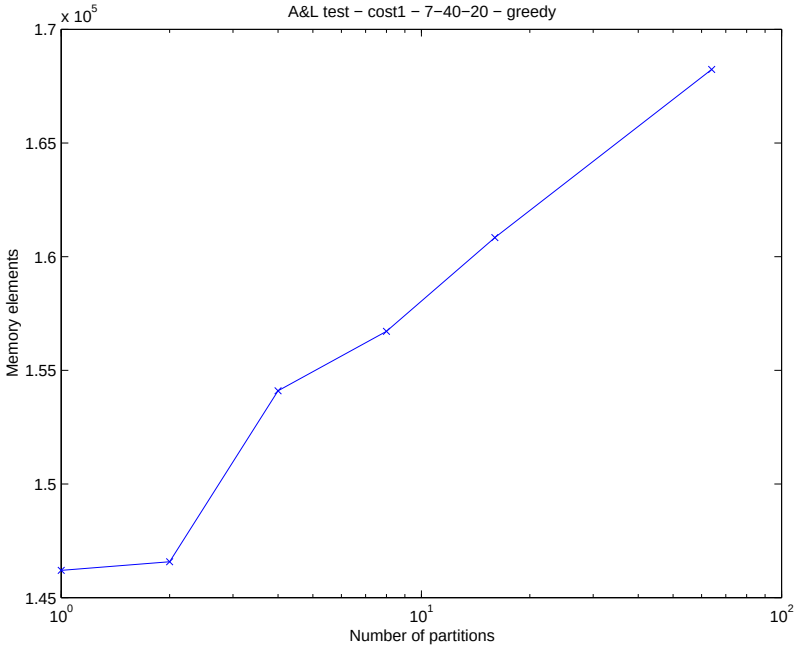

Figure 8.1: MNZ of a MADAND(CON) test for several different amount of processors.

The tested version of MADAND(CON) includes:

- Improved border (with use of the DM-decomposition).
- Constrained AMD.
- No mass elimination.

The parameters for the tests are described below:

- Cost function: we tested three different cost functions
  - $cost1 = |G|(1 + \frac{\max(|B|,|W|)}{\min(|B|,|W|)})$, the original cost function of A and L.
  - $cost2 = |G| + \max(|B|, |W|)$, to secure the balance.
  - $modcost1 = cost1$ if $\frac{\max(|B|,|W|)}{\min(|B|,|W|)} < \zeta$, and ∞ otherwise. For these tests $\zeta = 1.15$.
- Block creation variables: the values d_{max}, ω_{min} and ω_{max} should be chosen properly, since they have influence on the performance of our implementation. Experience shows that the value of d_{max} should imply that approximately 10% of the nodes are put in the border, and that $\frac{\omega_{max}}{\omega_{min}} = 2$ is the best choice to get blocks of similar sizes.
- Block creation: if a block is full and a new block is started, we can choose the first available node with the lowest index, or we can choose the an available node with a random index. Similarly, if we grow the blocks, we could start with low indexes first, or randomize. This is a form of greedy selection in contrast to random selection. We expect that random selection gives more balanced blocks and thus better results.

Table 8.6 shows the configurations for the 18 tests.

Table 8.6: Test configurations for testcase3.

7-40-20	greedy	random
cost1	test 1	test 4
cost2	test 2	test 5
modcost1	test 3	test 6

7-80-40	greedy	random
cost1	test 7	test 10
cost2	test 8	test 11
modcost1	test 9	test 12

7-60-30	greedy	random
cost1	test 13	test 16
cost2	test 14	test 17
modcost1	test 15	test 18

For each test there are two interesting output variables. The first one is the MNZ, because this is the number of elements that should be stored in the memory. The second one WLNZ, the amount of nonzero elements the busiest processor is handling, since this workload should give an indication how long the forward-backward substitution of the PCG-method will take. The results are shown in the tables en graphs below.

Table 8.7: MNZ for the MADAND(CON)-tests of testcase3

Ordering	2 proc.	4 proc.	8 proc.	16 proc.
MINOLD	237,278	237,278	237,278	237,278
Matlab AMD	144,281	144,281	144,281	144,281
test 1	146,574	154,098	156,713	160,836
test 2	146,681	155,662	159,516	164,572
test 3	146,843	154,883	157,527	160,482
test 4	146,574	153,249	156,326	158,427
test 5	146,433	154,990	160,404	166,097
test 6	146,485	152,870	156,973	158,490
test 7	146,396	152,431	157,076	unbalanced
test 8	146,466	154,745	159,710	164,812
test 9	146,396	155,278	159,387	no feasible cost
test 10	146,396	151,474	157,027	unbalanced
test 11	153,966	157,863	163,831	168,893
test 12	146,396	152,860	158,069	no feasible cost
test 13	146,652	155,377	157,433	unbalanced
test 14	154,349	156,576	160,798	165,680
test 15	151,482	157,179	161,000	163,894
test 16	146,680	153,182	155,576	157,062
test 17	153,431	155,949	161,560	167,513
test 18	146,680	153,182	155,576	158,514

Table 8.8: WLNZ for the MADAND(CON)-tests of testcase3

Ordering	2 proc.	4 proc.	8 proc.	16 proc.
MINOLD	237,278	237,278	237,278	237,278
Matlab AMD	144,281	144,281	144,281	144,281
test 1	84,268	52,363	38,156	31,106
test 2	76,410	49,701	38,312	33,787
test 3	76,541	49,475	36,011	33,151
test 4	84,268	52,162	38,327	31,107
test 5	76,334	50,293	36,884	33,398
test 6	75,733	49,027	34,334	29,104
test 7	78,178	52,914	38,038	unbalanced
test 8	76,122	49,919	38,103	34,076
test 9	78,178	49,740	37,709	no feasible cost
test 10	78,178	53,266	37,148	unbalanced
test 11	84,343	54,029	42,163	38,010
test 12	78,178	49,586	37,644	no feasible cost
test 13	75,523	58,943	40,731	unbalanced
test 14	84,140	52,697	41,649	37,713
test 15	82,194	53,738	38,534	34,204
test 16	75,630	50,877	35,879	30,055
test 17	83,203	52,114	42,309	39,390
test 18	75,630	50,877	35,879	29,961

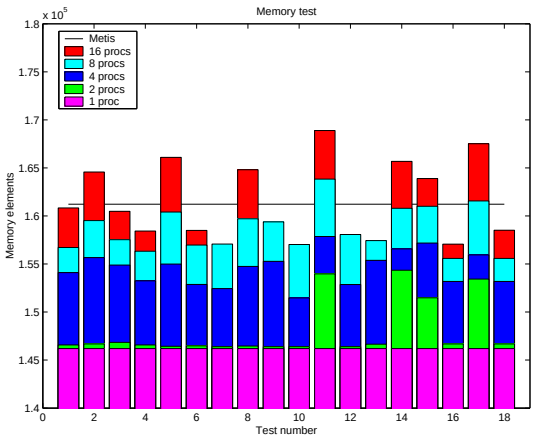

Figure 8.2: The MNZ for the 18 MADAND(CON) tests for 1, 2, 4, 8 and 16 processors in comparison with METIS.

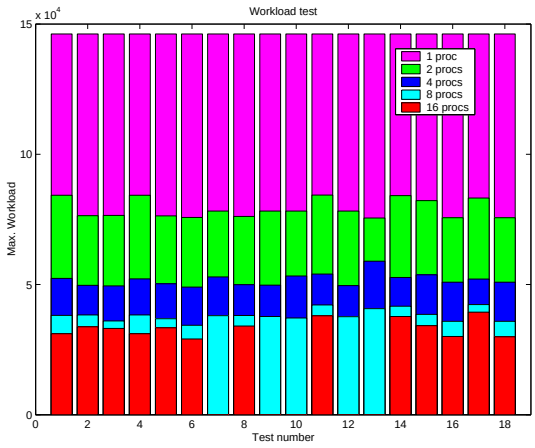

Figure 8.3: The WLNZ for the 18 MADAND(CON) tests for 1, 2, 4, 8 and 16 processors.

Note that the values of MNZ in Figure 8.2 start at approximately 140,000.

There are two possible problems with the dissection.

- The dissection is unbalanced. This means that a partition of level k is larger than a partition of level $k-1$.

- There is no feasible cost. For $modcost1$ there is a constraint for the balance between the two partitions. If the algorithm finds no configuration for which these constraints are satisfied it stops.

The results show that the number of processors has the greatest influence on the WLNZ, and in almost every case 16 processors is the best choice. Cost1 had difficulties with balance, and cost2 gives quite large separators. Modcost1 is a good alternative. However, we should think of a way to deal with cases for which there is no feasible cost. We could, for example, dissect that certain partition again with a looser bound. We also see that random block growing works best for cost1 and modcost1, and that greedy growing works best for cost2.

A very interesting observation is the minor increase in memory needed for 2 processors and the major increase for 4 processors This could be explained with the following conjecture: at the end of the minimum degree algorithm there are few nodes left which all form one clique. In a nested dissection algorithm with 2 partitions the separator consists also of few nodes that form a clique. But if there are 4 partitions, the 2 non-main separators form also a clique, and they are connected to all nodes in the main separator. So there will be an increasing amount of fill-in in the rows of the main separator.

We did also some tests of the MADAND(CON)-software for large Magma designs. For contest1, the chosen parameters are $d_{max} = 10, \omega_{max} = N/400, \omega_{min} = N/800$ and $modcost1$ with $\zeta = 1.3$. For contest2, the chosen parameters are $d_{max} = 10, \omega_{max} = N/200, \omega_{min} = N/400$ and $modcost1$ with $\zeta = 1.3$. The implemented block growth goes in a greedy fashion. The depth of the dissection is 3, so there are 1 + 2 + 4 = 7 partitions. We introduce:

- AMNZ = MNZ(MADAND(CON)) / MNZ(MADAND(AMD)).

- AWLNZ = WLNZ(MANDAND(CON)) / MNZ(MADAND(AMD)).

- AOT = OT(MANDAND(CON)) / OT(MADAND(AMD)).

Table 8.9: The MADAND(CON)-tests contest1 and contest2 for testcase5.

AMNZ	AWLNZ	AOT
1.00	0.30	1.15

PART	NOP	NZP
3	22.29	21.81
4	25.44	24.99
1	0.09	0.66
5	22.85	22.75
6	29.20	28.71
2	0.07	0.55
0	0.06	0.54

AMNZ	AWLNZ	AOT
1.00	0.30	1.10

PART	NOP	NZP
3	29.23	28.94
4	23.22	22.81
1	0.07	0.62
5	23.57	23.44
6	23.75	22.90
2	0.08	0.68
0	0.07	0.61

Table 8.10: The MADAND(CON)-tests contest1 and contest2 for testcase6a.

AMNZ	AWLNZ	AOT
1.01	0.39	1.40

PART	NOP	NZP
3	24.75	14.04
4	19.05	12.61
1	0.28	2.26
5	24.23	32.10
6	31.43	35.70
2	0.05	0.82
0	0.19	2.46

AMNZ	AWLNZ	AOT
1.01	0.36	1.19

PART	NOP	NZP
3	20.59	12.92
4	23.44	27.41
1	0.06	0.83
5	25.66	31.66
6	29.86	22.27
2	0.15	1.76
0	0.24	3.14

Table 8.11: The MADAND(CON)-tests contest1 and contest2 for testcase6b.

AMNZ	AWLNZ	AOT
1.01	0.31	1.16

PART	NOP	NZP
3	28.01	26.90
4	21.61	20.41
1	0.25	2.29
5	25.59	24.10
6	24.24	22.36
2	0.17	1.97
0	0.14	1.97

AMNZ	AWLNZ	AOT
1.01	0.31	1.11

PART	NOP	NZP
3	26.51	26.77
4	23.36	19.65
1	0.21	2.18
5	24.08	22.29
6	25.59	26.07
2	0.10	1.10
0	0.15	1.94

Table 8.12: The MADAND(CON)-tests contest1 and contest2 for testcase7a.

AMNZ	AWLNZ	AOT
1.01	0.30	1.22

PART	NOP	NZP
3	27.52	25.74
4	23.00	21.91
1	0.18	1.64
5	24.43	23.08
6	24.57	24.50
2	0.08	0.80
0	0.23	2.32

AMNZ	AWLNZ	AOT
1.01	0.31	1.13

PART	NOP	NZP
3	24.88	23.18
4	25.41	23.12
1	0.19	1.69
5	27.76	28.08
6	21.43	20.69
2	0.12	1.14
0	0.21	2.11

Table 8.13: The MADAND(CON)-tests contest1 and contest2 for testcase7b.

AMNZ	AWLNZ	AOT
1.02	0.30	1.24

PART	NOP	NZP
3	27.27	25.33
4	22.20	21.33
1	0.25	2.11
5	25.18	24.17
6	24.67	22.90
2	0.17	1.54
0	0.27	2.62

AMNZ	AWLNZ	AOT
1.01	0.32	1.10

PART	NOP	NZP
3	24.06	22.40
4	28.40	27.64
1	0.24	2.20
5	24.05	23.06
6	22.92	21.34
2	0.13	1.32
0	0.20	2.03

These results show that for large matrices MADAND(CON) generates an ordering that gives almost the same MNZ but a very small WLNZ compared to MADAND(AMD). Since the ordering time is often a small part of the total runtime (see also Table 8.4) and does not increase much for the MADAND(CON) case, it is very likely that using MADAND(CON) with a parallel solver will reduce the overall runtime substantially. The results for testcase6a are remarkable, partitions 3 and 5 have the same magnitude, but partition 5 has more than two times as much nonzero elements. So apparently the number of nodes in a partition is not always a good measure for the amount of fill-in.

8.2 Pre-Conjugate Gradient iterations

The results of ordering a matrix can be great considering fill-in, but what does ordering do with the number of iterations of the preconditioned conjugate gradient method?

In the table below, Table 8.14 we show for testcase3 that the number of iteration steps corresponds with the level of fill reducing of the ordering method (however, certainly not with the total amount of fill-in). The less fill-in, the less iteration steps.

Table 8.14: The number of iteration steps of the PCG-method for testcase3 for different values of ε and different orderings.

Ordering	ε = 0.01	ε = 0.1	ε = 1
MINOLD	5	7	11
Matlab AMD	4	6	10
METIS	4	6	11

As discussed in chapter 3 we can instead of choosing for equal rowsums of G and K also choose for equal diagonals of G and K. We tested Algorithm 12 with $\varepsilon = 0.1$ for three different ordering methods with $\vartheta = 0$ and $\vartheta = 1$. The results are shown in Figure 8.6.

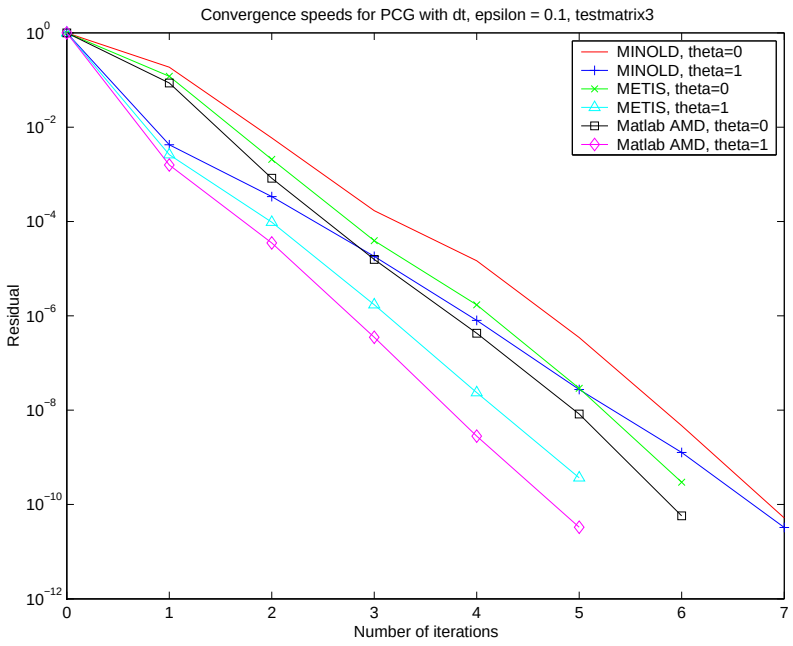

Figure 8.4: Convergence speed of the PCG-method for testcase3 for several methods.

The most interesting result is the difference in the first iteration step between $\vartheta = 0$ and $\vartheta = 1$. For all orderings, the results for $\vartheta = 1$ are significantly better than for $\vartheta = 0$ in the first step. After that, the fill-in of the matrix determines the convergence speed. So using $\vartheta = 1$ for $\varepsilon = 0.1$ can save an iteration step, which is about 20 percent.

We looked at the number of iteration steps for large Magma-designs and the results are shown in Table 8.15. The methods that are compared are MINOLD, MADAND(AMD) and MADAND(AMD) with $\vartheta = 1$.

Table 8.15: Number of iteration steps for different methods.

Testcase	MINOLD	MADAND, $\vartheta = 0$	MADAND, $\vartheta = 1$
testcase5	21	17	10
testcase6a	65	37	19
testcase6b	56	35	18
testcase7a	91	82	46

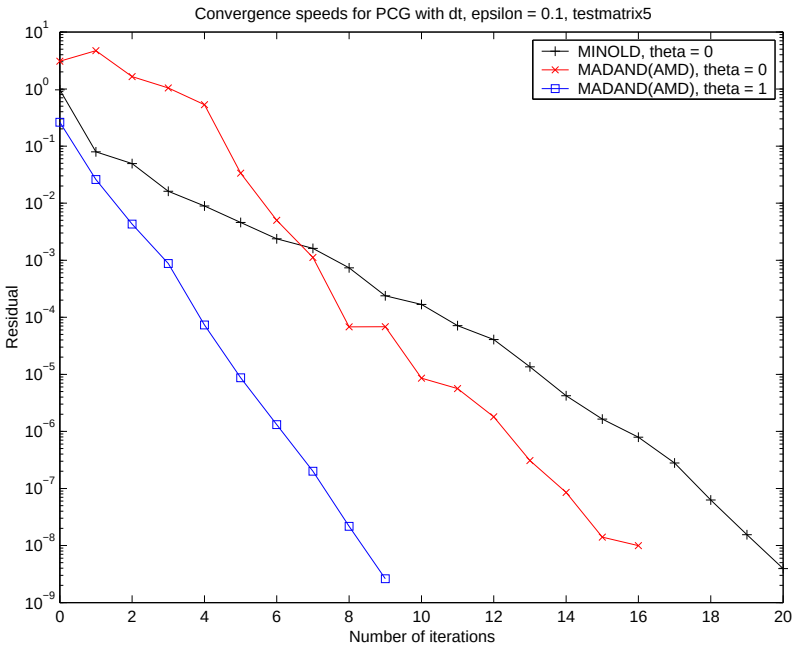

Figure 8.5: Convergence speed of the PCG-method for testcase5 for several methods

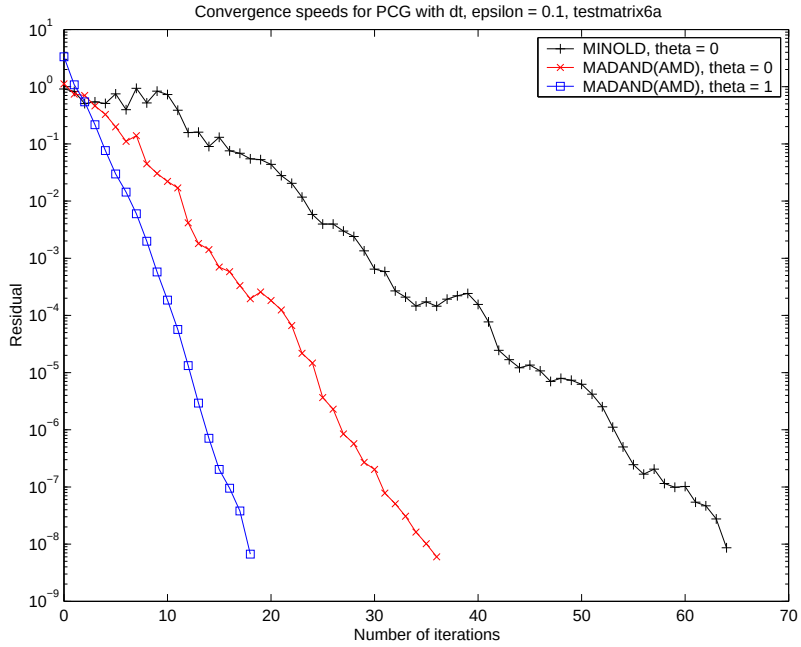

Figure 8.6: Convergence speed of the PCG-method for testcase6a for several methods

The value of ε has influence on the amount of fill-in and on the computing time. This is shown in Table 8.16.

- TCT = CT(MADAND(AMD,ϑ = 0)) / CT(MADAND(AMD,ϑ = 1))
- TMNZ = MNZ(MADAND(AMD,ϑ = 0)) / MNZ(MADAND(AMD,ϑ = 1))

Table 8.16: Results for MADAND(AMD) with $\vartheta = 1$, scaled with the results of MADAND(AMD) with $\vartheta = 0$

Testcase	TCT	TMNZ
testcase5	0.86	1.01
testcase6a	0.89	1.02
testcase6b	0.91	1.02
testcase7a	0.81	1.05

These results show that choosing $\vartheta = 1$ increases the amount of fill-in slightly, while the compute time of the solver is reduced considerably. Therefore taking $\vartheta = 1$ is a good addition for the MADAND-software.

Chapter 9

Conclusions and Recommendations

In this thesis we present the package MADAND, which consists out of two parts: the local ordering method MADAND(AMD) and the constrained ordering method MADAND(CON). MANDAND successfully improves the performance of the preconditioned conjugate gradient solver the company of Magma is currently using in comparison with the current ordering MINOLD. In addition, the amount of memory needed for the preconditioner matrix is reduced by 40 to 60 percent in comparison with the current ordering MINOLD.

. For general designs with an origin in circuit simulation local ordering methods appear to generate matrices who's Cholesky factor has less nonzero elements than with the use of global ordering methods. For non-parallel solving the MADAND(AMD) gives both little fill-in and fast ordering. In addition we have shown that equalizing the diagonal values of the original matrix and the preconditioner matrix can reduce the number of iteration steps considerably.

Global ordering methods can be used to solve a linear system parallel. The MADAND(CON)-software generates a nested dissection ordering for a small price considering fill-in and ordering time. However, the theory shows that using a parallel solver and the MADAND(CON)-software will lead to a faster runtime in comparison with MADAND(AMD).

The calculation time of the solver is reduced with the use of MADAND, but the ordering time may be done faster. First of all, the local ordering MADAND(AMD) can be done in parallel. If the partitions are found using the Nested Dissection ordering, all leaves can be processed independently. Since the AMD algorithm is the most time consuming part of the ordering, parallel ordering can reduce the total ordering time.

In addition, coarsening will lead to a smaller graph, which reduces the ordering time, since there are less nodes to eliminate and less edges. However, this coarsening is not trivial, since the AMD algorithm is local, and coarsening might interfere with this local search. Therefore a coarsening based on a local ordering [28] may result into both little fill-in and fast ordering.

Bibliography

[1] Amestoy, P., Davis, T., Duff, I., *An approximate minimum degree ordering algorithm*, SIAM J. Matrix anal. appl., Vol 17, no 4, pp. 886-905, 1996.

[2] Anis, M., Elmasry, M., *Multi-Threshold CMOS Digital Circuits, Managing Leakage Power*, Kluwer, Norwell, 2003.

[3] Ashcraft, C., Liu, J.W.H., *Using Domain Decomposition to find Graph Bisectors*, BIT, vol 37. pp. 506-534, 1997.

[4] Ashcraft, C., Liu, J.W.H., *Applications of the Dulmage-Mendelsohn Decomposition and Network Flow to Graph Bisection Improvement*, SIAM J. Matrix anal. appl., Vol 19, no 2, pp. 325-354, 1998.

[5] Barret, R., Berry, M., Chan, F., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philidelphia, 1994.

[6] Ciarlet, P.G., *Introduction to Numerical Linear Algebra and Optimisation*, Cambidge University Press, Cambridge, 1989.

[7] Duff, I.S., Van der Vorst, H.A., *Developments and Trends in the Parallel Solution of Linear Systems*, j. of Parallel Computing, vol 25, no 13-14, pp. 1931-1970, 1999.

[8] Garey, M., Johnson, D. Stockmeyer, L., *Some Simplified NP-complete Graph Problems*, Theoratical Computer Science, Vol. 1, pp. 237-267, 1976.

[9] George, A., Liu, W.L., *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, N.J., 1981.

[10] George, A., Liu, W.L., *The evolution of the minimum degree ordering algorithm* SIAM Rev., 31, pp. 1-19, 1989.

[11] George, A., McIntyre, D., *On the application of the minimum degree algorithm to finite element systems*, SINUM, vol 15, pp. 90-112, 1978.

[12] Golub, G.H., Van Loan, C.F., *Matrix Computations*, Johns Hopkins University Press, London, 1996.

[13] Gustafsson, I., *A class of 1st order factorization methods*, BIT, vol. 18, pp. 142-156, 1978.

[14] Heggernes, P., Eisenstat, S.C., Kumfert, G., Pothen, A., *The Computational Complexity of the Minimum Degree Algorithm*, Proceedings of 14th Norwegian Computer Science Conference, University of Troms, Norway, 2001.

[15] Hendrikson, B., Rothberg, E., *The CHACO User Guide. Version 2.0*, Technical Report SAND96-0868J, Sandia National Laboratories, Albuquerque, 1996.

[16] Houben, S.H.M.J., *Algorithms for Periodic Steady State Analysis on Electric Circuits*, Philips Electronics, Eindhoven, 1999.

[17] Houben, S.H.M.J., *Circuits in Motion: The numerical simulation of electrical oscillators*, Eindhoven University of Technology, Eindhoven, 2003.

[18] Karypis, G., Kumar, V., *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[19] Kernighan, B.W., Lin, S., *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, vol. 49, pp 291-307, 1970.

[20] Liu, W.L., *The minimum degree ordering with constraints*, SIAM J. Sci. Stat. Comp., Vol. 10, No. 6, pp. 1136-1145, 1989.

[21] Magma design automation, inc., *Enabling Low Power Design Within an RTL-to-GDSII Implementation Flow*, white paper, 2003.

[22] Mattheij, R.M.M., *Numerieke lineaire algebra*, Syllabus, Eindhoven University of Technology, Eindhoven.

[23] Meijerink, J.A., van der Vorst, H.A., *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, ACC, Utrecht, 1976.

[24] Ogrodzki, J., *Circuit Simulation Methods and Algorithms*, CRC Press Inc, Boca Raton, Florida, 1994.

[25] Rothberg, R., Eisenstat, C., *Node selection strategies for bottom-up sparse matrix ordering*, SIAM J. Matrix anal. appl. Vol 19, no. 3, pp 682-695, 1998.

[26] Saad, Y., *Iterative Methods for Sparse Linear Systems*, PWS Publishing Co., Boston, MA, 1996.

[27] Schilders, W.H.A., *Numerical methods in electromagnetics*, Eindhoven University of Technology, Eindhoven, 2004.

[28] Schulze, J., *Towards a Tighter Coupling of Bottem-Up and Top-Down Sparse Matrix Ordering Methods*, BIT, vol 41, pp 800-841, 2001.

[29] Varga, R.S., *Matrix Iterative Analysis*, Prentice-Hall Inc. New Jersey, 1962.

[30] Van der Vorst, H.A., *High performance preconditioning*, SIAM J. Sci. Stat. Comp. Vol 10, No 6, pp. 1174-1185, 1989.

[31] Zecevic, A.I., Siljak, D.D., *Balanced Decompositions of Sparse Systems for Multilevel Parallel Processing*, IEEE, Trans on circuits and systems 41, 1994.

[32] Zecevic, A.I., Siljak, D.D., *a Nested Decomposition Algorithm for Parallel Computations of very large Sparse Systems*, MPE, volume 1, pp 41-57, 1995.

Index

Appendix A

MADAND(NEST) Example

Consider the example in Figure A.1. The left figure shows a graph of 26 nodes. We choose the parameters $d_{max} = 5,\ \omega_{max} = 3,\ \omega_{min} = 2$. For this example we use $cost_1$ as the cost function.

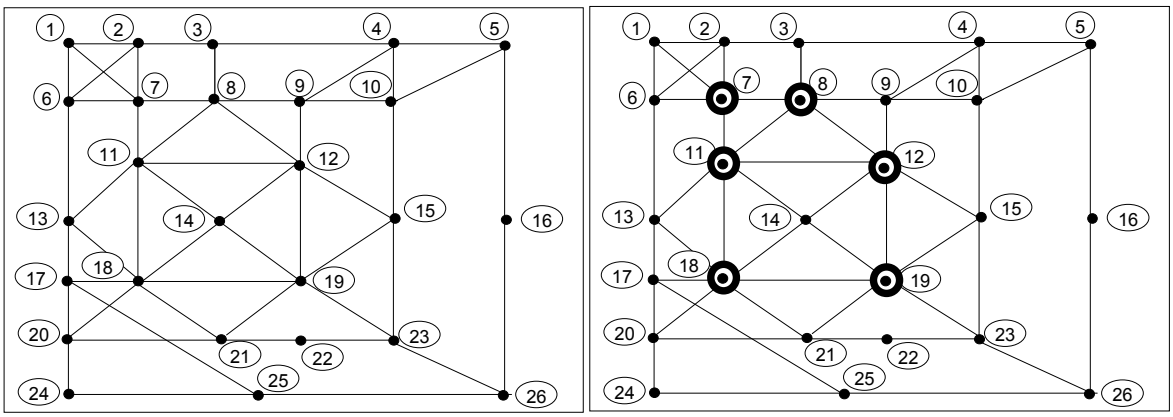

Figure A.1: MADAND(NEST) Example, Figure A.1.1 (left) and Figure A.1.2 (right).

Since $d_{max} = 5$, all nodes with degree 5 or higher go in the border, so $\Psi = \{v_7, v_8, v_{11}, v_{12}, v_{18}, v_{19}\}$. Nodes in the border are represented with a large circle and a dot inside. The next phase is to construct blocks in a breath-first way. So put node v_1 in block b_1. The $Reach$ becomes $\{v_2, v_6\}$. Now put node v_2 in b_1, the $Reach$ becomes $\{v_6, v_3\}$, and put node v_6 in b_1, the $Reach$ becomes $\{v_3, v_{13}\}$. Since $|b_1| = 3$, there is no room left in the block, so all nodes in the reach go into the border. This block-creation process is continued until all nodes are either in a block or in the border. This is described in Figure A.2.1. The result is of the block creation is

- $b_1 = \{v_1, v_2, v_6\}$
- $b_2 = \{v_4, v_5, v_9\}$
- $b_3 = \{v_{14}\}$
- $b_4 = \{v_{15}, v_{23}, v_{26}\}$
- $b_5 = \{v_{20}, v_{21}, v_{24}\}$

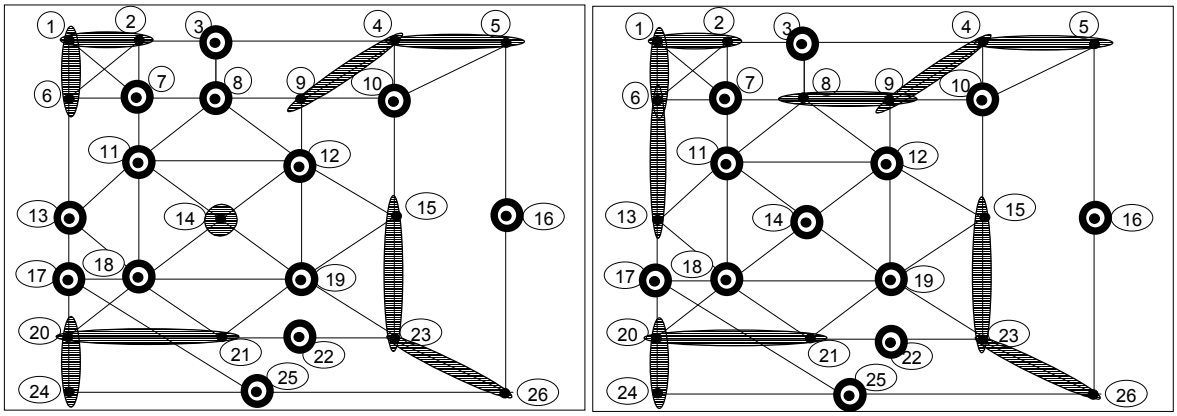

Figure A.2: MADAND(NEST) Example, Figure A.2.1 (left) and Figure A.2.2 (right).

Now all blocks with $|b_j| < 2$ are discarded, so block b_3 with $b_3 = \{v_{14}\}$ is discarded and v_{14} is added to the border. The remaining blocks can now grow. All nodes in the border with exactly one blockneighbour are considered, and the nodes with the smallest degree go first. The nodes with exactly one blockneighbour are $A = \{v_7, v_8, v_{13}, v_{17}, v_{18}\}$. Node v_{13} is picked, for it has degree 4, which is the lowest. After node v_{13} is added to b_1, nodes v_{17} and v_{18} are no longer considered, and there are no new nodes with one blockneighbour. Now v_8 is added to b_2, and the are no nodes left in A. The result is drawn in Figure A.2.2. The blocks are ready, but now the borderblocks must be formed. To make sure the dissection will be valid all adjacent nodes with no blockneighbours in common must be placed in the same borderblock. Node v_{11} has no blockneighbours in common with node v_{14} (since node v_{14} has no blockneighbours at all) so they are put in b_6. This b_6 has no blockneighbours in common with node v_{19}, so v_{19} is added to b_6. This is described in Figure A.3.1.

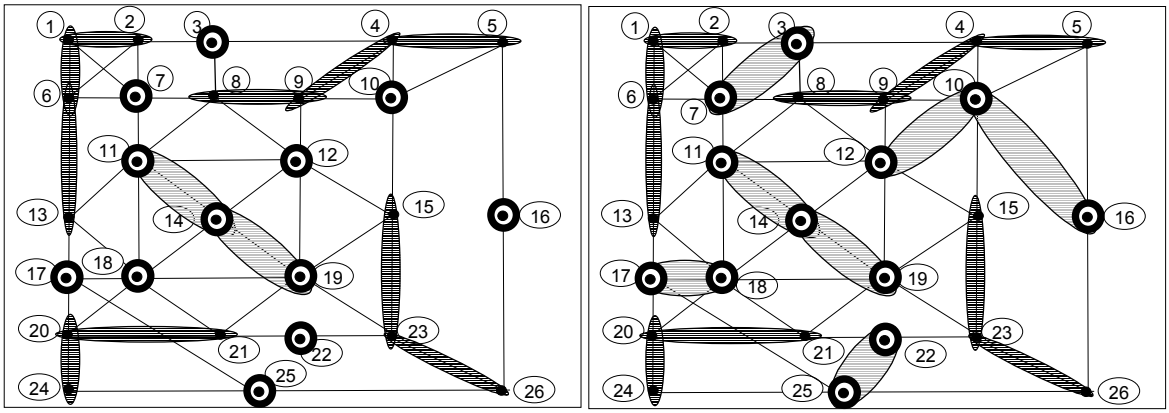

Figure A.3: MADAND(NEST) Example, Figure A.3.1 (left) and Figure A.3.2 (right).

The next step (Figure A.3.2) is to check if nodes that are not in a block yet can be added to an existing borderblock. They can be added if they have exactly the same blockneighbours. Node 3 divides b_1 and b_2, and since b_6 is the only borderblock that exists and divides b_1, b_2, b_4 and b_5 this node is put in b_7. Then node v_7 is also added to b_7. This is continued until all nodes are put in a (border)block. The configuration of the blocks is now

- $b_1 = \{v_1, v_2, v_6, v_{13}\}$

- $b_2 = \{v_4, v_5, v_8, v_9\}$
- $b_4 = \{v_{15}, v_{23}, v_{26}\}$
- $b_5 = \{v_{20}, v_{21}, v_{24}\}$
- $b_6 = \{v_{11}, v_{14}, v_{19}\}$
- $b_7 = \{v_3, v_7\}$
- $b_8 = \{v_{10}, v_{12}, v_{16}\}$
- $b_9 = \{v_{17}, v_{18}\}$
- $b_{10} = \{v_{22}, v_{25}\}$

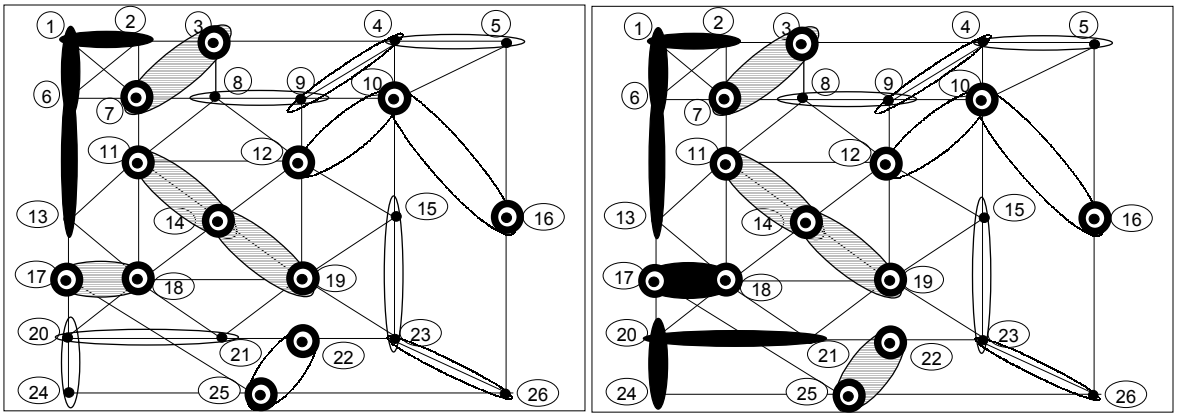

Figure A.4: MADAND(NEST) Example, Figure A.4.1 (left) and Figure A.4.2 (right).

All blocks are finished, it is time to paint them. Initially, all blocks are white and $mincost = \infty$. Take b_a, n_a an block. For all blocks $b_i (i < 6)$ that are not painted yet we calculate bordersize in the case the block would be painted, and minimum of these bordersizes is chosen. If b_1 is painted the bordersize is 7, which happens to be the minimum. After painting b_1, $mincost = 33.25$, since $cost = 7 * (1 + 15/4) = 33.25$. The next block that is painted is b_5, for which the bordersize is maintained. Now $mincost = 14.78$. See also Figure A.4. Now either b_4 or b_2 can be painted, but $cost$ will be higher then $mincost$. So the algorithm starts again with the configuration b_1 and b_5 black and b_2 and b_4 white, but this will not give a smaller $mincost$. The dissection of Figure A.4.2 gave $mincost$ and is therefore chosen.

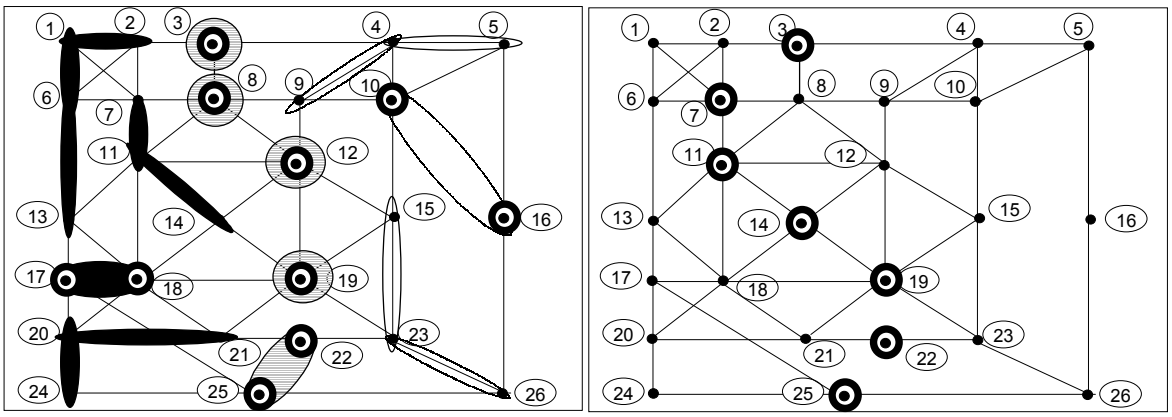

Figure A.5: MADAND(NEST) Example, Figure A.5.1 (left) and Figure A.5.2 (right).

A given dissection might become smaller using the Dulmage-Mendelsohn decomposition. The algorithm uses the border and the nodes adjacent to the border. Since White is the largest partition we try to move the border a bit to the white side. Consider all grey nodes G_n, $G_n = \{v_3, v_7, v_{11}, v_{14}, v_{19}, v_{22}, v_{25}\}$, and all adjacent white nodes W_G, $W_G = \{v_4, v_8, v_{12}, v_{15}, v_{23}, v_{26}\}$. Note that if W_G becomes grey and G_n becomes black, the dissection is still valid but smaller. However, for balance and construction arguments we prefer to switch only a small part of G_n and W_G. Consider all nodes in G_n and perform a maximal matching. This gives the pairs $\{v_3, v_4\}$, $\{v_7, v_8\}$, $\{v_{11}, v_{12}\}$, $\{v_{19}, v_{15}\}$, $\{v_{22}, v_{23}\}$ and $\{v_{25}, v_{26}\}$, and node v_{14} remains unmatched. So v_{14} is added to $switch_{grey}$. All adjacent white nodes (node v_{12}) are added to $switch_{white}$. Then all matches to the nodes in $switch_{white}$ (node v_{11}) are added to $switch_{grey}$. This procedure is repeated until $switch_{white}$ does not grow anymore, so it stops with $switch_{grey} = \{v_3, v_7, v_{11}, v_{14}\}$ and $switch_{white} = \{v_4, v_8, v_{12}\}$. If $switch_{grey}$ is painted black and $switch_{white}$ is painted white the border is smaller (Figure A.5.1). However, the new cost function $cost = 15$, so this smaller border is rejected. In the case the improvement is not rejected, the nodes are grow back to an existing block or form a new borderblock. The final configuration is shown in Figure A.5.2.

Appendix B

Pseudo code

B.1 MADAND(AMD)

Algorithm 17 MADAND(AMD)

$V = \{v_1, \ldots, v_n\},\ N = |V|,\ E = \{e_1, \ldots, e_m\},\ M = |E|,\ W = \emptyset$
for $i = 1 \ldots M$ **do**
  Pick $e_i = (v_j, v_k).\ \mathcal{A}_j = \mathcal{A}_j \cup v_k,\ \mathcal{A}_k = \mathcal{A}_k \cup v_j.$
end for
for $i = 1 \ldots N$ **do**
  Hash function initialization(v_i)
  $d_i = |\mathcal{A}_i|$, $w_i = -1$
end for
while $V \neq \emptyset$ **do**
  Pick v_p with $score(v_p) \leq score(v_j), v_j \in V$
  $\mathcal{A}_p = (\mathcal{A}_p \cup \bigcup_{v_j \in \mathcal{E}_p} \mathcal{A}_j) \backslash v_p$
  Edge manipulation(v_p)
  Degree update(v_p)
  Supernode detection
  Element creation(v_p)
end while

B.2 MADAND(NEST)

Algorithm 18 Edge manipulation(v_p)

for each $v_j \in \mathcal{A}_p$ **do**
  $\mathcal{A}_j = \mathcal{A}_j \backslash v_p$
  for each $v_k \in \mathcal{A}_j$ **do**
    if $v_k \in \mathcal{A}_p$ **then**
      $\mathcal{A}_j = \mathcal{A}_j \backslash v_k,\ \mathcal{A}_k = \mathcal{A}_k \backslash v_j$
    end if
  end for
  for each $v_k \in \mathcal{E}_j$ **do**
    if $v_k \in \mathcal{E}_p$ **then**
      $\mathcal{E}_j = \mathcal{E}_j \backslash v_k$
    else
      if $w_k < 0$ **then**
        $w_k = |\mathcal{A}_k|, W = W \cup v_k$
      end if
      $w_k = w_k - |v_j|$
    end if
  end for
end for

Algorithm 19 Degree update(v_p)

$bound1 = |V| - |v_p|$
for each $v_j \in \mathcal{A}_p$ **do**
  $bound2 = d_j + |\mathcal{A}_p| - |v_j|,\ bound3 = |\mathcal{A}_p| - |v_j|$
  for each $v_k \in \mathcal{A}_j$ **do**
    $bound3 = bound3 + |v_k|$
  end for
  for each $v_q \in \mathcal{E}_j$ **do**
    if $\mathcal{A}_q \backslash \mathcal{A}_p = \emptyset$ **then**
      $\mathcal{E}_k = \mathcal{E}_k \backslash v_q, \mathcal{A}_q = \emptyset$
    else
      $bound3 = bound3 + w_q$
    end if
  end for
  $d_j = \min(bound1, bound2, bound3),\ score(v_j) = scorefunction(d_j)$
end for

Algorithm 20 Element creation(v_p)

$V = V \backslash v_p,\ \mathcal{E}_p = \emptyset$
for each $v_k \in W$ **do**
  $w_k = -1$
end for
$W = \emptyset$

Algorithm 21 Hash-function initialization(v_i)

$hash_i = \sum \mathcal{A}_i + \sum \mathcal{E}_i \mod N$
$H(hash_i) = H(hash_i) \cup v_i$

Algorithm 22 Supernode detection

for $j = 0 \ldots N-1$ **do**
  if $|H(j)| > 1$ **then**
    for each $v_i \in H(j)$ **do**
      for each $v_k \in H(j), v_k \neq v_i$ **do**
        if $\mathcal{A}_i \cup v_i == \mathcal{A}_k \cup v_k$ **then**
          $ch_i = ch_i \cup v_k \cup ch_k$
          $V = V \backslash v_k, H(j) = H(j) \backslash v_k$
          $\mathcal{A}_k = \emptyset, \mathcal{E}_k = \emptyset$
        end if
      end for
    end for
  end if
end for
for $j = 0 \ldots N-1$ **do**
  $H(j) = \emptyset$
end for

Algorithm 23 Ashcraft and Liu (1997)

Choose $d_{max},\ \omega_{max},\ \omega_{min}, msb \in \mathbb{N},\ \omega_{max} > \omega_{min}$.
Initialize Border
Construct Blocks
Remove Small Blocks
Grow Remaining Blocks
Construct Borderblocks
Make Dissection

Algorithm 24 Initialize Border

$V = \{v_1, \ldots, v_n\},\ N = |V|,\ E = \{e_1, \ldots, e_m\},\ M = |E|,\ \Psi = \emptyset,$
for $i = 1 \ldots M$ **do**
  Pick $e_i = (v_j, v_k)$. $\mathcal{A}_j = \mathcal{A}_j \cup v_k,\ \mathcal{A}_k = \mathcal{A}_k \cup v_j$.
end for
for $i = 1, \ldots, N$ **do**
  if $|\mathcal{A}_i| \geq d_{max}$ **then**
    $\Psi = \Psi \cup v_i$
  end if
end for

Algorithm 25 Construct Blocks

$Z = V \backslash \Psi,\ t = -1, R = \Psi, Reach = \emptyset$
while $Z \neq \emptyset$ **do**
  Pick $v_{seed} \in Z$
  $t = t + 1,$ New block b_t
  $b_t = b_t \cup v_{seed},\ Z = Z \backslash v_{seed},\ Reach = adj(v_{seed}) \backslash R,\ R = R \cup v_{seed}$
  while $Reach \neq \emptyset$ **do**
    Pick $v_{reach} \in Reach$
    if $|b_t| \leq \omega_{max}$ **then**
      $b_t = b_t \cup v_{reach},\ Z = Z \backslash v_{reach},\ R = R \cup v_{reach}.\ Reach = (Reach \cup Adj(v_{reach})) \backslash (R \cup v_{reach}), R = R \cup Reach$
    else
      $\Psi = \Psi \cup v_i,\ Z = Z \backslash v_{reach},\ Reach = Reach \backslash v_{reach},\ R = R \cup v_{reach}$
    end if
  end while
end while
$\sigma = t$

Algorithm 26 Remove Small Blocks

for $j = 1, \ldots, t$ **do**
  if $|b_j| < \omega_{min}$ **then**
    for each $v_i \in b_j$ **do**
      $\Psi = \Psi \cup v_i$
    end for
    $b_j = \emptyset$
  end if
end for

Algorithm 27 Grow Remaining Blocks

$A = \emptyset$
for each $v_j \in \Psi$ **do**
  if $|\mathcal{NB}_j| = 1$ **then**
    $A = A \cup v_j$
  end if
end for
while $A \neq \emptyset$ **do**
  Pick $v_{min} \in \{v_u \in T || adj(v_u)| \leq |adj(v_w)|, v_w \in T\},\ A = A / v_{min}$
  if $|\mathcal{NB}_{min}| = 1$ **then**
    NB is the only block in $\mathcal{NB}_{min}$
    $NB = NB \cup v_{min},\ \Psi = \Psi \backslash v_{min}$
    for each $v_a \in \mathcal{A}_{min}$ **do**
      if $v_a \in \Psi$ **and** $|\mathcal{NB}_a| = 1$ **then**
        $A = A \cup v_a$
      end if
    end for
  end if
end while

Algorithm 28 Construct Borderblocks

$R = V \backslash \Psi,\ Reach = \emptyset$
for each $v_j \in \Psi$ **do**
  for each $v_a \in \mathcal{A}_j$ **do**
    if $\mathcal{NB}_j \cap \mathcal{NB}_a = \emptyset$ **then**
      $t = t + 1$, new block $b_t,\ b_t = b_t \cup v_j \cup v_a,\ \Psi = \Psi \backslash (v_a \cup v_j),\ R = R \cup v_j \cup v_a,\ Reach = (\mathcal{A}_a \cup \mathcal{A}_j) \backslash R$
      break
    end if
  end for
  while $Reach \neq \emptyset$ **do**
    Pick $v_{reach} \in Reach,\ Reach = Reach \backslash v_{reach}$
    if $\mathcal{NB}_{reach} \cap \mathcal{NB}_{b_t} = \emptyset$ **then**
      $b_t = b_t \cup v_{reach}, \Psi = \Psi \backslash v_{reach},\ R = R \cup v_{reach}, Reach = Reach \cup \mathcal{A}_{reach}$
    end if
  end while
end for
for each $b_j,\ j > \sigma$ **do**
  for each block $b_p,\ p > j$) **do**
    if $\mathcal{NB}_{b_j} = \mathcal{NB}_{b_p}$ **then**
      $b_j = b_j \cup b_p$
      $b_p = \emptyset$
    end if
  end for
end for
for each $v_i \in \Psi$ **do**
  for each block $b_j,\ j > \sigma$ **do**
    if $\mathcal{NB}_{b_j} = \mathcal{NB}_i$ **then**
      $b_j = b_j \cup v_i, \Psi = \Psi \cap v_i$
      break
    end if
  end for
  if $v_i \in \Psi$ **then**
    $t = t + 1,$ new block b_t
    $b_t = b_t \cup v_i, \Psi = \Psi \cap v_i$
  end if
end for

Algorithm 29 Make Dissection

$part = 0,\ P_0 = \bigcup b_j$
while $P_{part} > msb$ **do**
  Dissect-Partition(P_{part})
end while

Algorithm 30 Dissect-Partition(P_{part})

$H = \emptyset,\ S = \emptyset,\ W = P_{part},\ B = \emptyset,\ G = \emptyset,\ W^* = \emptyset,\ B^* = \emptyset,\ G^* = \emptyset$
for each $b_j \in P_{part}$ **do**
  if $j < \sigma$ **then**
    $H = H \cup b_j$
  end if
end for
$mc = \infty$
repeat
  while $H \neq \emptyset$ **do**
    $sc = \infty$
    for each $b_j \in H$ **do**
      if paint(b_j)$\rightarrow min|G|$ **then**
        $b_{paint} = b_j$
      end if
    end for
    Paint Block(b_{paint})
    Update sc
    if $sc < mc$ **then**
      $mc = sc,\ W^* = W,\ B^* = B,\ G^* = G$
    end if
  end while
  $msc = mc,\ H = S,\ W = W^*,\ B = B^*,\ G = G^*$
until mcs = mc

Algorithm 31 Paint Block(b_{paint})

if $b_{paint} \in W$ **then**
  for each $b_j \in \mathcal{NB}_{b_{paint}}$ **do**
    if $b_j \in G$ **and** $b_k \in B$ **for all** $b_k \in \mathcal{NB}_{b_j}$ **then**
      $G = G \backslash b_j,\ B = B \cup b_j$
    else if $b_j \in W$ **then**
      $G = G \cup b_j,\ W = W \backslash b_j$
    end if
  end for
  $W = W \backslash b_{paint}, B = B \cup b_{paint}$
else if $b_{paint} \in B$ **then**
  for each $b_j \in \mathcal{NB}_{b_{paint}}$ **do**
    if $b_j \in G$ **and** $b_k \in W$ **for all** $b_k \in \mathcal{NB}_{b_j}$ **then**
      $G = G \backslash b_j,\ W = W \cup b_j$
    else if $b_j \in B$ **then**
      $G = G \cup b_j,\ B = B \backslash b_j$
    end if
  end for
  $B = B \backslash b_{paint}, W = W \cup b_{paint}$
end if

Algorithm 32 Improve Border

if $|W| > |B|$ **then**
  repeat
    Improve(W^*, B^*)
  until $G = G^*$
  repeat
    Improve(B^*, W^*)
  until $G = G^*$
else
  repeat
    Improve(B^*, W^*)
  until $G = G^*$
  repeat
    Improve(W^*, B^*)
  until $G = G^*$
end if
Ready nodes
Grow Remaining Blocks
Construct Borderblocks

Algorithm 33 Improve(source,target)

$\Omega = \{v_i | v_i \in b_i, b_i \in G\}$
$\Delta = \emptyset$
for $v_i \in \Omega$ **do**
  for $v_j \in Adj(\Omega), v_j$ has colour source **do**
    if $v_i \notin \Delta \wedge v_j \notin \Delta$ **then**
      match $m(v_i, v_j)$, $\Delta = \Delta \cup v_i \cup v_j$
    end if
  end for
end for
$Q = \Omega \cap \Delta$, $Z = \emptyset$
repeat
  $Q = Q \cup Z$
  $X = \{v_j \in Adj(Q) \mid v_j$ has colour source $\}$
  $Z = \{v_k \mid \exists m(v_k, v_j), v_j \in X\}$
until $Z \cap Q = \emptyset$
$Q = Q \cup Z$
if $|Q| > |X| \wedge impsc < sc$ **then**
  $store(target) = store(target) \cup Q$
  $store(G) = store(G) \cup X$
end if

Algorithm 34 Ready Nodes

$G = \emptyset$
$G = G \cup store(G)$
for $v_i \in store(W)$ **do**
  $\Psi = \Psi \cup v_i$
end for
for $v_i \in store(B)$ **do**
  $\Psi = \Psi \cup v_i$
end for

Appendix C

Results

The tables are constructed in the following way: the first column consists of the number of processors and the total number of nonzero elements of the matrix (the MNZ). The second column contains the partition numbers. The next two columns contain the number of nonzero elements of that partition (NZP) and the number of nodes in that partition, respectively. Then, depending on the number of processors, there are m (with number of processors $p = 2^m$) times 3 additional columns. The first of these columns represent the NZP between the partitions $1 \ldots \frac{2^m}{p}, \ldots, (p-1)\frac{2^m}{p} + 1 \ldots 2^m + 1$, the second number is the number of nodes in that separator and the third number is the MNZP of this partition, defined by MNZP $:= \max($ MNZP1,MNZP2) + NZP, with MNZP1 and MNZP2 the MNZP of the 2 blocks that are dissected by this separator. For the first level, the MNZP = NZP. Finally, the last MNZP is bold and is equal to the WLNZ.

1 proc MNZ				costfunction block-var. greedy/random												TEST X
	1	**WLNZ**														
2 proc MNZ	1	nz(1)	#1	3 = b(1|2)												
	2	nz(2)	#2	nz(3)	#3	**WLNZ**										
4 proc MNZ	1	nz(1)	#1	5 = b(1|2)												
	2	nz(2)	#2	nz(5)	#5	wl(5)										
	3	nz(3)	#3	6 = b(3|4)			7=b(5|6)									
	4	nz(4)	#4	nz(6)	#b6	wl(6)	nz(7)	#7	**WLNZ**							
8 proc MNZ	1	nz(1)	#1	9=b(1|2)												
	2	nz(2)	#2	nz(9)	#9	wl(9)										
	3	nz(3)	#3	10=b(3|4)			13=b(9|10)									
	4	nz(4)	#4	nz(10)	#10	wl(10)	nz(13)	#13	wl(13)							
	5	nz(5)	#5	11=b(5|6)												
	6	nz(5)	#6	nz(11)	#11	wl(11)										
	7	nz(7)	#7	12=b(7|8)			14=b(11|12)			15=b(13|14)						
	8	nz(8)	#8	nz(12)	#12	wl(12)	nz(14)	#14	wl(14)	nz(15)	#15	**WLNZ**				
16 proc MNZ	1	nz(1)	#1	17=b(1|2)												
	2	nz(2)	#2	nz(17)	#17	wl(17)										
	3	nz(3)	#3	18=b(3|4)												
	4	nz(4)	#4	nz(18)	#18	wl(18)	nz(25)	#25	wl(25)							
	5	nz(5)	#5	19=b(5|6)												
	6	nz(5)	#6	nz(19)	#19	wl(19)										
	7	nz(7)	#7	20=b(7|8)												
	8	nz(8)	#8	nz(20)	#20	wl(20)	nz(26)	#26	wl(26)	nz(29)	#29	wl(29)				
	9	nz(9)	#9	21=b(9|10)												
	10	nz(10)	#10	nz(21)	#21	wl(21)										
	11	nz(11)	#11	22=b(11|12)												
	12	nz(12)	#12	nz(22)	#22	wl(22)	nz(27)	#27	wl(27)							
	13	nz(13)	#13	23=b(13|14)												
	14	nz(14)	#14	nz(23)	#23	wl(23)										
	15	nz(15)	#15	24=b(15|16)												
	16	nz(16)	#16	nz(24)	#24	wl(24)	nz(28)	#28	wl(28)	nz(30)	#30	wl(30)	nz(31)	#31	**WLNZ**	

1 proc 146200	1	**146200**																
							COST1			7-40-20			GREEDY			TEST 1		
2 proc 146574	1	80678	5862															
	2	62306	4677	3590	35	**84268**												
4 proc 154098	1	39193	2941															
	2	37358	2878	5436	43	44629												
	3	29306	2178															
	4	30563	2456	4508	43	35071	7734	35	**52363**									
8 proc 156713	1	16757	1483															
	2	17517	1418	4520	40	22037												
	3	19041	1485															
	4	14424	1355	4137	38	23178	7242	43	30420									
	5	12251	1060															
	6	12174	1081	4404	37	16655												
	7	13487	1163															
	8	12972	1246	3927	47	17414	6130	43	23544	7736	35	**38156**						
16 proc 160836	1	6749	714															
	2	6761	730	3248	39	10009												
	3	7093	706															
	4	6913	674	3618	38	10711	5219	40	15930									
	5	7280	706															
	6	7110	727	4152	52	11432												
	7	6285	680															
	8	5522	635	2742	40	9027	4854	38	16286	6910	43	23196						
	9	4872	522															
	10	4450	481	3312	57	8184												
	11	4663	528															
	12	4479	505	3313	48	7976	5125	37	13309									
	13	4698	542															
	14	5841	580	2818	41	8659												
	15	3112	345															
	16	8490	881	1630	20	10120	4859	47	14979	6768	43	21747	7910	35	**31106**			

1 proc 146200	1	**146200**		COST2		7-40-20		GREEDY				TEST 2			
2 proc	1	72447	5264												
146681	2	70271	5263	3963	47	**76410**									
4 proc	1	36841	2636												
	2	32710	2584	4251	44	41092									
155662	3	36563	2629												
	4	32159	2578	4411	56	40974	8727	47	**49819**						
8 proc	1	14054	1267												
	2	15647	1319	6550	50	22197									
159516	3	13938	1302												
	4	15512	1235	3897	47	19409	6795	44	28992						
	5	15573	1293												
	6	14027	1285	5410	51	20983									
	7	15478	1302												
	8	12709	1233	3615	42	19093	6991	57	27974	9320	47	**38312**			
16 proc	1	6071	666												
	2	5437	557	3256	44	9327									
164572	3	5509	596												
	4	6687	675	4067	48	10754	5423	50	16177						
	5	5618	616												
	6	5473	639	3164	47	8782									
	7	5334	557												
	8	5763	625	3517	53	9280	5175	47	14455	7324	44	23501			
	9	6021	621												
	10	5755	619	4133	53	10154									
	11	4941	608												
	12	5749	622	3294	55	9043	5706	51	15860						
	13	5510	621												
	14	6621	631	2998	50	9619									
	15	5474	587												
	16	4493	594	3494	52	8968	4638	42	14257	8340	57	24200	9587	47	**33787**

1 proc 146200	1	**146200**																	
		MODCOST1 7-40-20 GREEDY																	TEST 3
2 proc 146843	1	70302	5109																
	2	72384	5426	4157	39	**76541**													
4 proc 154883	1	34766	2551																
	2	31565	2516	4561	42	39327													
	3	35160	2615																
	4	34570	2772	4113	39	39273	10148	39	**49475**										
8 proc 157527	1	14825	1301																
	2	14511	1205	5840	45	20665													
	3	13060	1239																
	4	15125	1237	3669	40	18794	6597	42	27262										
	5	12148	1296																
	6	14634	1247	8011	72	22645													
	7	15352	1277																
	8	16022	1443	4367	52	20389	5688	39	28333	7678	39	**36011**							
16 proc 160490	1	5554	625																
	2	6311	631	3271	45	9582													
	3	5649	579																
	4	5387	581	3187	45	8836	4958	45	14540										
	5	5257	607																
	6	5267	589	2982	43	8249													
	7	5194	557																
	8	6426	634	3536	46	9962	4313	40	14275	6525	42	21065							
	9	6029	651																
	10	5618	628	679	17	6708													
	11	5401	586																
	12	4893	581	5148	80	10549	8365	72	18914										
	13	6791	621																
	14	5917	625	2737	30	9528													
	15	6806	699																
	16	6937	719	1757	25	8694	5358	53	14886	5500	39	24414	8737	39	**33151**				

1 proc 146200	1	**146200**		COST1	7-40-20	RANDOM							TEST 4		
2 proc 146574	1	80678	5862												
	2	62306	4677	3590	35	**84268**									
4 proc	1	38444	2949												
	2	36928	2871	5465	42	43909									
153249	3	30284	2464												
	4	29418	2171	4457	42	34741	8253	35	**52162**						
8 proc	1	16454	1461												
	2	17894	1444	4974	44	22868									
156326	3	18768	1480												
	4	14293	1353	4370	38	23138	7485	42	30623						
	5	13584	1165												
	6	13303	1256	3076	43	16660									
	7	11721	1055												
	8	12122	1077	4544	39	16666	6034	42	22700	7704	35	**38327**			
16 proc	1	6377	693												
	2	6761	731	3161	37	9922									
158427	3	6882	703												
	4	7203	703	3603	38	10806	5360	44	16166						
	5	7379	717												
	6	6962	712	3885	51	11264									
	7	6448	689												
	8	5369	625	2684	39	9132	5045	38	16309	7149	42	23458			
	9	4997	560												
	10	5676	564	2870	41	8546									
	11	7616	819												
	12	3825	413	1945	24	9561	4208	43	13769						
	13	4667	505												
	14	4798	513	2557	37	7355									
	15	3439	425												
	16	5802	602	3257	50	9059	4335	39	13394	6518	42	20287	7649	35	**31107**

1 proc 146200	1	**146200**					COST2		7-40-20		RANDOM						TEST 5		
2 proc	1	72870	5287																
146433	2	70099	5248	3464	39	**76334**													
4 proc	1	35262	2620																
	2	33263	2618	4168	49	39430													
154990	3	32004	2597																
	4	35503	2597	5038	54	40541	9752	39	**50293**										
8 proc	1	14446	1284																
	2	14869	1284	6749	52	21618													
160404	3	13675	1287																
	4	16235	1283	4472	48	20707	6531	49	28149										
	5	14379	1260																
	6	13714	1276	4623	61	19002													
	7	15306	1277																
	8	13827	1269	5433	51	20739	7581	54	28320	8564	39	**36884**							
16 proc	1	5561	592																
	2	6316	647	2877	45	9193													
166097	3	5605	581																
	4	5757	612	5117	91	10874	6132	52	17006										
	5	5646	618																
	6	5201	623	3125	46	8771													
	7	5801	593																
	8	5918	634	3643	56	9561	5365	48	14926	7595	49	24601							
	9	5323	606																
	10	5458	598	3467	56	8925													
	11	6206	619																
	12	6148	630	1488	27	7694	6118	61	15043										
	13	5944	616																
	14	5921	607	4044	54	9988													
	15	5485	636																
	16	5551	596	2551	37	8102	5706	51	15694	8231	54	23925	8797	39	**33398**				

1 proc 146200	1	**146200**																		
2 proc 146485	1	70752	5134																	
	2	72132	5405	3601	35	**75733**														
4 proc 152870	1	32488	2514																	
	2	32686	2579	5403	41	38089														
	3	33266	2543																	
	4	35668	2821	3583	41	39251	9776	35	**49027**											
8 proc 156973	1	13894	1231																	
	2	15294	1241	5667	42	20961														
	3	12727	1217																	
	4	16979	1324	4037	38	21016	5150	41	26166											
	5	13526	1270																	
	6	14422	1226	5518	47	19940														
	7	16203	1356																	
	8	15443	1421	4497	44	20700	5448	41	26148	8168	35	**34334**								
16 proc 158490	1	4861	569																	
	2	5721	605	3605	57	9326														
	3	5712	584																	
	4	5970	615	3178	42	9148	4785	42	14111											
	5	5381	596																	
	6	4815	579	2613	42	7994														
	7	6821	650																	
	8	6262	630	3173	44	9994	4474	38	14468	6099	41	20567								
	9	5396	630																	
	10	6168	603	2711	37	8879														
	11	5991	610																	
	12	5399	576	3086	40	9077	4985	47	14062											
	13	6705	611																	
	14	6802	657	3150	38	9952														
	15	7185	695																	
	16	6583	649	1707	27	8892	4819	44	14771	6140	41	20911	8193	35	**29104**					

MODCOST1 7-40-20 RANDOM

TEST 6

1 proc				COST1	7-80-40	GREEDY							TEST 7		
146200	1	**146200**													
2 proc	1	74727	5446												
146396	2	68218	5093	3451	35	**78178**									
4 proc	1	35928	2800												
	2	35377	2602	4196	44	40124									
149034	3	40619	3175												
	4	20619	1877	3854	41	44473	8441	35	**52914**						
8 proc	1	13655	1305												
	2	18320	1452	5047	43	23367									
157077	3	15727	1290												
	4	14041	1264	5460	48	21187	6374	44	29741						
	5	19636	1623												
	6	16517	1506	5311	46	24947									
	7	10849	964												
	8	8718	874	4331	39	15180	5959	41	30906	7132	35	**38038**			
16 proc	1														
	2					0									
0	3														
	4					0			0						
	5														
	6					0									
	7														
	8					0			0			0			
	9														
	10					0									
	11														
	12					0			0						
	13														
	14					0									
	15														
	16					0			0			0			**0**

1 proc 146200	1	**146200**																	
2 proc 146466	1	72441	5292																
	2	70344	5244	3681	38	**76122**													
4 proc 154745	1	34049	2690																
	2	33936	2544	5132	58	39181													
	3	31709	2538																
	4	36498	2657	4346	49	40844	9075	38	**49919**										
8 proc 159710	1	13655	1305																
	2	16699	1342	4773	43	21472													
	3	14093	1244																
	4	12959	1229	7074	71	21167	7803	58	29275										
	5	13240	1245																
	6	14694	1245	3791	48	18485													
	7	16270	1366																
	8	14336	1245	5062	46	21332	6433	49	27765	8828	38	**38103**							
16 proc 164812	1	5447	605																
	2	5536	659	2671	41	8207													
	3	5614	604																
	4	6988	687	3516	51	10504	5425	43	15929										
	5	5784	591																
	6	5171	591	3508	62	9292													
	7	6065	617																
	8	4142	555	3381	57	9446	6802	71	16248	8707	58	24955							
	9	5987	596																
	10	5526	613	2203	36	8190													
	11	5037	576																
	12	5699	608	3806	61	9505	5107	48	14612										
	13	5939	626																
	14	6707	689	3769	51	10476													
	15	5031	531																
	16	6136	668	3175	46	9311	5144	46	15620	7668	49	23288	9121	38	**34076**				

COST2 7-80-40 GREEDY

TEST 8

1 proc 146200	1	**146200**																	
		MODCOST1 7-80-40 GREEDY																TEST 9	
2 proc	1	74727	5446																
146396	2	68218	5093	3451	35	**78178**													
4 proc	1	35928	2800																
	2	35377	2602	4196	44	40124													
155278	3	33363	2495																
	4	30037	2525	8058	73	41421	8319	35	**49740**										
8 proc	1	13655	1305																
	2	18320	1452	5047	43	23367													
159387	3	15727	1290																
	4	14041	1264	5460	48	21187	6374	44	29741										
	5	12542	1175																
	6	14666	1255	6082	65	20748													
	7	13370	1160																
	8	12096	1313	5046	52	18416	9911	73	30659	7050	35	**37709**							
16 proc	1																		
	2					0													
0	3																		
	4					0			0										
	5																		
	6					0													
	7																		
	8					0			0			0							
	9																		
	10					0													
	11																		
	12					0			0										
	13																		
	14					0													
	15																		
	16					0			0			0							**0**

1 proc																				
146200	1	**146200**																		
2 proc	1	74727	5446																	
146396	2	68218	5093	3451	35	**78178**														
4 proc	1	34535	2602																	
	2	35951	2800	3756	44	39707														
151474	3	23966	1915																	
	4	39843	3139	4288	39	44131	9135	35	**53266**											
8 proc	1	14041	1264																	
	2	15925	1290	5399	48	21324														
157027	3	13655	1305																	
	4	18623	1452	5195	43	23818	5928	44	29746											
	5	11393	1006																	
	6	8561	866	4366	43	15759														
	7	18747	1533																	
	8	17156	1559	4938	47	23685	5698	39	29383	7402	35	**37148**								
16 proc	1																			
	2					0														
0	3																			
	4					0			0											
	5																			
	6					0														
	7																			
	8					0			0			0								
	9																			
	10					0														
	11																			
	12					0			0											
	13																			
	14					0														
	15																			
	16					0			0			0								**0**

COST1 7-80-40 RANDOM

TEST 10

1 proc 146200	1	**146200**															TEST 11
2 proc 153966	1	73178	5207														
	2	69623	5294	11165	73	**84343**											
4 proc 157863	1	34403	2596														
	2	33341	2564	4099	47	38502											
	3	31991	2655														
	4	34860	2596	4181	43	39041	14988	73	**54029**								
8 proc 163831	1	14150	1272														
	2	15660	1274	5863	50	21523											
	3	14280	1216														
	4	13924	1301	4879	47	19159	6182	47	27705								
	5	14883	1284														
	6	11192	1270	7863	101	22746											
	7	14336	1235														
	8	13250	1279	7952	82	22288	5972	43	28718	13445	73	**42163**					
16 proc 168893	1	4291	579														
	2	5689	626	4256	67	9945											
	3	5554	589														
	4	6992	624	3482	61	10474	5742	50	16216								
	5	5810	588														
	6	5325	579	3490	49	9300											
	7	5053	581														
	8	6232	676	2763	44	8995	5408	47	14708	7870	47	24086					
	9	6016	597														
	10	6178	655	2627	32	8805											
	11	5921	631														
	12	4907	631	249	8	6170	8702	101	17507								
	13	5638	584														
	14	5646	608	2958	43	8604											
	15	6032	656														
	16	5377	581	2771	42	8803	7590	82	16393	6400	43	23907	13924	73	**38010**		

1 proc																		
146200	1	**146200**																
2 proc	1	74727	5446															
146396	2	68218	5093	3451	35	**78178**												
4 proc	1	34535	2602															
	2	35951	2800	3756	44	39707												
152860	3	31255	2536															
	4	33717	2515	3767	42	37484	9879	35	**49586**									
8 proc	1	14041	1264															
	2	15925	1290	5399	48	21324												
158069	3	13655	1305															
	4	18623	1452	5195	43	23818	5928	44	29746									
	5	13234	1242															
	6	14813	1249	3789	45	18602												
	7	13699	1211															
	8	13840	1252	6124	52	19964	5906	42	25870	7898	35	**37644**						
16 proc	1																	
	2					0												
0	3																	
	4					0			0									
	5																	
	6					0												
	7																	
	8					0			0			0						
	9																	
	10					0												
	11																	
	12					0			0									
	13																	
	14					0												
	15																	
	16					0			0			0						**0**

MODCOST1 7-80-40 RANDOM

TEST 12

1 proc 146200	1	**146200**				COST1	7-60-30		GREEDY			TEST 13		
2 proc 146652	1	71600	5378											
	2	71129	5160	3923	36	**75523**								
4 proc 155377	1	32992	2555											
	2	34522	2782	4080	41	38602								
	3	42061	3036											
	4	24840	2082	4740	42	46801	12142	36	**58943**					
8 proc 157433	1	16305	1298											
	2	12765	1206	4649	51	20954								
	3	16620	1345											
	4	14887	1396	3346	41	19966	5613	41	26567					
	5	17490	1498											
	6	19393	1499	5152	39	24545								
	7	12179	1098											
	8	9874	949	2974	35	15153	6838	42	31383	9348	36	**40731**		
16 proc 0	1													
	2					0								
	3													
	4					0			0					
	5													
	6					0								
	7													
	8					0			0			0		
	9													
	10					0								
	11													
	12					0			0					
	13													
	14					0								
	15													
	16					0			0			0		**0**

1 proc 146200	1	**146200**															TEST 14
2 proc 154349	1	75358	5248														
	2	70209	5245	8782	81	**84140**											
4 proc 156576	1	33803	2617														
	2	33400	2569	8502	62	42305											
	3	32180	2584														
	4	33446	2608	4853	53	38299	10392	81	**52697**								
8 proc 160798	1	13482	1264														
	2	14286	1290	6894	63	21180											
	3	14086	1220														
	4	13428	1304	4355	45	18441	9102	62	30282								
	5	14733	1237														
	6	13833	1306	4877	41	19610											
	7	15713	1332														
	8	12047	1228	5816	48	21529	6779	53	28308	11367	81	**41649**					
16 proc 165680	1	5585	573														
	2	6002	646	2868	45	8870											
	3	5329	582														
	4	6462	664	3310	44	9772	6903	63	16675								
	5	5472	584														
	6	5385	589	3417	47	8889											
	7	5119	589														
	8	5556	670	2925	45	8481	4705	45	13594	9529	62	26204					
	9	5819	610														
	10	5503	583	2988	44	8807											
	11	5177	623														
	12	5180	617	4299	66	9479	4946	41	14425								
	13	5850	614														
	14	6613	667	4229	51	10842											
	15	5335	548														
	16	4309	637	2580	43	7915	5059	48	15901	7717	53	23618	11509	81	**37713**		

1 proc 146200	1	**146200**					MODCOST1 7-60-30 GREEDY									TEST 15
2 proc 151482	1	74447	5328													
	2	69288	5181	7747	65	**82194**										
4 proc 157179	1	34679	2612													
	2	34686	2671	4613	45	39299										
	3	35802	2558													
	4	29463	2581	4712	42	40514	13224	65	**53738**							
8 proc 161000	1	14761	1257													
	2	14558	1299	5793	56	20554										
	3	16536	1361													
	4	13767	1269	4832	41	21368	6487	45	27855							
	5	15285	1291													
	6	15624	1222	4699	45	20323										
	7	16132	1367													
	8	10828	1175	3748	39	19880	7271	42	27594	10679	65	**38534**				
16 proc 163894	1	5994	595													
	2	6719	634	2288	28	9007										
	3	5956	642													
	4	6127	615	2611	42	8738	6035	56	15042							
	5	6302	653													
	6	6329	660	3852	48	10181										
	7	5400	607													
	8	5341	614	2981	48	8381	4752	41	14933	7498	45	22540				
	9	5708	593													
	10	6031	646	3542	52	9573										
	11	5557	591													
	12	5559	578	3273	53	8832	5830	45	15403							
	13	5205	643													
	14	6437	653	4861	71	11298										
	15	5194	579													
	16	5098	579	536	17	5730	4077	39	15375	7452	42	22855	11349	65	**34204**	

1 proc 146200	1	**146200**																								
2 proc 146680	1	72128	5408																							
	2	71050	5131	3502	35	**75630**																				
4 proc 153182	1	34353	2584																							
	2	34644	2784	4003	40	38647																				
	3	35507	2727																							
	4	29305	2365	6259	39	41766	9111	35	**50877**																	
8 proc 155576	1	13582	1235																							
	2	16225	1298	4539	51	20764																				
	3	16166	1349																							
	4	14920	1397	3671	38	19837	5767	40	26531																	
	5	15822	1325																							
	6	15276	1356	6207	46	22029																				
	7	12769	1128																							
	8	12561	1196	4221	41	16990	5875	39	27904	7975	35	**35879**														
16 proc 157062	1	5752	642																							
	2	5425	558	2725	35	8477																				
	3	6540	645																							
	4	5879	615	3211	38	9751	5310	51	15061																	
	5	6837	626																							
	6	6923	696	2188	27	9111																				
	7	7322	721																							
	8	6188	646	1616	30	8938	4258	38	13369	6018	40	21079														
	9	5008	548																							
	10	7857	739	3365	38	11222																				
	11	5829	647																							
	12	6348	668	3779	41	10127	4653	46	15875																	
	13	4542	515																							
	14	5127	570	2831	43	7958																				
	15	4657	552																							
	16	5417	602	2804	42	8221	4473	41	12694	6481	39	22356	7699	35	**30055**											

COST1 7-60-30 RANDOM

TEST 16

1 proc 146200	1	**146200**														TEST 17
2 proc 153431	1	75370	5248													
	2	70228	5245	7833	81	**83203**										
4 proc 155949	1	33968	2592													
	2	33629	2585	7919	71	41887										
	3	31896	2581													
	4	33659	2613	4651	51	38310	10227	81	**52114**							
8 proc 161560	1	13564	1264													
	2	13764	1257	7510	71	21274										
	3	14398	1232													
	4	13395	1305	4692	48	19090	9217	71	30491							
	5	13833	1306													
	6	14312	1234	4542	41	18854										
	7	14724	1267													
	8	12306	1283	7022	63	21746	6463	51	28209	11818	81	**42309**				
16 proc 167513	1	5671	612													
	2	6233	615	2356	37	8589										
	3	4734	553													
	4	6441	659	3269	45	9710	7478	71	17188							
	5	5907	604													
	6	5290	577	3431	51	9338										
	7	5071	590													
	8	5593	670	2790	45	8383	4859	48	14197	10091	71	27279				
	9	5969	647													
	10	4964	602	3541	57	9510										
	11	6040	618													
	12	5301	575	2815	41	8855	5111	41	14621							
	13	6041	604													
	14	5708	604	4047	59	10088										
	15	6069	651													
	16	4273	590	2227	42	8296	6241	63	16329	7841	51	24170	12111	81	**39390**	

COST2 7-60-30 RANDOM

1 proc 146200	1	**146200**																
		MODCOST1 7-60-30 RANDOM															TEST 18	
2 proc 146680	1	72128	5408															
	2	71050	5131	3502	35	**75630**												
4 proc 153182	1	34353	2584															
	2	34644	2784	4003	40	38647												
	3	35507	2727															
	4	29305	2365	6259	39	41766	9111	35	**50877**									
8 proc 155576	1	13582	1235															
	2	16225	1298	4539	51	20764												
	3	16166	1349															
	4	14920	1397	3671	38	19837	5767	40	26531									
	5	15822	1325															
	6	15276	1356	6207	46	22029												
	7	12769	1128															
	8	12561	1196	4221	41	16990	5875	39	27904	7975	35	**35879**						
16 proc 158514	1	5044	601															
	2	5450	582	3812	52	9262												
	3	6510	646															
	4	5879	615	3150	37	9660	5243	51	14903									
	5	6837	626															
	6	6923	696	2188	27	9111												
	7	6075	631															
	8	6657	720	2520	46	9177	4747	38	13924	6059	40	20962						
	9	5904	597															
	10	6657	677	3664	51	10321												
	11	5383	617															
	12	6685	692	3745	47	10430	5310	46	15740									
	13	4542	515															
	14	5127	570	2831	43	7958												
	15	4657	552															
	16	5417	602	2804	42	8221	4473	41	12694	6505	39	22245	7716	35	**29961**			