

# A Bulk Synchronous Parallel Algorithm for Minimum Degree Ordering

Frank van Lingen

Master's Thesis  
Department of Mathematics  
Utrecht University  
August 1996



# A Bulk Synchronous Parallel Algorithm for Minimum Degree Ordering

Frank van Lingen

Thesis Supervisor: Rob Bisseling

August 1996

The figure on the front page is the nonzero structure of the lower triangular part of the symmetric positive definite matrix 1138BUS. The size of this matrix is 1138 and it contains 2595 nonzeros. This matrix is used in modeling power system networks.



# Preface

This is my masters thesis. I have written it in the period of December 1995 until August 1996.

In this thesis we present a parallel minimum degree algorithm. The minimum degree algorithm is a algorithm that reorders the columns of a symmetric positive definite matrix and tries to minimize the fill in one gets during Choleski factorisation. We used the BSP model ([11], [3]) for analyzing the costs and the Oxford BSP library for building the parallel program.

As a preliminary study we first build a sequential program using a couple of the acceleration techniques described in ([4],[8]).

In the first chapter we explain Choleski factorisation and the connection between symmetric matrices and undirected graphs. We also present the basic minimum degree algorithm and describe some of its acceleration techniques. The second chapter describes the sequential data structure we used and the separate program functions we used in our program. Furthermore we present an alternative data structure and the results of our implementation on several matrices. In chapter three we describe a parallel data structure and give a description of the program functions we use. Furthermore we give a cost analysis of the functions and the results of our implementation on different numbers of processors.

I would like to thank my thesis supervisor Rob Bisseling for his advice and the numerous discussions we had about the subject. Furthermore my thanks go out to Kees, Olive, Marc and Eric. They have been a support for years. I also want to thank Adam, Angela, Edo, Denise, Jeanet, Jelka, Hanneke, Michel, Patrick, Peter and Sandra for the various (positive) distractions from my thesis. Finally, I thank Frans for all the funny emails he sent during the time I was working on my thesis.

Frank van Lingen  
Utrecht, August 1996.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Choleski factorization . . . . .	1
1.2	Sparse Choleski factorization . . . . .	2
1.3	Graph representation . . . . .	2
1.4	Minimum degree algorithm . . . . .	4
1.5	Acceleration techniques . . . . .	5
1.5.1	Mass elimination . . . . .	6
1.5.2	Indistinguishable nodes . . . . .	6
1.5.3	Clique representation . . . . .	6
1.5.4	Element absorption . . . . .	7
1.5.5	Multiple elimination . . . . .	7
<b>2</b>	<b>Sequential minimum degree</b>	<b>9</b>
2.1	Clique structure . . . . .	10
2.2	Degree structure . . . . .	11
2.3	Program functions . . . . .	13
2.3.1	Merge . . . . .	13
2.3.2	DegreeUpdate . . . . .	14
2.3.3	MultipleElimination . . . . .	15
2.3.4	MassElimination . . . . .	16
2.4	Alternative: a static data structure . . . . .	16
2.5	Results . . . . .	19
<b>3</b>	<b>Parallel minimum degree</b>	<b>23</b>
3.1	Introduction . . . . .	23
3.2	Data structure . . . . .	26
3.3	Program functions . . . . .	28
3.3.1	Choose minimum degree node . . . . .	29
3.3.2	Merging cliques . . . . .	29
3.3.3	Merge . . . . .	30
3.3.4	Updating nodes . . . . .	32
3.3.5	Degree update . . . . .	34
3.3.6	Remove minimum degree node . . . . .	36
3.4	Results . . . . .	36

<b>4</b>	<b>Conclusions/Future work</b>	<b>41</b>
<b>A</b>	<b>The sequential program</b>	<b>43</b>
<b>B</b>	<b>The parallel minimum degree program</b>	<b>59</b>
	<b>Bibliography</b>	<b>89</b>

# Chapter 1

## Introduction

### 1.1 Choleski factorization

Solving symmetric positive definite systems  $A\mathbf{x} = \mathbf{b}$  is a problem which arises in various applications. Here  $A$  is a symmetric positive definite matrix,  $\mathbf{b}$  a vector and  $\mathbf{x}$  the unknown vector. One way to solve this problem is by factorization of the matrix  $A = LL^T$ . Here  $L$  is a lower triangular matrix, with non zeros on the diagonal. This factorization is unique (see [5] p. 141). With this information one can derive a simple algorithm for finding  $L$ . Let's say that we have found the first  $k$  columns of  $L$ . So we have found the entries  $l_{i,j}$   $0 \leq i < n$ ,  $0 \leq j < k$ , with  $l_{i,j} = 0$  if  $j > i$ . We now want to find the  $(k+1)$ 'th column of  $L$ , which are the entries  $l_{i,k}$   $k \leq i < n$ . We can do this by analysing the factorization  $LL^T = A$  and deriving the following relations between the entries of  $L$  and  $A$  :

$$\sum_{p=0}^k l_{k,p}^2 = a_{k,k} \Rightarrow l_{k,k} = \sqrt{a_{k,k} - \sum_{p=0}^{k-1} l_{k,p}^2} \quad (1.1)$$

$$\sum_{p=0}^k l_{i,p}l_{k,p} = a_{i,k} \Rightarrow l_{i,k} = \frac{a_{i,k} - \sum_{p=0}^{k-1} l_{i,p}l_{k,p}}{l_{k,k}} \quad i > k \quad (1.2)$$

This leads to the following simple algorithm for Choleski factorization:

```
FOR k=0..(n-1)
   $l_{k,k} = \sqrt{a_{k,k} - \sum_{p=0}^{k-1} l_{k,p}^2}$ 
  FOR i=k+1..(n-1)
     $l_{i,k} = \frac{a_{i,k} - \sum_{p=0}^{k-1} l_{i,p}l_{k,p}}{l_{k,k}}$ 
```

Algorithm for Choleski factorization

If we subtract two real numbers from each other that both are nonzero, we will make the assumption that the outcome will never be zero. The number of times that it becomes zero is small compared to the number of subtractions, since a subtraction

of two real numbers only becomes zero if both numbers have the same floating point representation. Even if this happens, one can ask the question: Is this because of roundoff errors or not? Looking at the algorithm for Choleski factorization above: if  $a_{i,j} \neq 0$  with  $j < i$  then  $l_{i,j} \neq 0$ . We thus define fill in to be these entries  $l_{i,j}$ ,  $i > j$ , of  $L$  such that  $a_{i,j} = 0 \wedge l_{i,j} \neq 0$ . Furthermore, the  $k$ 'th pivot is that column in matrix  $A$  that is not in the Choleski factorization yet, which will be used in the  $k$ 'th step of the algorithm. In the above algorithm the  $k$ 'th pivot is the  $k$ 'th column of  $A$ .

## 1.2 Sparse Choleski factorization

In many problems  $A$  is a sparse matrix. For example ten percent of the entries maybe nonzero. It would be nice if the lower triangular matrix  $L$  has that same amount of sparsity (in the lower triangular part). This would give us an advantage in processing time, storage capacity and numerical stability. The sparsity of  $L$  depends strongly on the permutation of the columns of  $A$ . If  $A$  has the following nonzero structure ( $x$ =nonzero),  $L$  would have a large fill in.

$$A = \begin{pmatrix} x & x & x & \dots & x \\ x & x & 0 & \dots & 0 \\ x & 0 & x & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ x & 0 & \dots & 0 & x \end{pmatrix} \quad (1.3)$$

Using formulas 1.1 and 1.2  $l_{i,k}$  with  $k < i < n$  and  $0 \leq k < n$  can only be zero if  $a_{i,k} = 0$  and  $\sum_{p=0}^{k-1} l_{i,p} l_{k,p} = 0$ . Because  $a_{j,0} \neq 0$  in  $A$ ,  $l_{j,0} \neq 0$  for  $0 \leq j < n$ . This means  $l_{i,0} \neq 0$  and  $l_{k,0} \neq 0$ , and thus  $l_{i,0} l_{k,0} \neq 0$ . This implies  $\sum_{p=0}^{k-1} l_{i,p} l_{k,p} \neq 0$  and thus  $l_{i,k} \neq 0$  with  $k < i < n$  and  $0 \leq k < n$ . Since  $a_{k,k} \neq 0$  also  $l_{k,k} \neq 0$ . Thus  $l_{i,k} \neq 0$  with  $0 \leq k < n$  and  $k \leq i < n$ . This means that all entries of the lower triangular part of the Choleski matrix  $L$  will be nonzero. So the fill in will be enormous. Using the same techniques as above, it's easy to see that if we permute the columns of  $A$  in the following manner,  $L$  will suffer no fill in.

$$A = \begin{pmatrix} x & 0 & \dots & 0 & x \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & x & 0 & x \\ 0 & \dots & 0 & x & x \\ x & \dots & x & x & x \end{pmatrix}$$

So we hope to find a permutation matrix  $P$  such that if we factorize  $PAP^T$ ,  $L$  will suffer minimum fill in.

## 1.3 Graph representation

To get a better understanding of the fill in, we will now look at the matrix  $A$  as a graph.  $A$  is an  $n \times n$  matrix. Every column represents a vertex in graph  $G$ . Column

$i$  is identified with vertex  $v_i$  in graph  $G$ . There is an undirected edge between vertex  $v_i$  and vertex  $v_j$  if and only if  $i \neq j$  and  $a_{i,j} \neq 0$ . We obtain the following graph  $G(V, E)$ :  $V = \{v_0, v_1, \dots, v_{n-1}\}$  and  $E = \{(v_i, v_j) \mid a_{i,j} \neq 0 \wedge i \neq j\}$ . Furthermore, the degree of a node  $v_i$  is the number of neighbours in graph  $G$ . We can on the other hand rewrite  $A$  like:

$$A = \begin{pmatrix} d & z^T \\ z & B \end{pmatrix}.$$

Here,  $B$  is an  $(n-1) \times (n-1)$  matrix,  $d \in \mathfrak{R}$ ,  $z \in \mathfrak{R}^{n-1}$ .

We now can factorize  $A$  like:

$$\begin{pmatrix} d & z^T \\ z & B \end{pmatrix} = L_0 \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & B - \frac{zz^T}{d} & \\ 0 & & & \end{pmatrix} L_0^T, \quad (1.4)$$

such that  $L_0 = \begin{pmatrix} \sqrt{d} & 0 & \dots & \dots & 0 \\ & 1 & 0 & \dots & 0 \\ \frac{z}{\sqrt{d}} & 0 & \ddots & \ddots & \vdots \\ & \vdots & \ddots & \ddots & 0 \\ & 0 & \dots & 0 & 1 \end{pmatrix}.$

Here, the first column of  $L_0$  is the first column of  $L$ . The next column of  $L$  can be found by factorizing matrix  $B - \frac{zz^T}{d}$ , the same way as  $A$ . This will result in a product of three matrices:  $B - \frac{zz^T}{d} = L_1 C L_1^T$ . Here  $L_1$  is a  $(n-1) \times (n-1)$  matrix in which the second upto the  $(n-1)$ 'th diagonal entry is 1, and with the same structure as  $L_0$ . The first column of  $L_1$  is the second column of  $L$  (the  $i$ 'th column of  $L$  has  $(n-i+1)$  entries because we do not count zeros). We can repeat the procedure with the  $(n-2) \times (n-2)$  submatrix of  $C$ , and thus find the third, fourth, ...  $n$ 'th column of  $L$  in this order. To avoid fill in in the Choleski factorization, the entries which are zero in  $B$  should remain zero after computing the first column of  $L$  by (1.4). Because for the second column of  $L$  we will factorize  $B - \frac{zz^T}{d}$  and the first column of  $L_1$  is the second column of the Choleski factor  $L$ . When is an entry  $(i, j)$  from matrix  $B - \frac{zz^T}{d}$  nonzero? This is the case when  $b_{i,j} \neq 0$  ( $b_{i,j}$  is an entry from  $B$ ) or  $(z_i \neq 0 \wedge z_j \neq 0)$ . If we look at the matrix interpretation, fill in occurs at entries:  $\{b_{i,j} \mid b_{i,j} = 0 \wedge a_{i,0} \neq 0 \wedge a_{j,0} \neq 0\}$ . After the factorization in (1.4) we will proceed with the smaller matrix  $B - \frac{zz^T}{d}$  to find the next column of  $L$ . If we look at graph  $G(V, E)$  induced by matrix  $A$ , the following operations will be done when we use the first column of  $A$  as the first pivot:

$$G_0(V_0, E_0) = G(V, E)$$

$$E_1 = (E_0 \setminus \{(v_0, v_j) \mid (v_0, v_j) \in E_0\}) \cup \{(v_i, v_j) \mid i \neq j \wedge (v_i, v_0), (v_j, v_0) \in E_0\}$$

$$V_1 = V_0 \setminus \{v_0\}$$

Use the new graph  $G_1(V_1, E_1)$  in pivoting the next column/node.

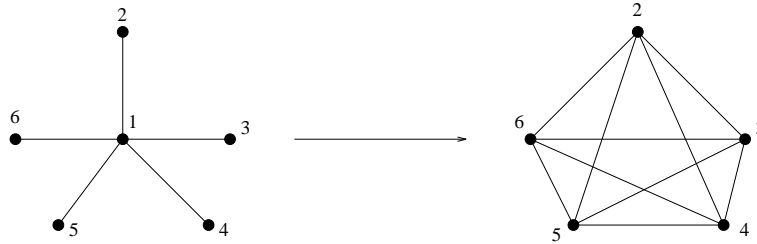


Figure 1.1: The effect of fill in

$G_1$  is the same as the graph induced by matrix  $B - \frac{zz^T}{d}$  (only the indices differ). If we repeat the factorization with  $B - \frac{zz^T}{d}$ , and we repeat the above procedure with  $v_1$ , graph  $G_2(V_2, E_2)$  is the graph induced by the submatrix of  $C$ . In general one can say that using the  $k$ 'th column from  $A$  as the  $k$ 'th pivot for Choleski factorization implies the following operations on graph  $G_{k-1}$ , such that  $G_k$  equals the graph induced by the remaining submatrix:

$$E_k = (E_{k-1} \setminus \{(v_{k-1}, v_j) \mid (v_{k-1}, v_j) \in E_{k-1}\}) \cup \{(v_i, v_j) \mid i \neq j \wedge (v_i, v_{k-1}), (v_j, v_{k-1}) \in E_{k-1}\}$$

$$V_k = V_{k-1} \setminus \{v_{k-1}\}$$

Use the new graph  $G_k(V_k, E_k)$  in pivoting the next column/node.

The amount of fill in when pivoting a column/node is exactly equal to the number of edges one has to insert to connect the former neighbours to become a clique (a clique is a set of nodes that are completely connected). So the order in which you pivot the columns can have a dramatic effect on the overall fill in. If we first pivot column/node 1 in figure (1.1) then there is a large fill in. Equation (1.3) shows the structure of the matrix that induced the graph in figure (1.1).

The fill in would be zero if we would first pivot the columns/nodes 2,3,4,5 or 6. So what we need is a permutation of the columns that minimizes the fill in. Unfortunately this problem is NP-complete (see [12]), so that the best we can do is to find an approximate minimum.

## 1.4 Minimum degree algorithm

One of the approximation algorithms for this problem is the minimum degree algorithm. This algorithm works as follows:

```

 $G_0(V_0, E_0) = G$ 
 $k = 0$ 
WHILE ( $k < n$ ) DO
{
  Select minimum degree node  $v \in G_k(V_k, E_k)$ 
 $E_{k+1} = (E_k \setminus \{(v, v_i) \mid (v, v_i) \in E_k\}) \cup$ 
 $\{(v_i, v_j) \mid i \neq j \wedge (v, v_i), (v, v_j) \in E_k\}$ 
 $V_{k+1} = V_k \setminus \{v\}$ 
 $k = k + 1$ 
}

```

We start with the initial graph  $G = G_0$  induced by the  $n \times n$  matrix  $A$ . We search for the node with minimum degree, remove it and connect all its neighbours with each other, so that they form a clique. This is our next graph  $G_1$ . We repeat this procedure on  $G_1$  and so forth until we obtain graph  $G_{n-1}$  which contains one node. The sequence of nodes we get from this, is the order in which we have to pivot the columns of our matrix  $A$ . The idea behind this heuristic is that if we eliminate a node that has minimum degree, the number of edges we have to insert to connect its neighbours is low. Because inserting an edge means fill in in the Choleski factorization, one hopes that the total fill in stays low. The graphs  $G_0, \dots, G_{n-1}$  are called elimination graphs. If one implements this heuristic on a computer the degree update (calculating  $E_{k+1}$  in the above algorithm) of the neighbours of the eliminated node is the most time consuming part. There are a couple of acceleration techniques to speed up the algorithm. We will discuss several techniques. For more detailed information about these techniques see [4] and [8].

## 1.5 Acceleration techniques

We shall use the notation  $Adj_G(v)$  to refer to the nodes adjacent to  $v$  in elimination graph  $G$ . Furthermore,  $v \notin Adj_G(v)$ . The degree of node  $v$  is denoted by  $Degree_G(v)$ , which is simply  $|Adj_G(v)|$ . First we will derive a theorem which will help us in describing some of the acceleration techniques.

Define:  $x \equiv y$  in  $G \Leftrightarrow \{x\} \cup Adj_G(x) = \{y\} \cup Adj_G(y)$

If two nodes have this property we say that  $x$  and  $y$  are *indistinguishable* in  $G$ . Note that if  $x \equiv y$  in  $G_i$  and  $x, y \in G_j$  with  $j > i$  then  $x \equiv y$  in  $G_j$ .

Define:  $K_x^j = \{y \mid x \equiv y \text{ in } G_i \wedge y \in G_j\}$  for  $j \geq i$ .

Let  $z \in K_x^j$  be a node of minimum degree in elimination graph  $G_j$  that we eliminate to get graph  $G_{j+1}$ .

**Lemma 1.1** *All nodes in  $K_x^j$  have minimum degree*

proof: Let  $y \in K_x^j$  then  $y \equiv z$  and  $Degree_{G_j}(y) = |Adj_{G_j}(y)| = |\{y\} \cup Adj_{G_j}(y)| - 1 = |\{z\} \cup Adj_{G_j}(z)| - 1 = |Adj_{G_j}(z)| = Degree_{G_j}(z)$ .

**Lemma 1.2**  $(y \in K_x^j \wedge y \neq z) \Rightarrow Degree_{G_{j+1}}(y) = Degree_{G_j}(y) - 1$

proof:  $Degree_{G_{j+1}}(y) = |Adj_{G_{j+1}}(y)| = |Adj_{G_j}(y) \setminus \{z\} \cup Adj_{G_j}(z) \setminus \{y\}| =$   
 $|Adj_{G_j}(z) \cup \{z\} \setminus \{y\} \setminus \{z\} \cup Adj_{G_j}(z) \setminus \{y\}| = |Adj_{G_j}(z) \setminus \{y\} \cup Adj_{G_j}(z) \setminus \{y\}| =$   
 $|Adj_{G_j}(z) \setminus \{y\}| = |Adj_{G_j}(z)| - 1 = Degree_{G_j}(z) - 1 = Degree_{G_j}(y) - 1$

**Lemma 1.3** *If the minimum degree in elimination graph  $G_j$  is  $d_j$  then the minimum degree in graph  $G_{j+1}$  is greater than or equal to  $d_j - 1$*

proof: We eliminate one node, so the degrees of the remaining nodes can only decrease by one in  $G_{j+1}$ .

Above lemma's lead to the following theorem:

**Theorem 1.1** *Let  $z \in K_x^j$  be a minimum degree node in elimination graph  $G_j$ . Then all nodes in  $K_x^{j+1}$  have minimum degree in  $G_{j+1}$ .*

**Corollary 1.1** *We can choose a minimum degree node in  $G_{j+1}$  from the set  $K_x^{j+1}$ .*

Because all nodes in  $K_x^{j+1}$  have minimum degree in elimination graph  $G_{j+1}$  we can apply above lemma's again to  $K_x^{j+1}$ . This leads to the next corollary:

**Corollary 1.2** *If  $z$  is a minimum degree node in  $K_x^j$  then this set can be selected next in any order in the minimum degree algorithm.*

### 1.5.1 Mass elimination

Above lemma's and theorem say that if we have found a node  $z$  of minimum degree in elimination graph  $G_i$  and we have found the set  $K_z^i$  then we can eliminate these nodes next in any order. The advantage of this is that we only have to do a degree update after we eliminated the set of nodes. Usually we had to do a degree update every time we eliminated one node. Thus we are saving on the numbers of degree updates. Of course one has to find the set  $K_z^i$  but the cost of finding this set is usually smaller than the costs of doing a degree update every time a node is eliminated. The process of finding such a set  $K_z^i$  if we have found node  $z$  is called mass elimination.

### 1.5.2 Indistinguishable nodes

If nodes are indistinguishable from each other this can save us a lot of work. Since they all have the same degree in the elimination graphs they can be eliminated together as soon as one of them has minimum degree (because then they all have minimum degree). Another advantage of using indistinguishable nodes is that if we perform a degree update we only have to perform it on one representative of the nodes which are indistinguishable from each other. This can save a lot of work, since a degree update is expensive. A set of indistinguishable nodes is also called supernode, because different nodes are treated as one node (you only have to manipulate the representative).

### 1.5.3 Clique representation

Instead of explicit storage of an adjacency list for each node, we can store cliques of the elimination graph in such a way that we can reconstruct the adjacency lists

of the remaining nodes. Initially we start with graph  $G(V, E)$  and the following cliques:  $K_1^0, K_2^0, \dots, K_{|E|}^0$ . Every clique consists of two nodes  $v_i$  and  $v_j$  such that  $(v_i, v_j) \in E$ , and no two cliques are the same. Every time we eliminate a node we merge all the cliques that contain this node together and create a new clique without this node. If at a certain moment we merge cliques  $K_0, \dots, K_p$  such that they all contain the node  $y$  to be eliminated next, then  $y \in \bigcap_{i=0}^p K_i$ . If  $K = \bigcup_{i=0}^p K_i \setminus \{y\}$  then  $\sum_{i=0}^p |K_i| \geq |\bigcup_{i=0}^p K_i| > |K|$ . This means that the amount of storage we will need for our cliques will decrease during execution of the algorithm because we don't have to store cliques  $K_1, \dots, K_p$  anymore. Furthermore, if we want to find the adjacency list of a node  $y$ , then we can calculate it by  $\text{Adj}_G(y) = \bigcup (K_i | y \in K_i)$ .

#### 1.5.4 Element absorption

Another acceleration technique is element absorption. If we work with a clique representation, we can reconstruct the current elimination graph by connecting nodes  $i, j$  if and only if there is a clique  $K$  such that  $i, j \in K$ . If we have cliques  $K_1$  and  $K_2$  such that  $K_2 \subseteq K_1$ , all information from  $K_2$  needed for reconstructing the elimination graph is also contained in  $K_1$  because  $i, j \in K_2 \Rightarrow i, j \in K_1$ . This means that if we want to reconstruct the current elimination graph we don't need clique  $K_2$  anymore. So we can remove  $K_2$ . One can say that clique  $K_1$  absorbed clique  $K_2$ . The advantage of such an action is that it will reduce overhead. This means that if we want to reconstruct an adjacency list of a node in the current elimination graph we probably have to traverse fewer cliques. Element absorption thus can speed up the degree update part in the minimum degree algorithm. For example, if we have cliques  $(1,4), (1,4,3,5), (3,5)$ , then cliques  $(1,4)$  and  $(3,5)$  can be absorbed by clique  $(1,4,3,5)$ .

#### 1.5.5 Multiple elimination

Because a degree update is the most costly part of the minimum degree algorithm, one wants to postpone this work as much as possible. One way of doing this is by multiple elimination. If one eliminates the minimum degree node  $y$  at some stage of the algorithm, then only the degrees of  $\text{Adj}(y)$  can change. Because of that, we can eliminate another node  $x \notin \text{Adj}(y)$  which has the same degree as  $y$ . In general one can find a set  $M$  of nodes that all have minimum degree and are not neighbours of each other, and eliminate them in one step. After that one can perform a degree update on all neighbours of the set  $M$ ; these neighbours are elements of the newly formed cliques. If one uses this technique, the ordering one gets will slightly differ from the genuine minimum degree ordering but it can save some time in the degree update. The gain is in common neighbours of nodes to be eliminated. For example, if we have the elimination graph of figure (1.1) and we eliminate simultaneously nodes 2,3,4,5 and 6 we only have to do one degree update on node 1.



## Chapter 2

# Sequential minimum degree

If we do not take the acceleration techniques into account, at every step of the minimum degree algorithm, a node of minimum degree is eliminated, and its neighbours become a clique afterwards. This means that the data structure has to keep track of the degrees of the remaining nodes. This is only possible if we can reconstruct the adjacency lists of the remaining nodes, whose lists change during the process, because at every step nodes are eliminated and cliques are formed. A naive approach would be to keep track explicitly of the adjacency lists of all the nodes that still have to be eliminated. If node  $y$  is eliminated next, all neighbours of  $y$  add to their adjacency list the list of  $y$ , and then subtract  $y$  itself. So the degrees of the neighbours of  $y$  will change; they are simply recalculated by counting the number of elements in their new adjacency list. The only problem is that we do not know how large these adjacency lists will become. At most we will need  $O(n^2)$  storage capacity if we start with  $n$  nodes. This can happen if the adjacency lists of the remaining nodes grow too fast. So we don't know in advance how much storage capacity we need. We do know an upper bound, namely  $O(n^2)$ . Our desire is to find a data structure which starts with a minimal amount of memory, and which does not need more memory during the execution of the algorithm. Furthermore, the minimum degree algorithm takes  $n$  steps (since we have  $n$  columns). Every step should take a constant amount of time, let's say  $O(c^2)$ . Here  $c$  is the average degree of a vertex of the graph induced by matrix  $A$ . The reason for this is that the algorithm for sparse Choleski factorization takes  $O(d^2n)$  time, with  $d$  the maximum number of non zeros in a column of the Choleski factor. If the preprocessing algorithm takes more time, then there will be hardly any speedup compared to the Choleski factorization without preprocessing. It should be noted however, that the constant factor  $d$  can be much smaller if one executes a preprocessing algorithm first. So it might be better to use a preprocessing algorithm even if it takes more time than the Choleski factorization itself.

We will now give a description in pseudo C of the data structure we want to use in the sequential minimum degree algorithm, using cliques described in the previous chapter. Our data structure consists of two parts:

- part 1: a structure which represents the cliques
- part 2: a structure which keeps track of the degree of the remaining nodes in the elimination graph

## 2.1 Clique structure

Cliques will be created dynamically and they will be arrays of int, in which every integer represents a node. Because we have at most  $|E|$  cliques (we start with  $|E|$  cliques and their number only decreases), we will need at most  $|E|$  variables for dynamically creating cliques. Thus we need an array of pointers to int of length  $|E|$ :

```
int *Cliques[|E|]
```

When we create a new clique  $K$  with  $|K|$  elements, we create a new array with the help of an entry from *Cliques*:

```
Cliques[i]= new(int[|K|])
```

We also keep track of the length of the cliques by using an array:

```
int Length[|E|]
```

In this case  $Length[i]=|K|$ . We will need the length if we traverse a dynamically created clique. It should be noted that the above data structure for creating cliques can give some problems in using computer memory. When we merge cliques  $K_0, \dots, K_{p-1}$  to become a new clique  $K$  then  $|K| \geq |K_i| - 1$  for all  $i \in \{0, \dots, p-1\}$ . So the amount of dynamically allocated memory for clique  $K$  will usually be larger than the amount of memory used by a clique  $K_i$  with  $i \in \{0, \dots, p-1\}$ . Cliques  $K_i, i \in \{0, \dots, p-1\}$ , will be removed and clique  $K$  will be inserted. So one frees the memory used by cliques  $K_i, i \in \{0, \dots, (p-1)\}$ . If this free memory is not reused, then we have to allocate an entirely new part of the memory for inserting clique  $K$ . Because our algorithm creates at most  $n$  new cliques (initially we have  $n$  vertices), the worst case is that we have to allocate  $n$  times an entirely new part of the memory for the new cliques. The question is: how much memory do we have to allocate in the worst case? The new clique created has the size of the minimum degree  $d_{i-1}$  in the current elimination graph  $G_i$ . Since we have at most  $n$  elimination graphs ( $G_0, \dots, G_{n-1}$ ), the amount of memory we have to allocate extra in the worst case will be  $\sum_{i=0}^{n-1} d_i$ . This equals the number of non zeros in the Choleski factor  $L$ .

Furthermore, we have the data structure:

```
structure Place
{
int CliqueName
Place *Next
}

Place *Location[|V|]
```

Index  $i$  in array *Location* matches node  $i$ . *Location*[ $i$ ] is the header of a linked list *Place* which tells us in what cliques node  $i$  is contained in the current elimina-

tion graph. When we eliminate a node, we need all cliques containing that node. This can be done by traversing all cliques to see if they contain this node. But such an operation will cost  $\Omega(n)$  time, which is too much. Instead we use *Location*[*i*] to locate all cliques containing node *i*

## 2.2 Degree structure

The next thing we want, is a data structure which keeps track of the degrees of the nodes in such a way, that we can find the minimum degree node in at most  $O(c^2)$  time. We use an array of length *n* in which index *i* matches node *i*. We use the following declaration for it:

```

structure Vertex
{
int NextNode, PrevNode /* 0 ≤ NextNode,PrevNode < n */
int Degree
int SNL /* SuperNodeList */
}

Vertex Vertices[n]

```

*Vertex*[*i*].*Degree* tells us the degree of node *i* in the elimination graph. If node *i* and node *j* have the same degree then *Vertices*[*i*] and *Vertices*[*j*] are elements of the same cyclic double linked list, induced by integers *NextNode* and *PrevNode*. *SNL* is used when we have indistinguishable nodes or supernodes. Suppose we have a supernode *i*. All indistinguishable nodes *j* in this supernode which are not the representative *i* have instead of the degree in their *Vertices*[*j*].*Degree* the number  $-i$ . It gives us the representative (namely *i*), and tells us that we do not have to update this node because of the negative number. The integer *SNL* is used to create a single linked list of indistinguishable nodes which are contained in one supernode.

We have another array of length *l* (with *l* a parameter), in which every entry *i* contains a node of degree *i* if there is a node left with this degree in the current elimination graph.

```
int Degrees[l]
```

If we expect the degrees of the nodes to be less than  $\frac{1}{2}n$ , then we can choose  $l \leq \frac{1}{2}n$ . If we want to find a node that has degree *i* at that moment we can use array *Degrees* to find such a node in  $O(1)$  time. If we keep track of the minimum degree during the execution of the program we can also find a minimum degree node in  $O(1)$  time. We do this in the following way: If we do a degree update and a node has a smaller degree than the minimum degree then this degree becomes the new minimum degree. This means that if we want to find the minimum degree node we either have it already (using array *Degrees*) or we have to traverse the array *Degrees* from the index of the previous minimum degree up to the next index that is the degree of a node in the current elimination graph. Since the minimum degree can only decrease by one if we eliminate a node, the minimum degree can only decrease *n* times by one during the execution of the algorithm. Since the minimum degree in

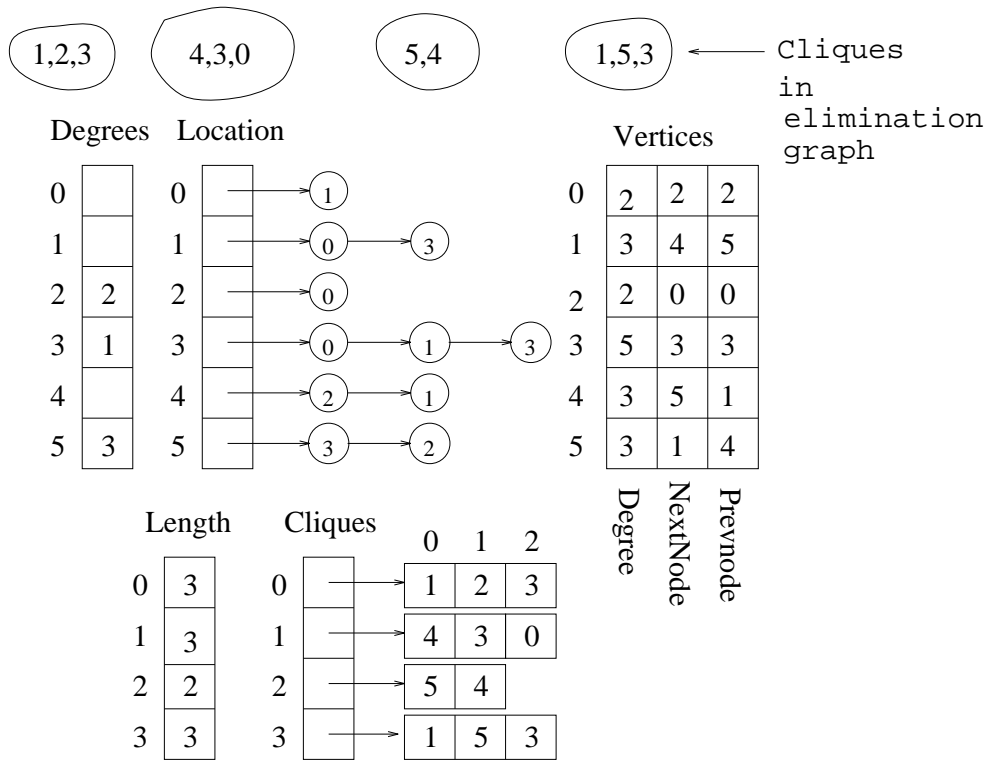


Figure 2.1: Example of data structure representation

$G_{n-1}$  is always zero, the number of times the minimum degree increases by one step is at most  $n$  (what goes up must come down). Therefore the amount of time we need to search for the minimum degree if we only have the previous minimum degree is  $O(n)$  during the execution of the algorithm. This means that we need  $O(n)$  time during the execution of the algorithm to find the minimum degree node. Since we have  $n$  nodes at the start of the program this implies that we need on average  $O(1)$  time to find the minimum degree node in the current elimination graph.

Suppose there are nodes left in the elimination graph which have degree  $d_j$ .  $Degrees[d_j]$  gives us an index  $k$ , such that  $Vertices[k].Degree=d_j$ . If the minimum degree in the elimination graph is  $d_j$ , then we want to eliminate a node with this degree. This means removing the cliques which contain this node and merging these cliques to form a new clique without this node and inserting this new clique into the existing data structure. The node is found using  $Degrees[d_j]$ . If this is node  $k$  then cliques containing  $k$  can be found by using  $Location[k]$  and traversing along the  $Next$  pointers. Figure 2.1 gives an example of the data structure which we just described. For this example we use the cliques:  $(1,2,3)$ ,  $(4,3,0)$ ,  $(5,4)$ ,  $(1,5,3)$

## 2.3 Program functions

We will now give a description of the functions we will use in the sequential minimum degree algorithm. For this we use a kind of pseudo programming language. In all functions we will use the data structure described above. We will use the term supernode frequently. A supernode is a representative of a set of nodes that haven't been eliminated yet and are indistinguishable from each other. The advantage of supernodes is that we only have to manipulate these nodes instead of manipulating the set they represent. Initially all nodes are therefore supernodes because they represent themselves.

First we introduce some notation for these functions. Initially, we start with  $|E|$  cliques, each consisting of two nodes. We therefore introduce the set of cliques:

$$Cliques = \{K_0, K_1, \dots, K_{|E|-1}\}$$

During the execution of the algorithm some cliques can grow in size and others may become empty. For example, if we merge cliques  $K_i$  and  $K_j$ , then our new clique can have cliquename  $K_i$  because we do not need clique  $K_j$  anymore. So we can overwrite the old clique:  $K_i = K_i \cup K_j$  and  $K_j = \emptyset$ . At every stage in the algorithm cliquenames representing the current elimination graph are a subset of the cliquenames of *Cliques*.

Furthermore, we have sets  $Location_i$  which are sets of indices  $\Phi(i, 0), \Phi(i, 1), \dots, \Phi(i, p_i - 1)$  such that  $i \in K_{\Phi(i,0)}, K_{\Phi(i,1)}, \dots, K_{\Phi(i,p_i-1)}$ . If  $i \in K_r$  then  $r \in Location_i$ .

$Degree_i$  is the degree of node  $i$  which is not yet eliminated and which is a supernode.

We will use a couple of basic operations in our minimum degree algorithm. These operations are: Merge, DegreeUpdate, MassElimination, and MultipleElimination.

### 2.3.1 Merge

{ Given a set  $S$  of nodes with minimum degree which are indistinguishable and have to be eliminated. We merge all cliques containing at least one of the nodes of  $S$ .  $V$  is the set of cliques of nodes that need a degree update and  $N$  is the set of nodes that are already eliminated }

$K = \emptyset$

**for all**  $i \in S$  **do**

**for**  $j = 0$  **to**  $p_i - 1$  **do begin**

**if**  $delete_{\Phi(i,j)} = 0$  **then**

$delete_{\Phi(i,j)} = 1$

**if** ( $i$  is the supernode) **then**

$K = K \cup K_{\Phi(i,j)} \setminus S$

$q = \Phi(i, 0)$

{insert  $K$  in one of the cliques  $q$  that have to be removed }

$K_q = K$

**for every**  $j \in K$  **do**

$Location_j = Location_j \cup \{-q\}$

$update_j = \text{True}$

$V = V \cup \{q\}$  {set of cliques of nodes that need a degree update }

$delete_q = 2$  {mark the name of the new clique }

$N = N \cup S$

The variables *update* and *delete* are used for marking nodes and cliques. Because if we want to perform a degree update we only have to update the nodes marked in *update*. We find these nodes using the set  $V$ , which contains cliquenames of newly created cliques whose nodes need a degree update.  $N$  is the set of nodes that have to be eliminated. In creating clique  $K$  we only have to use cliques that contain the supernode. Because merging all cliques that contain at least one node from  $S$  is the same as merging all cliques that contain the supernode from  $S$ . But we also have to mark the cliques that only contain nodes from  $S$  that are not supernodes, because we want to eliminate these cliques too. The variable *delete* marks cliques which have to be removed/emptied. We give  $delete_i$  with  $i = q$  the value 2. This means that  $K_i$  is a new clique and does not have to be removed/emptied. Furthermore, we insert in each set  $Location_j$  of the nodes  $j$  from the new clique an index  $-q$ , where  $K_q$  becomes the new clique. We do this because the new clique  $K_q$  overwrites the old clique  $K_q$ . Some nodes in the new clique already contain this index  $q$  and some do not. Therefore we insert the negative index of the new cliquename. Otherwise we had to check if a set already contained index  $q$  or not. When we perform a degree update we make sure that a set never contains a negative and a positive value of a cliquename. If we have done a couple of Merges on sets of nodes  $S_1, S_2, \dots, S_l$  such that  $S_i \cap S_j = \emptyset$  for all  $i \neq j$  and  $S_i \cap Adj(S_j) = \emptyset$  for  $i \neq j$ , we can do an degree update. We could also update the degrees after each Merge, but since degree updates are expensive we postpone this work as much as possible.

### 2.3.2 DegreeUpdate

In DegreeUpdate we perform a degree update on all supernodes of the newly created cliques. All these cliques are contained in set  $V$ . Some new cliques contain the same nodes. Because of that we use the array *update* to check if we have not already updated the degree. If this is not the case then we check if the node is a supernode. Because if it is not a supernode then we do not have to do a degree update on this node because we update the representative of this node (namely the supernode). If node  $j$  is a supernode, we will check set  $Location_j$ . All cliquenames in this set that are marked nonzero in array *delete* have to be deleted. Because these are the cliques that were used in making new cliques in function Merge. Furthermore, we change the negative integers to positive integers. So if  $Location_j$  contains a negative cliquename  $-i$  then it becomes  $i$ . We do this because in function Merge we inserted the negative cliquename of the newly created clique. After that the degree of supernode  $j$  is the cardinality of the union of all cliques whose cliquenames are contained in set  $Location_j$ . After this we eliminate all cliques which were used in making the new cliques. For this we use the set  $N$ . We want to eliminate all cliques containing at least one of these nodes. These cliques can be found in the sets  $Location_i : i \in N$ . We only delete cliques  $j$  such that  $delete_j=1$ , because if  $delete_j=2$  then we inserted a new clique on that position. After this we can empty  $V$  and  $N$  for the next elimination round.

{  $V$  contains indices of cliques whose supernodes need a degree update.  $N$  is the set of nodes which have to be removed. }

```

for every  $i \in V$  do
  for every  $j \in K_i$  do
    if ( $update_j = True$ ) then
       $update_j = False$ 
      if ( $j$  is a supernode) then
         $Location_j = Location_j \setminus \{l \mid delete_l > 0 \wedge l > 0\}$ 
        change all negative indices to positive in  $Location_j$ 
         $Degree_j = |\bigcup_{z=0}^{p_j-1} K_{\Phi(j,z)}|$ 
for every  $i \in N$  do begin
  for every  $j \in Location_i$  do
    if ( $delete_j = 1$ ) then
       $K_j = \emptyset$ 
       $delete_j = 0$ 
 $N = \emptyset$ 
 $V = \emptyset$ 

```

### 2.3.3 MultipleElimination

{ Let  $i$  be a node of minimum degree  $d_i$ . MultipleElimination will create a set  $M$  of nodes with minimum degree in the current graph such that  $i \in M$  and for all  $q, k \in M : k \notin Adj(q)$  }

```

compatible=True
 $M = \{i\}$ 
for all  $\{j \mid Degree_j = d_i \wedge j \neq i\}$  do
   $q=0$ 
  compatible=True
  while ( $q < p_j$  and compatible=True) do
    {check if the new node contains one of the nodes
    from set  $M$  in its adjacency list}
    if ( $K_{\Phi(j,q)} \cap M \neq \emptyset$ ) then
      compatible=False
       $q = q + 1$ 
  if compatible=True then
     $M = M \cup \{j\}$ 

```

So multiple elimination finds a set of nodes that are not neighbours of each other. This function searches for other nodes that also have minimum degree  $d_i$ . But if we want to find a larger set  $M$  we can introduce a parameter  $\alpha$  such that  $\alpha \geq 1$  and check all nodes that have a degree which is less than  $\alpha d_i$  to form a set  $M$ . Using such a parameter, we can find larger sets  $M$  and thus eliminate more nodes per elimination step. This can increase the speed of our algorithm. However, if we choose  $\alpha$  too large then it can influence the quality of our ordering in a negative way.

### 2.3.4 MassElimination

{  $i$  is a node with minimum degree, and you want to find the neighbours of  $i$  which are indistinguishable from  $i$ .  $S$  is a set of indices, such that  $i \in S$ .  $\text{Adj}(i)$  is the adjacency list of  $i$ . The adjacency list of  $i$  can be reconstructed by taking the union of all cliques in which  $i$  is contained. We can do this by marking the positions of the clique elements in an array. }

```

D = ∅
for  $c_1 = 0$  to  $p_i - 1$  do
    for every  $k \in K_{\Phi(i,c_1)}$  do
        if  $k \notin D$  and  $k$  a supernode then
            Indistinguishable=True
             $c_2 = 0$ 
            while  $c_2 < p_k$  and Indistinguishable=True do
                for every  $q \in K_{\Phi(k,c_2)}$  do
                    if  $q \notin \text{Adj}(i)$  then
                        Indistinguishable=False
                 $c_2 = c_2 + 1$ 
             $D = D \cup \{k\}$ 
        if Indistinguishable=True then
             $S = S \cup \{k\}$ 
             $\text{Degree}_k = -i$ 
            insert  $\text{SNL}$  list  $k$  in  $\text{SNL}$  list from  $i$ 
D = ∅

```

We thus check the adjacency list of node  $i$  to see if its neighbours are indistinguishable from  $i$ . We insert these indistinguishable nodes in set  $S$ . The degree of these nodes will get value  $-i$ . We do this so we know that these nodes are not supernodes and have representative  $i$ . Furthermore, we use a set  $D$  to make sure we do not check the same node twice for indistinguishability.  $D$  can be implemented by using an array of length  $n$  with  $n$  the size of the matrix and marking all indices in this array that are contained in  $D$ .

## 2.4 Alternative: a static data structure

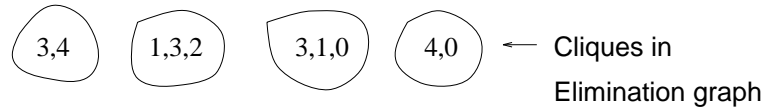
Instead of using the above data structure we can also use a slightly different one to represent cliques. The structures *Vertices* and *Degrees* however stay the same. Instead of using the structures *Location*, *Length* and *Cliques* we can use the following structures:

```

structure Element
{
    int PrevNode, NextNode
    int Node
    int PrevCliq, NextCliq
}

structure Element Cliques[2][ $E$ ]

```



PrevNode	2	4	0	1	3	8	9	5	7	6
NextNode	2	3	0	4	1	7	9	8	5	6
Node	4	1	3	3	2	3	0	1	0	4
PrevCliq	9	7	5	2	4	3	8	1	6	0
NextCliq	9	7	3	5	4	2	8	1	6	0
	0	1	2	3	4	5	6	7	8	9

Figure 2.2: An alternative data structure

```
int Location[|V|]
```

*Clique* is an array of length  $2|E|$ , because initially we have  $|E|$  edges. *Clique*[*i*].*Node* represents a node in the current elimination graph or it is empty. This node is contained in several cliques. These cliques can be found by traversing *PrevCliq* and *NextCliq*, which is a double linked cyclic list. They will point to the same node in another clique. *PrevCliq* and *NextCliq* return an index *j* in array *Clique* such that *Clique*[*i*].*Node* = *Clique*[*j*].*Node*. *PrevNode* and *NextNode* will give us the previous and next node in a clique. They also form a double linked cyclic list. For example, if we have cliques (1,2,3),(4,3),(0,4) and (1,0,3) then array clique can look like figure (2.2). If we know *Clique*[2].*Node*=3, we can find all cliques containing node 3 by *y=Clique*[2].*NextCliq*, and *Clique*[*y*].*NextCliq*. If we want all elements which are also contained in the same clique as *Clique*[2].*Node* then we can find them by *y=Clique*[2].*NextNode*. In this case the clique has two nodes, which are node 3 and *Clique*[*y*].*Node*=4. Thus if we know an index *k* such that *Clique*[*k*].*Node*=*i*, we can find all cliques containing node *i* and all nodes contained in these cliques. We therefore also need an array which gives us such an index. For this we use the array *Location*. *Location*[*i*] gives us an index *k* in array *Clique* such that *Clique*[*k*].*Node*=*i*.

We will now look to the amount of storage both data structures need. Both data structures need an array *Vertices* of length  $|V|$  and an array *Degrees* of length  $|V|$ . Array *Vertices* needs  $4|V|$  of storage because we store 4 items per index in it. Array *Degrees* needs  $|V|$  of storage. Thus a total of  $5|V|$  storage. We now look at the static data structure. For array *Clique* we will need  $5 \times 2 \times |E|$  of storage. For *Location* we need  $|V|$ . Thus a total of  $10|E| + |V| + 5|V|$ . For our previous data structure we have an array *Cliques* of length  $|E|$ , that contains  $|E|$  cliques of size two (so  $2|E|$  storage). Furthermore, we need an array *Location* of size  $|V|$  and a linked list *Place* which contains  $2|E|$  entries in the beginning and of course  $2|E|$  pointers. And we need an array *Length* of size  $|E|$  for storing the length of the cliques. Because we designed our data structure in such a way that the amount of

storage only decreases during the execution of the algorithm, this is the maximum amount of storage we need. Adding this up we have a total of  $7|E| + |V| + 5|V|$ . If we take into account our worst case analysis of the extra use of memory because of the dynamical creation of cliques, then the total amount is:  $7|E| + 6|V| + \sum_{i=0}^{n-1} d_i$ .

The question is: Which data structure is the best to use implementing the minimum degree algorithm? Without the worst case analysis the dynamical data structure will have less storage capacity. But the alternative data structure makes better use of memory. Using the dynamical data structure one has to allocate and free parts of the memory frequently when creating new cliques. These operations cost time. The other data structure does not have this problem because it uses arrays of a certain length which were allocated at the beginning of the program. When we create a new clique using the dynamical data structure we first have to allocate memory for it, before inserting the new clique elements. Furthermore, we have to insert the new cliquename in the array location for every clique element. This means that we have to allocate memory again. If we insert a new clique in the other data structure we only have to insert the new clique elements in the existing array. For every new clique element we have to adapt PrevNode, NextNode, Node, PrevCliq and NextCliq. This means five operations per new clique element. It all depends on how efficient a computer allocates and frees memory. Our feeling is that allocating memory will be much more expensive than manipulation of static arrays. Which means that implementing the minimum degree algorithm using static arrays will probably yield a faster sequential program.

Because we used the sequential programs as a preliminary study for our parallel program we choose for the dynamical data structure. The reason for this choice is as follows: when we build a data structure for the parallel program we want to use parts of the sequential data structure. The easiest thing to do if we have  $p$  processors, would be to divide the sequential data structure over the  $p$  processors. Usually this is not possible. But using the sequential data structure as a starting point for the parallel data structure is a strategy that is used often. The most important thing in the parallel data structure is how to distribute the cliques. If we look at the static data structure, an idea we could derive from it is to allocate arrays at the beginning of the program in all processors for storing cliques. But during the execution of the program cliques can grow in size. Thus the question arises: how large should these arrays be? We know that the amount of storage for the cliques will never exceed  $2|E|$ . But locally cliques or part of cliques can grow in size and can not be stored locally in the processor anymore because the static arrays per processor are too small. A solution for this problem would be to allocate  $2|E|$  memory per processor for the cliques. But then we would allocate globally too much memory compared to the amount of data we have. Another solution would be if a clique or a part of a clique can not be stored locally anymore, to store the rest of it somewhere else (another processor). But this will give rise to complicated storage schemes which in principle could be implemented but one can ask oneself if this can be done efficiently. Because in the dynamical data structure one dynamically creates cliques one does not have the limitation of arrays which are too small. Our feeling is that using this data structure as a starting point for our parallel data structure instead of the static data structure will help us more in building a parallel data structure. For that reason we have implemented the sequential minimum degree algorithm using the dynamical data structure.

## 2.5 Results

We have implemented the above sequential minimum degree algorithm and tested it on several matrices of the Harwell-Boeing collection. We used six different minimum degree programs. The programs differ only in the use of acceleration techniques. All programs use the same dynamic data structure as described above. We used the following programs:

program	mass elimination	multiple elimination	indistinguishable nodes
mdegbas	no	no	no
mdeg4	yes	no	no
mdeg5	yes	yes, with minimum degree	no
mdeg6	yes	yes, with minimum degree	yes
mdeg7	yes	yes, with 2× minimum degree	no
mdeg8	yes	yes, with 2× minimum degree	yes

All programs check if the degree of the minimum degree node is one less than the number of remaining nodes in the elimination graph. If this is the case we do not have to proceed with the algorithm and can select these nodes in an arbitrary sequence.

We tested the programs using the following symmetric matrices from the Harwell-Boeing collection:

matrix	size	non zeros
<i>1138bus</i>	1138	2596
<i>bcsstm13</i>	2003	11973
<i>bcsstk28</i>	4410	111717
<i>lshp3466</i>	3466	91392
<i>bcsstk13</i>	2003	42943
<i>bcsstk18</i>	11948	90346
<i>bcsstk25</i>	15439	172528
<i>bcsstk17</i>	10974	228748

We tested every program on every matrix five times. Every time we permuted the columns (rows) of the input matrix randomly to see if the fill in would change drastically. On the same matrix we used the same permutations for every program. In that way we can compare the different versions of the minimum degree algorithm. Although we permuted the matrices five times for every program, the amount of fill in did not change drastically. So we will only show the minimum time (min time)

used by a program for every matrix with its fill in and also the maximum time (max time). Time is measured in seconds and is the total time for the program to execute. We have tested these programs on a Sun sparc 10 with 32 MB internal memory. For the bigger matrices we could not always calculate the fill in because the computer ran out of memory and our program for calculating the fill in is not memory efficient.

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3260
<i>bcsstm13</i>	15	17	51897
<i>bcsstk28</i>	175	239	368893
<i>lshp3466</i>	15	19	90940
<i>bcsstk13</i>	182	226	275742
<i>bcsstk18</i>	298	704	723171
<i>bcsstk25</i>	1994	2560	--
<i>bcsstk17</i>	823	1086	781604

mdegbas

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3260
<i>bcsstm13</i>	3	4	50710
<i>bcsstk28</i>	23	30	354157
<i>lshp3466</i>	3	4	88420
<i>bcsstk13</i>	20	25	276478
<i>bcsstk18</i>	78	101	723157
<i>bcsstk25</i>	465	633	--
<i>bcsstk17</i>	81	112	752375

mdeg4

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3257
<i>bcsstm13</i>	3	4	51292
<i>bcsstk28</i>	23	29	357262
<i>lshp3466</i>	3	4	91392
<i>bcsstk13</i>	19	25	277868
<i>bcsstk18</i>	52	68	740917
<i>bcsstk25</i>	139	180	--
<i>bcsstk17</i>	74	102	741436

mdeg5

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3257
<i>bcsstm13</i>	4	5	51292
<i>bcsstk28</i>	21	26	358234
<i>lshp3466</i>	3	4	93273
<i>bcsstk13</i>	21	26	277868
<i>bcsstk18</i>	62	83	719540
<i>bcsstk25</i>	153	205	--
<i>bcsstk17</i>	69	94	735808

mdeg6

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3337
<i>bcsstm13</i>	2	3	56716
<i>bcsstk28</i>	22	26	390340
<i>lshp3466</i>	3	4	99458
<i>bcsstk13</i>	16	21	289634
<i>bcsstk18</i>	40	52	768947
<i>bcsstk25</i>	92	124	--
<i>bcsstk17</i>	63	89	821854

mdeg7

matrix	min time	max time	fill in
<i>1138bus</i>	0	0	3344
<i>bcsstm13</i>	3	3	59812
<i>bcsstk28</i>	20	25	390868
<i>lshp3466</i>	3	4	109209
<i>bcsstk13</i>	16	20	343066
<i>bcsstk18</i>	43	60	>823052
<i>bcsstk25</i>	79	105	--
<i>bcsstk17</i>	58	79	909267

mdeg8

We see that the acceleration techniques speed up the process, even on small matrices. But if we look at the different programs using the acceleration techniques we see that on small matrices the differences in time are small. This can be due to the fact that using more acceleration techniques costs time and on small matrices the time saved by these techniques is almost equal to the time using them. On larger matrices the differences in time for different programs are somewhat larger. Here the costs of using acceleration techniques are smaller than the costs of not using them. If we look at the fill in caused by programs mdeg4, mdeg5 and mdeg6, we see that the fill in hardly increases and often decreases compared to the program mdegbas. The programs mdeg7 and mdeg8 do cause an increase in fill of sometimes 10 to 15% compared to mdegbas. This is probably the result of using multiple elimination and selecting nodes that have a degree less than twice the minimum degree. If we look at the minimum time and maximum time used by the program then the difference in time is sometimes 30% (mdeg7, bcsstk17). We also tested the MATLAB minimum degree program on the matrices bcsstk28 and bcsstk25. On matrix bcsstk28 the MATLAB program was almost twice as slow (43 seconds) as our program mdeg4, but on matrix bcsstk25 the MATLAB program was two times faster (49 seconds) than mdeg7. The reason for this may be the fact that the MATLAB program uses more acceleration techniques (it also uses element absorption and incomplete degree update). On relatively small matrices such as bcsstk28 these acceleration techniques take more extra time than the time they save. But on larger matrices such as bcsstk25 these techniques save some time. Furthermore, we implemented the minimum degree algorithm using a dynamical data structure and as said before a static data structure might be more efficient.

We put the times of the different programs in a table and indexed the time of mdegbas as 100. This yields the following table:

matrix	mdegbas	mdeg4	mdeg5	mdeg6	mdeg7	mdeg8
<i>1138bus</i>	--	--	--	--	--	--
<i>bcsstm13</i>	100	20	20	26.9	13.3	20
<i>bcsstk28</i>	100	13.1	13.1	12.0	12.6	11.4
<i>lshp3466</i>	100	20.0	20.0	20.0	20.0	20.0
<i>bcsstk13</i>	100	11.0	10.4	11.5	9.0	9.0
<i>bcsstk18</i>	100	13.7	9.2	10.9	7.0	7.6
<i>bcsstk25</i>	100	23.3	7.0	7.7	4.6	4.0
<i>bcsstk17</i>	100	9.8	9.0	8.4	7.7	7.0
<i>average</i>	100	15.8	12.6	13.9	10.6	11.3

We see that mdeg5 has a lower index than mdeg6 and mdeg7 has a lower index than mdeg8. This means that using indistinguishable nodes the way we implemented it is less efficient. On the whole one can say that acceleration techniques can speed up the process, especially when the problem size is large. The larger the problem the more acceleration techniques one can use to speed up the process. However one has to take into account that certain acceleration techniques create more fill in that slows down the Choleski factorization. Not every acceleration technique creates more fill in. For example mass elimination and multiple elimination with nodes having minimum degree can lead to even less fill in or only a small increase.

## Chapter 3

# Parallel minimum degree

### 3.1 Introduction

We will now describe the data structure for the parallel algorithm. Also we will try to explain why we choose this data structure. If we build a parallel data structure, there are several things that we have to keep in mind. One of them is the size of the data structure. If we have  $p$  processors and our sequential data structure has size  $D$  then we do not want the parallel data structure to have size  $O(pD)$  for the same problems. This is because we want to be able to solve problems of size  $O(pD_{max})$  or  $O(\sqrt{p}D_{max})$  where  $D_{max}$  is the maximum problem size of the sequential algorithm. This will not be possible if the size of the data structure grows linearly with the number of processors. Another aspect is the communication costs. We do not want them to be too high.

Among the objects that the data structure has to store, are the cliques that are created during the execution of the program. There are two ways of storing the cliques: store a clique as a whole on one processor, or scatter the clique over several processors. If we take the first possibility, then we have to ask the question: how can we merge cliques in our parallel computer? What one can do is first merge cliques locally (in a processor), scatter the resulting cliques into subcliques (parts of a clique) over several processors, merge these subcliques locally and merge the resulting subcliques in the processor where you want to store the new clique. This can cause quite a lot of communication. An idea one can extract from this is: why don't we scatter the cliques in the first place and store them in that way? If we merge cliques together, we can merge sets of subcliques, such that subcliques from different sets are mutually disjoint. We can therefore merge these sets of subcliques in parallel, to become subcliques of the new clique. For example, if we have cliques  $a$  and  $b$  and subcliques  $a_0, a_1, a_2, b_0, b_1, b_2$  such that  $a = a_0 \cup a_1 \cup a_2$ ,  $b = b_0 \cup b_1 \cup b_2$ ,  $a_i \cap a_j \neq \emptyset \Rightarrow i = j$ ,  $b_i \cap b_j \neq \emptyset \Rightarrow i = j$  and  $a_i \cap b_j \neq \emptyset \Rightarrow i = j$ . Then  $c = a \cup b = \bigcup_{i=0}^2 (a_i \cup b_i)$ . Let  $c_i = a_i \cup b_i$ . If we allocate  $a_i$  and  $b_i$  to processor  $i \in \{0, 1, 2\}$ , then we can merge  $a$  and  $b$  in parallel such that  $c$  has the same distribution as  $a$  and  $b$  and  $c_i \cap c_j \neq \emptyset \Rightarrow i = j$ , see figure (3.1)

One advantage of this structure is inherited from the sequential case: on a global scale we will never need more memory than we need at the start of the algorithm. Locally however this does not have to be the case. Another advantage is that the

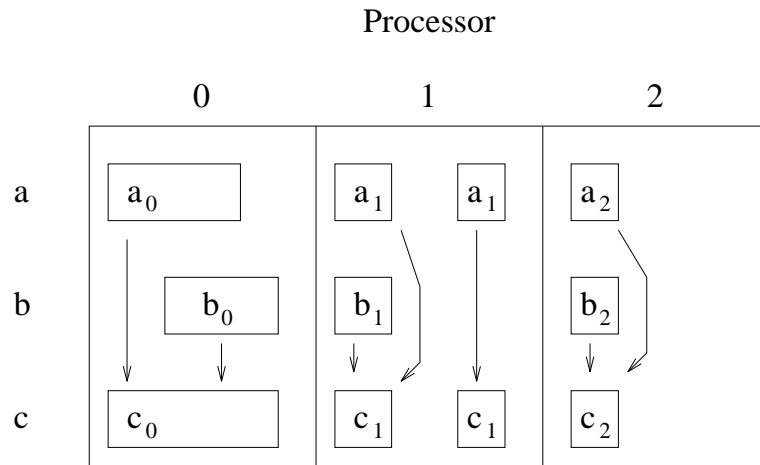


Figure 3.1: Merging disjoint subcliques

new clique has the same distribution as the cliques we merged together. We can generalise this idea. Suppose we have a parallel computer of  $p = NM$  processors ( $N$  processor rows and  $M$  processor columns). We can distribute complete cliques among the processor rows and scatter each clique over the processors in that row in the same way as above: in disjoint subcliques, such that subcliques in the same processor column are mutually disjoint to subcliques in other processor columns. If we merge cliques together, we first merge subcliques in the same processor. After that we merge these newly created subcliques in the same processor column together to become a subclique of the new clique. We will illustrate this idea with an example. Suppose  $N = M = 3$  and we have cliques  $K_0 = (0, 4, 5, 7)$ ,  $K_1 = (6, 10, 4, 8, 9)$ ,  $K_2 = (0, 5, 8, 6)$  and  $K_3 = (11, 2, 6, 5, 4)$ . We distribute these cliques as follows: Clique  $K_i$  is distributed over processor row  $i \bmod N$  and  $x \in K_i$  is put in processor  $p(i \bmod N, x \bmod M)$ . This leads to the distribution of figure (3.2)

		M		
		0	1	2
N	0	(0) (6)	(4,7) (4)	(5) (11,2,5)
	1	(6,9)	(10,4)	(8)
	2	(0,6)		(5,8)

Figure 3.2: Distributing cliques

		M		
		0	1	2
N	0	(0,6)	(4,7)	(11,2,5)
	1	(6,9)	(10,4)	(8)
	2	(0,6)		(5,8)

Figure 3.3: Merging subcliques locally

We see that clique  $K_1 = (6, 10, 4, 8, 9) = (6, 9) \cup (10, 4) \cup (8)$ . If we want to merge the four cliques we first merge the subcliques locally, see figure (3.3).

After that we merge the new subcliques in the same processor column, see figure (3.4), to become subcliques of the clique  $K = K_0 \cup K_1 \cup K_2 \cup K_3$ .

We will use the above idea in distributing our cliques among the processors. Initially we have  $|E|$  cliques of length two (all our edges in the initial elimination graph). These all have different cliquenames  $1, 2, \dots, |E|$ . During the execution of the algorithm some cliques are merged together and old cliques are removed (so their cliquename becomes empty/nil) and new cliques are created. These new cliques will get cliquenames that are not used at that moment. Suppose we have at some moment nonempty cliques  $K_{\Psi(0)}, K_{\Psi(1)}, \dots, K_{\Psi(q-1)}$  such that clique  $K_{\Psi(i)}$  has cliquename  $\Psi(i)$ . Then we will distribute the cliques in the following manner: Let  $x \in K_{\Psi(i)}$  be a clique element (a vertex in the remaining elimination graph). The element  $x$  will be put in processor  $p(\Psi(i) \bmod N, x \bmod M)$ . So a clique is scattered throughout a processor row. It is easy to see that subcliques in different columns are mutually disjoint. This means that if we merge cliques together we only have to merge subcliques in the same processor column and we can do this in parallel. This distribution does not guarantee a good distribution throughout the

		<b>M</b>		
		0	1	2
<b>N</b>	0	(6,0,9)	(4,7,10)	(11,2,5,8)
	1			
	2			

Figure 3.4: Merging subcliques in same processor column

execution of the algorithm, but we expect that on the average it will behave nicely. There are of course many other distributions possible if you have  $p$  processors, but we think at this point that this is the best alternative.

The next question that arises is what is the best choice for  $M$  and  $N$  if  $NM = p$ ? We can choose  $M = N = \sqrt{p}$ ,  $N = 1$  and  $M = p$ ,  $N = p$  and  $M = 1$  or something in between. If we choose  $N = 1$  and  $M = p$  then we only have one processor row. If we merge cliques then this means only merging the subcliques in the processor and this will result in the subcliques of the new clique. The advantage is that we do not have to send subcliques to another row, and the subcliques per processor will be small. If we have more rows, then we first have to send the resulting subcliques in the same column to a processor in that column and merge these subcliques to become a subclique of the new clique (see figure 3.3 and 3.4). If we have more rows we will have less subcliques per processor. But because  $M$  becomes smaller (since  $NM = p$ ) these subcliques will be bigger than if  $N = 1$  and  $M = p$ . Furthermore communication is needed. It looks like  $N = 1$  and  $M = p$  is a good choice if we only consider the merging of cliques. But this does not have to be the best choice for the whole program. We will return to this issue once we have described the parallel data structure to make clear that  $N = 1$  and  $M = p$  does not have to be the best choice.

## 3.2 Data structure

In essence we will use the same data structure as we used in the sequential algorithm. The only difference is that we will now distribute this data structure over  $NM$  processors. Furthermore, we are not only dealing with cliques but also with subcliques. In the sequential algorithm we used an array *Cliques* for storing the remaining cliques during the execution of the algorithm. If we want to store the subcliques in the parallel case then because of the distribution of cliques we are guaranteed to have at most  $\frac{|E|}{N}$  subcliques per processor. But this means we need an array of size  $\frac{|E|}{N}$  per processor to store the subcliques. As a consequence of that we need  $O(MN \frac{|E|}{N}) = O(M|E|)$  storage capacity for storing the subcliques. In

the sequential algorithm we only needed  $O(|E|)$  storage. The amount of storage increases with the number of processors. This is not always bad but it is better to avoid it. Instead of using an array of length  $\frac{|E|}{N}$  per processor for storing the subcliques, we will use an array of length  $\frac{|E|}{MN}$ . This means that it can happen that a processor contains more subcliques than the length of the array. Therefore we use a hashing function on this array to store all subcliques contained by the processor. This means that on the same position in the array we can have more than one subclique of different cliques. We expect that the number of subcliques in the same processor that is stored in the same position of the array will not be large, for two reasons. In the beginning of the algorithm the cliques will be small. This means that if we scatter a clique over a processor row not all processors in that row will contain a subclique of that clique because the clique is small. Furthermore, during the execution the size of the cliques will grow, but also the number of cliques will decrease, which lowers the probability of putting two subcliques in the same position in an array.

We also have to keep track of the degree of the vertices. In the sequential algorithm we used an array *Vertices*[SIZE] with SIZE the size of the matrix. Furthermore we also used an array *Location* for storing the names of cliques in which nodes are contained. Since we have  $NM$  processors we will also distribute these two arrays. We will locally keep track of the minimum degree. Thus processor  $p(s, t)$  knows which node in processor  $p(s, t)$  is the minimum degree node of the set of nodes stored in processor  $p(s, t)$ . This means that we need an array *Degrees* (as in the sequential case) in every processor to keep track of this. *Degrees*[ $i$ ] is a pointer to a linked list of nodes that have degree  $i$ . Since this takes too much storage we will use a smaller array per processor to store this information and we will use hashing on this array to store all information in it.

It is of course possible not to distribute these arrays and put them all on one processor, because updating these arrays is not the most costly part of the algorithm. But the problem we could encounter is that if our input gets too large, there will not be enough memory on one processor to store these three arrays. Again we described two extreme cases; there are many other ways of distributing these arrays, but we will not describe them here. We believe that it is best to distribute data structures and data as much as possible, because if a processor does not have any data then it can not perform any useful work.

As a consequence of what we just said about the data structure, every processor will contain the following data structures:

```

struct Name{
int CliqueName;
int SClength; /* length subclique */
int *SClique; /*subclique itself */
struct Name *Next;
}

```

This structure is used for creating a linked list and putting subcliques into the hash array *Cliques*. Because we can have more than one subclique at the same

place in the array we will use a pointer structure for this. The subcliques are put into the following array:

```
struct Name *Cliques[ $\lceil \frac{|E|}{p} \rceil$ ];
```

Here,  $|E|$  is the number of edges in the original graph and  $p$  is the number of processors. Subcliques are put in the array *Cliques* as follows: If we have a clique  $i$  then this clique will be put in processor row  $i \bmod N$  and subcliques of clique  $i$  will be put in an array *Cliques* on position  $((i) \bmod N) \bmod \lceil \frac{|E|}{p} \rceil$ .

Furthermore, we have three arrays that are also used in the sequential algorithm:

```
struct Place *Location[ $\lceil \frac{|V|}{M} \rceil$ ];
```

```
struct DegLoc{
int Degree, Node
struct DegLoc *Next
}
```

```
struct DegLoc *Degrees[ $\lceil \frac{|V|}{p} \rceil$ ];
```

```
struct Vertex Vertices[ $\lceil \frac{|V|}{p} \rceil$ ];
```

These arrays are almost the same as described in the sequential case only now we have distributed them. In the sequential program *Location*[ $i$ ] contained the names of cliques that contained node  $i$ . In our parallel program this data is distributed among the processors in processor column  $i \bmod M$ . The cliquenames of cliques that contain node  $i$  are distributed in the following way: If  $i \in K_j$  then cliquename  $j$  is stored in processor  $(j \bmod N, i \bmod M)$  in the list *Location*[ $i \bmod M$ ]. This means that globally we need  $O(NM \frac{|V|}{M})$  storage capacity for storing the cliquenames. This is  $N$  times more than in the sequential case. But in our parallel program we have to send cliquenames to other processors. This is because every processor has to know which subcliques it has to merge. Since we distributed the cliquenames, we can do this in parallel. Otherwise we had to send these cliquenames from one processor which could result in an expensive  $h$ -relation. We had to make a choice between memory and speed. Node  $i$  is stored in processor  $i \bmod p$  on position  $i \bmod p$  in array *Vertices* which locally keeps track of the degrees of the nodes. *Degrees*[ $i$ ] in our sequential program contained a node of degree  $i$  if there were any left in the elimination graph. *Degrees*[ $i$ ] in processor  $q$  contains nodes  $j$  from the current elimination graph such that  $j \bmod p = q$  and the degree of node  $j$  modulo  $\frac{|V|}{p}$  equals  $i$ . Array *Degrees* is a hash array and *Degrees*[ $i$ ] contains a list of nodes.

### 3.3 Program functions

The basic minimum degree algorithm consists of four steps:

1. Choose minimum degree node
2. Merge cliques containing this node
3. Update nodes that need it

#### 4. Remove minimum degree node

We will give a description of these steps and a time analysis in terms of the BSP parameters. For further information about the BSP model see([3], [11])

### 3.3.1 Choose minimum degree node

In the sequential algorithm we could find the minimum degree node in  $O(1)$  time because we only used one processor. If we put all information about the nodes stored in arrays *Degree* and *Vertices* on one processor we could also do this in the parallel case. But we distributed this information, so we first have to determine the minimum degree node locally. After that we broadcast this degree and the node to all processors in the same row. Then we choose the minimum degree node from this set and broadcast it to processor in the same column. After this every processor knows the minimum degree and the minimum degree node. This means that we have five steps in determining the minimum degree:

1. Search minimum degree locally
2. Broadcast it to processors in the same processor row
3. Choose minimum degree per row
4. Broadcast it to processors in the same processor column
5. Choose minimum degree from the values broadcast in step 4

Time analysis:

Step 1 takes 1 time step. Broadcasting it to  $M - 1$  processors is a  $2(M - 1)$  relation, because we send the minimum degree node and its degree. So step 2 takes  $2(M - 1)g$  time. Then we have to check  $M$  minimum degree candidates. Thus step 3 takes  $2M$  operations. Step 4 is a  $2(N - 1)$  relation and takes  $2(N - 1)g$  time. Step 5 takes  $2N$  steps using the same argument as step 3. Furthermore, we have synchronisation after step 2 step 4 and step 5. So the total amount of time for finding the minimum degree is:

$$1 + 2(M - 1)g + 2M + 2(N - 1)g + 2N + 3l = (2N + 2M + 1) + 2(M + N - 2)g + 3l$$

This is quite a lot more time than in the sequential case. But this step is not the most time consuming step in our algorithm. So we allow this step to be more costly than in the sequential case.

### 3.3.2 Merging cliques

After we have found a minimum degree node, we have to merge all cliques containing this node to create a new clique without this node. Because of our distribution this means merging the subcliques in the same processor column. First we have to broadcast the names of the cliques we want to merge to all processors of the responsible processor row. If one of these cliquenames is  $i$  then all processors in row  $i \bmod N$  need to know this cliquename to see if they contain a subclique of it. After that, processors that contain subcliques of cliques to be merged, first merge subcliques locally and then send the result to one responsible processor in the same column. This processor merges all subcliques. Thus we have the following steps for

merging cliques:

1. Broadcast cliquenames to the processor rows
2. Check if the processor has any subcliques of these cliques and merge them locally
3. Send these results to one processor in the column
4. Merge these results received by a processor and insert the new subcliques

### 3.3.3 Merge

{ $k$  is the minimum degree node and  $p(s, t)$  is the processor number}

superstep(0)

**if** ( $t = k \bmod M$ ) **then**

Put a cliquename from set  $Location_{(k)div(M)}$   
into set  $Name$  of processors  $p(*, t) \neq p(s, t)$   
If  $Location_{(k)div(M)}$  is empty  
then send a  $-1$ .

**for all**  $\Psi(i) \in Location_{(k)div(M)}$ ,  $i = \{0, 1, \dots, |Location_{(k)div(M)}| - 1\}$   
Put  $\Psi(i)$  into set  $Names1$  in processor  $p(s, i \bmod M)$

superstep(1)

**if** ( $t = k \bmod M$ ) **then**

$ncn$  = smallest positive number in set  $Name$   
 $Name = \emptyset$   
Put  $ncn$  into  $ncn$  in processors  $p(s, *) \neq p(s, t)$

Put all cliquenames in set  $Names1$  into set  $Names2$   
in processors  $p(s, *) \neq p(s, t)$

superstep(2)

$SC = \emptyset$

**for each** ( $l \in Names1 \cup Names2$ )

Check if list  $Cliques_{((l)div(N)) \bmod (\lceil \frac{|E|}{p} \rceil)}$   
contains a subclique  $SC_l$  of clique  $l$   
 $SC = SC \cup SC_l$   
Remove  $SC_l$  from list  $Cliques_{((l)div(N)) \bmod (\lceil \frac{|E|}{p} \rceil)}$   
Put  $SC$  into  $SC_s$  in processor  $(ncn \bmod N, t)$

superstep(3)

**if**  $s = ncn \bmod N$  **then**

$SC = \bigcup_{i=0}^{N-1} SC_i$ ,  $i \neq s$   
Insert  $SC$  as subclique of clique  $ncn$  in list  
 $Cliques_{((ncn)div(N)) \bmod (\lceil \frac{|E|}{p} \rceil)}$

Time analysis:

We will now give a time analysis of the merging of cliques using BSP parameters. First we will make some assumptions to simplify the analysis. Suppose we have a node  $k$  and  $d$  cliques of size  $c$  that contain this node. We assume that the cliques are equally distributed among the processor rows and each processor has an equal part of all cliques in his processor row. Thus every processor contains  $\frac{d}{N}$  subcliques of size  $\frac{c}{M}$ . Note that this is a very ideal case. We want to eliminate node  $k$  and merge these cliques into a new clique  $K$  such that  $k \notin K$ . What will this cost us?

In superstep 0 we send a cliquename or a -1 (a nil element) from processors  $p(s, k \bmod N)$  to other processors in that processor column because we have to determine the new cliquename. This is a  $(N - 1)$ -relation. The information which cliques contain node  $k$  is located in processors  $p(*, k \bmod M)$  in array *Location*. These  $d$  cliquenames are equally distributed among these processors and we send them to processors in the same processor row. This is a  $\frac{d}{N}$ -relation.

In superstep 1 we choose a cliquename for the new clique from a set of  $N$  members. This costs us  $N$  time. We also send this new cliquename to processors not located in processor column  $k \bmod N$ . Every processor in this processor column sends it to all processors in its own processor row. This is a  $M - 1$ -relation. Furthermore, we send all cliquenames we received to all other processors in that processor row. This is the second phase of a two phase broadcast (see [2] p. 6). Every processor sends  $\frac{d}{NM}$  cliquenames and receives  $\frac{d}{NM}(M - 1)$  cliquenames. Thus a  $\frac{d}{NM}(M - 1)$ -relation.

In superstep 2 every processor has received the proper cliquenames and will look into its hash array *Cliques* to see if it contains subcliques of these cliques. This costs  $\frac{d}{N}O(1)$  time, since we do not expect more than  $O(1)$  subcliques per entry in hash array *Cliques*. For merging the subcliques locally we use an array for marking and unmarking the elements. Because we have  $\frac{d}{N}$  subcliques per processor of size  $\frac{c}{M}$  merging them locally costs at most  $2\frac{cd}{MN}$  time. Furthermore we send the result to the responsible processor in the processor column. Since the result can have at most size  $\frac{c}{M}$  this will be a  $(N - 1)\frac{c}{M}$  relation.

In superstep 3 we will merge these results using an array for marking and unmarking the clique elements. This will cost us at most  $2N\frac{c}{M}$  time. Furthermore we use  $4l$  synchronisation. This results in the following equation in BSP parameters for the time of a merge operation:

$$(N + M + 2\frac{d}{N} + \frac{N}{M}c - 2 - \frac{d}{NM} - \frac{c}{M})g + N + \frac{d}{N}O(1) + 2\frac{cd}{MN} + 2\frac{N}{M}c + 4l \quad (3.1)$$

If we would merge these cliques sequentially then we would use  $d$  steps to find the cliquenames and  $dc$  time to mark the elements in an array to determine the length and the elements of the new clique. After that we need at most  $dc$  steps to unmark the array. We thus need a total of  $2dc + d$  time. If we look at equation (3.1) we see that if  $dc$  is relatively small compared to  $N$  and  $M$ ,  $(N - 1 + M - 1)g$ ,  $N$  and  $4l$  will become a substantial part of the total time of the parallel merge. Chances of getting any speed up will then be small. Furthermore there will be no speed up

if  $\frac{cN}{M}g \geq 2dc$  or  $\frac{2dg}{N} \geq 2dc$ . If  $M = N = \sqrt{p}$  then  $g \leq 2d$  and  $g \leq c\sqrt{p}$  is needed for any speed up. Above inequalities and equations show that if we want any speed up,  $d$  and  $c$  must not be small. This can be a problem at the start of the program. Since the size of the cliques at the beginning of the program is two and  $d$  equals the degree of the minimum degree node. This means that the parallel program will probably run slower in the beginning compared to the sequential program. But can the parallel program make up for this lost time? This will only happen if  $d$  and  $c$  will be larger during the during the rest of the program. Furthermore, the assumptions we made are very ideal. This means that in reality things will only be worse in the parallel case.

### 3.3.4 Updating nodes

We have merged cliques containing a node  $k$  to become a new clique  $K$ . The degrees of nodes in clique  $K$  may have changed. Thus we have to calculate the new degree of these nodes. When one wants to determine the degree of a node  $j$  then this is the cardinality of the union of all cliques containing  $j$  minus 1. Thus in determining the degree of node  $j$  we can merge all cliques containing node  $j$  in the same way as we did in function Merge. But this means sending subcliques to other processors which will result in a lot of communication.

Is it possible to avoid sending entire subcliques in full integer representation to other processors and still be able to calculate the degree of node  $j$ ? We can do the following: first one locally merges the subcliques of cliques containing  $j$ . Then one codes the resulting subcliques to one number. If we have processor  $p(s, t)$  then a subclique in processor  $p(s, t)$  can only contain the elements  $t, t + M, t + 2M, \dots, t + \lfloor \frac{V-t}{M} \rfloor M$ . We code the resulting subclique into a 32 bit binary number in the following way: If  $x$  is an element from a resulting subclique in processor  $p(s, t)$  the bitnumber  $(x \text{div} M) \bmod 32$  becomes a 1. After this we send the integer value of the binary number to all other processors in the processor column. All processor compare the binary representation of their own number with the ones they received. If their binary number has a one on a certain position and other binary numbers have a zero on that position, then this processor is the only processor in that column that contains numbers in the resulting subclique that are coded to that position. This means that processors in the same column can find out which elements of their subclique are unique. These numbers they do not have to send to the responsible processor in the column since they are unique. They thus send the amount of unique numbers and the numbers that are not unique. The numbers that are not unique are merged together and their number is added to the number of unique numbers. If one adds up this value from different columns minus one then one gets the degree of  $j$ . The advantage of such a technique is that one does not have to send entire subcliques to other processors. Thus one expects that the amount of communication decreases. A drawback of this technique is that we need an extra superstep for sending the coded numbers to other processors.

We compared this method (method 2) with the conventional method of sending entire subcliques to other processors (method 1). We used one processor column. One processor generated for every processor in the column a random subclique of elements between 0 and 999. The elements were chosen with a certain probability  $t$  ( $0 < t \leq 1$ ). After that we let method 1 and method 2 calculate the cardinality

of the union of these cliques (subcliques if we have more columns). We timed both methods to see which one would be faster. Furthermore we calculated the size of the  $h$ -relations and the number assignments. We tested these methods on a cluster of Sun 4/20 workstations using an Ethernet. We used different numbers of processors varying from 4 to 8 and we tested it 40 times. It turned out that method 2 was always faster than method 1. Usually three to four times faster. Even when the size of the  $h$ -relations and the number of assignments were larger than method 1. This is very strange because using the BSP model one would conclude that method 1 would be faster if it has lower  $h$ -relation and less assignments than method 2. We also tested the extreme case:  $t = 1$ . Thus not a single element is unique. Clearly method 2 has higher  $h$ -relations because it has to send coded numbers to other processors and check out which elements are unique. Besides this extra work method 2 has to do the same amount of work as method 1. Even in this extreme case method 2 was three to four times faster. One can say that a cluster of workstations is not a very good BSP computer which is true. But still the results are somewhat strange. We also tested both methods on a cray T3D and still method 2 was three to four times faster, even in the extreme case that every processor has a subclique of size 1000 and no clique element in a subclique of a processor is unique.

Experiments show that method 2 is about four times faster than method 1. If one would analyse this with the BSP model or any other model one would not come to this conclusion. Why does this happen? Because of the strange results we got we used method 1 in our degree update. We perform a degree update on nodes of the newly created clique. This clique is located in a processor row.

When we update the degrees we first calculate the size of the clique because every processor has to know how many nodes will get a degree update. We also calculate the subsize which we use in determining the order nodes will get a degree update. When we do a degree update we first calculate the subdegrees per column the sum of these minus one is the new degree. We thus have to do the following steps:

- 1 calculate the size of the clique
- 2 calculate the subsize of the clique
- 3 send size to all processors
- 4 **for**  $i = 0$   $i < \text{size} - 1$  **do**
  - Update a node  $j$  from the clique
  - Check which cliquenames ought to be removed from sets *Location* of node  $j$
  - Merge remaining cliques
  - Calculate subdegree
  - Send subdegrees to one processor
  - Calculate new degree
  - Insert new degree into data structure

### 3.3.5 Degree update

{  $ncn$  is the new cliquename,  $K_t$  is a subclique of the new clique  $K$  in processor  $(ncn \bmod N, t)$ ,  $\text{length}_{K_t}$  is the length of this subclique.  $names1$  and  $names2$  are empty. All nodes in clique  $K$  will get a degree update.}

superstep(0)

**if**  $s = ncn \bmod N$  **then**  
     put  $\text{length}_{K_t}$  in  $\text{length}_{K_t}$  in processor  $p(s, *)$

superstep(1)

sublength=0;  
**if**  $s = ncn \bmod N$  **then**  
      $\text{length} = \sum_{i=0}^{M-1} \text{length}_{K_i}$  /\* size of the clique \*/  
     /\* number of nodes in other processors that will first \*/  
     /\* get a degree update \*/  
      $\text{sublength} = \sum_{i=0}^{t-1} \text{length}_{K_i}$   
     put  $\text{length}$  in processor  $p(*, t)$   
     cand=-1

j=0

**for**  $i = 0$  to  $i < \text{length}$  **do**

    superstep(2)

**if**  $s = ncn \bmod N$  and  $i \geq \text{sublength}$  and  $i < \text{sublength} + \text{length}_{K_t}$  **then**  
             node= $K_t[j]$   
             put node in cand in processor  $p(*, \text{node} \bmod M)$   
             put node in cand in processor  $(\text{node} \bmod p)$   
             j=j+1

superstep(3)

**if** cand  $\neq -1$  and  $t = \text{cand} \bmod M$  **then**  
      $\text{Location}_{\text{cand} \bmod M} = \text{Location}_{\text{cand} \bmod M} \setminus \text{Cliques used for creating } K$   
     **for all**  $\Phi(i) \in \text{Location}_{\text{cand} \bmod M}$   
         put  $\Phi(i)$  into set  $names1$  in processor  $p(s, i \bmod M)$

superstep(4)

    Put all cliquenames in set  $Names1$  into set  $Names2$   
     in processors  $p(s, *) \neq p(s, t)$

superstep(5)

$SC = \emptyset$   
     **for each**  $(l \in Names1 \cup Names2)$   
         Check if list  $\text{Cliques}_{((l \bmod N) \bmod (\lceil \frac{N}{p} \rceil))}$   
         contains a subclique  $SC_l$  of clique  $l$   
          $SC = SC \cup SC_l$   
     Put  $SC$  into  $SC_s$  in processor  $p(ncn \bmod N, t)$

```

superstep(6)
  if  $s = ncn \bmod N$  then
     $SC = \bigcup_{i=0}^{N-1} SC_i, i \neq s$ 
     $\text{subdegree}_t = |SC|$ 
    put  $\text{subdegree}_t$  in  $\text{subdegree}_t$  in processor  $p(s, 0)$ 

superstep(7)
  if  $s = ncn \bmod N$  and  $t = 0$  then
     $\text{degree} = (\sum_{i=0}^M \text{subdegree}_i) - 1$ 

superstep(8)
  if  $\text{cand} \neq 0$  and  $p(s, t) = \text{cand} \bmod M$  then
    get degree from processor  $p(ncn \bmod N, 0)$ 

superstep(9)
  if  $\text{cand} \neq 0$  and  $p(s, t) = \text{cand} \bmod M$  then
    insert new degree
   $\text{cand} = -1$ 

```

Time analysis:

We will now give a time analysis of the function degree update. We will make the same assumptions as we did with the function Merge. Furthermore, we start with a clique of size  $c$  whose nodes will get a degree update.

In superstep(0) we send the lengths of subcliques of  $K$  to all processors in that processor row. This is a  $M - 1$  relation.

In superstep(1) we calculate the size of  $K$  en a subsize used for the degree updates. The size we send to all processors in the processor column. Thus we have  $2(M - 1)$  time for calculating the sizes and we have a  $N - 1$  relation.

In superstep(2) we send a node to a processor column and to a processor. Thus this is a  $N$  relation.

In superstep(3) we remove cliques we do not need anymore and the result we send to processors in the processor row. Using the assumptions this costs  $\frac{d}{N}$  time and a  $\frac{d}{N}$  relation. Using the same arguments and assumptions it is easy to see that superstep(4) is a  $\frac{d}{N} - \frac{d}{NM}$  relation.

Using the same arguments as in superstep(2) and superstep(3) of function Merge Superstep(5) costs:  $2\frac{cd}{MN} + \frac{d}{N}O(1) + (N - 1)\frac{c}{M}g$  time. Superstep(6) costs:  $2N\frac{c}{M}$  time and we have a  $M - 1$  relation because we send  $M - 1$  subdegrees to one processor.

In superstep(7) we add up  $M$  subdegrees thus this costs  $M$  time. Then the responsible processor gets the degree from this processor and inserts it if necessary into the data structure. Thus superstep(8) is a 1 relation and superstep(9) costs  $O(1)$  time.

For the time of a degree update for one node we have to divide the time of superstep 0 en 1 because these are called once during the degree update of a clique of size  $c$ . This results in the following formula:

$$\begin{aligned} & \left( \frac{N-1}{c} + \frac{M-1}{c} + M + N + 2\frac{d}{N} + N\frac{c}{M} - \frac{d}{NM} - \frac{c}{M} \right)g \\ & + 2\frac{M-1}{c}M + \frac{d}{N} + 2\frac{cd}{MN} + \frac{d}{N}O(1) + 2N\frac{c}{M} + O(1) - 2 + 7l + \frac{2}{c}l \end{aligned} \quad (3.2)$$

We see that part of the formula looks like the cost formula from the merge function. This is not strange since we use a merge to determine the degree of the nodes. If we do a sequential degree update on node  $i$  then we merge cliques that contain this node and the size of this new clique minus one is the new degree. Furthermore we need  $O(1)$  time to insert the new degree. Thus the time for a degree update is the time for a merge plus  $O(1)$ :  $2cd + d + O(1)$ . We see that a parallel degree update costs more than a parallel merge, but the sequential degree update only costs  $O(1)$  time more than the sequential merge. Thus the problems we had with the parallel merge compared with the sequential merge we also have with the parallel degree update compared with the sequential degree update.

### 3.3.6 Remove minimum degree node

After we have done a degree update on nodes of the new clique we remove the minimum degree node  $k$ . This means we remove the minimum degree node from array *Vertices* in processor  $k \bmod p$  and choose a new local minimum degree node in this processor. Furthermore we eliminate the cliquenames of cliques that contained node  $k$  in array *Location* and we reset array *cliquecount* in which we marked the cliquenames of cliques containing node  $k$ . This means we have to do the following steps:

- 1 eliminate the minimum degree node locally
- 2 chose new minimum degree node locally
- 3 remove cliquenames of minimum degree node and reset array *cliquecount*
- 4 insert minimum degree node in the ordering created so far

step 1,2 and 4 takes a total of  $O(1)$  time since we know the position of node  $k$  in array *Vertices* and we locally keep track of the minimum degree node in the same way as we did in the sequential case. If we use the same assumptions about the cliques and the cliquenames as we did with function *merge* step 3 takes  $\frac{d}{N}g$  time to reset array *cliquecount* and  $\frac{d}{N}$  time to remove the cliquenames of minimum degree node  $k$ . Furthermore we synchronise after step 2 and step 4. Thus the total time for this function is:

$$O(1) + \frac{d}{N}g + \frac{d}{N} + 2l \quad (3.3)$$

## 3.4 Results

We tested the parallel program (using no acceleration techniques) on the matrix *bcsttk17* and used several types of row-column configurations. The overall time of the program and the time used by supersteps separately was measured. Furthermore, we tested the parallel program on one processor on several matrices to see

how it would perform compared to the sequential program. A Cray T3D was used for our measurements of the parallel program. The comparison with the sequential program was done on a Sun sparc 10 with 32 MB internal memory. Time is measured in seconds and we got the following results for our program:

matrix	time
1138bus	0
bcsstm13	30
bcsstk28	223
lshp3466	15
bcsstk13	182
bcsstk18	567
bcsstk25	1992
bcsstk17	822

Parallel program tested on one processor on different matrices

If we compare these results with the basic sequential minimum degree program we see that in the worst case the parallel program is twice as slow (bcsstm13 and bcsstk18) but on other matrices it is just as fast (bcsstk17 and bcsstk25).

$(row) \times (column)$	time
$1 \times 1$	695
$2 \times 2$	729
$4 \times 4$	589
$2 \times 8$	565
$8 \times 2$	706
$1 \times 16$	732
$1 \times 4$	696

Different configurations tested on a Cray T3D

There is some decrease in time if we used a larger number of processors but this decrease is not significant compared to the number of processors we used. If one uses 16 processors one would expect that the program runs at least  $\sqrt{16} = 4$  times faster which is clearly not the case.

Parts of the program that can be done in parallel are found in functions Merge and DegreeUpdate. These are superstep 14,15,16 and 17 (Merge) and 20,21,...30 (DegreeUpdate) in our parallel program (see Appendix B). Since a merge occurs at most 10018 times the amount of time used by the function Merge is not very large. We will concentrate our discussion on the function DegreeUpdate. The total time used by function DegreeUpdate on a  $4 \times 4$  configuration is 516.88 seconds. This is 87% of the total time. Just like in the sequential case the degree update is the most time consuming part of the program. The supersteps in function DegreeUpdate are called at least 1,000,000 times for every configuration we used. This means that although function DegreeUpdate consumes most of the time during the execution of the program, the average time of a superstep in function DegreeUpdate is relatively small. Most time is spent in superstep 26. This is the superstep in which subcliques are merged together in one processor and sent to a processor in the same column. On a  $1 \times 1$  configuration it takes 0.000274 seconds and on a  $2 \times 2$  configuration it

takes 0.000216 seconds.

Why does the algorithm not run faster on more processors than on one processor? To answer this question one has to look at the times of the separate supersteps. In our program we have 25 supersteps. These supersteps have a number between 1 and 31. We measured the total time of each superstep, the average time and the number of times it was called. Superstep 3,4 and 5 are used for reading the non zeros (cliques of size two) and putting them in the data structure in the correct processor. The times for this differ from 28.4 seconds on a  $1 \times 1$  configuration to 33.5 seconds on a  $1 \times 16$  configuration. This difference is not very large compared to the total time. It is understandable that it takes more time if we use more than one processor since we also have communication. If we do not take the reading of the data into account superstep 13 has the highest average time. The differences of time of this superstep on different configurations are small. This is not strange since in superstep 13 only processor 0 performs a task. The times for superstep 13 differ from 28.8 on a  $4 \times 4$  configuration to 32.7 on a  $2 \times 2$  configuration. It is understandable that above supersteps more or less use the same amount of time on different configurations, since it is difficult to perform these supersteps in parallel.

If we look at the total times of the program on different configurations, the difference is at most 23%. Part of this differences originate in the fact that if we use different number of processors we can get a different ordering. The reason for this is that if we have  $p$  processors we choose the new minimum degree node from  $p$  candidates. If more than one has minimum degree then this choice does not have to be unique. One ordering can be harder to calculate than an other. Thus the differences in total time are not very large. This means that the time saved by doing some things of the algorithm in parallel is lost by time spent on communication and synchronisation that are needed in a parallel program.

So the question remains: How can we decrease the  $h$ -relations and the volume of communication? Large parts of communication involve sending cliquenames and subcliques to processors. If one wants to decrease communication of cliquenames then this means storing more information locally. This means a larger data structure. The communication of subcliques is more or less a consequence of the choice of our data structure.

In our theoretical time analysis we already saw that there probably would not be a large speed up. So one can ask the question: Is there a better data structure? If one wants to find such a data structure one has to search for a structure that can perform a degree update very efficiently. Furthermore, one can perform a degree update for different nodes at the same time. The problem with this is that some nodes can be contained in the same cliques.

The input of the program is very irregular. This means that if two input graphs have the same number of nodes the amount of edges and the way they are connected can differ. As a consequence of that the distribution of every graph is different and thus also the volume and pattern of communication. Furthermore the information that is communicated is used in the next superstep. This means that if we could use the new BSP library we do not have to allocate static arrays for information of

other processors. This would make the program somewhat faster. But the problem lies in the data structure and the amount of communication. One wants a data structure that does not have to send entire subcliques to other processors if we do a degree update on a node.



## Chapter 4

# Conclusions/Future work

Using the BSP model we saw that the functions merge and degree update would cost a lot of time. We also saw the cause of it: too much communication. Especially many-to-one communications. Thus if we want to improve the algorithm then we have to lower the number of many-to-one communications. One also could see from the experimental results that communication is a bottleneck. The times on one processor were just as fast as times on more processors which suggests that communication is responsible for it. It also shows that the BSP model is a good model in explaining the (disappointing) parallel results one gets.

The results of the parallel minimum degree program are not very promising. Nevertheless we believe that some things can be improved. Especially if we use the new BSP library (see [6]). The advantage of the new library is that we do not have to allocate static arrays anymore for communication when we use the communicated data the next step. This is very useful since in our program the communication is variable and so is the number of processors that send data to other processors. This is due to the fact that our input is a graph with  $|V|$  nodes and  $|E|$  edges and we do not know if there is any structure in it.

The problem of the basic parallel minimum degree algorithm is the amount of communication and its structure. In functions Merge and DegreeUpdate we have a large many-to-one communication, sending subcliques to one processor in a processor column. After that only processors in that row do something. The other  $NM - M$  processors are idle. A solution for this problem is multiple elimination. With this we change the many-to-one communication to a many-to-many communication and the next step all processors have to do something useful.

If we have a set  $S$  of  $N$  nodes for multiple elimination we can thus make  $N$  new cliques in  $N$  processor rows. At the moment that all processors have the cliquenames that will be used for merging, every processor will look in its hash array *Cliques* to see if it contains subcliques of cliques that contain a node from  $S$ . Every processor makes  $N$  subcliques and sends these subcliques to the responsible processor in its processor column. Thus every processor sends  $N - 1$  subcliques and receives  $N - 1$  subcliques. After that we can create  $N$  new cliques and delete  $N$  nodes at once. Using this we do not have a many-to-one communication anymore and none of the processors is idle after this communication step.

If multiple elimination saves some time then this will not be enough since almost 85% of the time of the minimum degree program is used by function `DegreeUpdate`. Thus we also have to lower the time used by function `DegreeUpdate`. We can decrease the number of degree updates by using mass elimination. If we have a set  $S$  of nodes for mass elimination we can merge all cliques containing nodes from  $S$ .

If we merge cliques or do an degree update it often happens that we have cliquenames in a processor that has to send this information to all processors in its processor row so that every processor can look if it contains a subclique of these cliques. Thus one always sends this information to all processors in the processor row even if only two processors contain a subclique of this clique. What one can do is to introduce a master for every clique. If we send cliquenames this information is only relevant for processors that contain subcliques of these cliques. When we send cliquenames from lists from arrays *Location* we first send the cliquename to the processor that is the master of this clique. This processor knows which processors contain a subclique of this clique and only sends the cliquename to these processors. If we distribute the masters we will get a better load balance.

Furthermore we described two methods for doing a degree update. It is possible to use the second method even when it is not clear why it is faster than the first method. All these improvements can lead to a faster minimum degree algorithm. If we can use the new BSP library this will also lead to less static arrays for communication.

If we want to improve the parallel minimum degree program we have to implement multiple elimination and mass elimination. Furthermore, we can implement the second method for communication of subcliques and we can use the idea of masters for cliques. To implement it efficiently we will have to use the new BSP library.

## Appendix A

# The sequential program

We will now present a listing of the sequential minimum degree program that we used. The program uses the acceleration techniques mass elimination and multiple elimination and uses the datastructure described above. The input is standard input and the output is standard output. The first line of the input has to contain the size of the matrix  $V$  and the number of nonzeros  $E$ , because before we start we have to initialize a couple of arrays. Every other line contains 2 integers that are the location of a nonzero in the matrix and every location of a nonzero occurs once in the inputfile. Because this is a program for symmetric matrices we only read nonzeros from the lower triangular part of the matrix.

### Main program

```
#include<stdio.h>
#include <stdlib.h>
#define NONZ 300000
#define N 20000

/*****Variable Declaration*****/

/*****Datastructure*****/

int *Cliques[NONZ], Length[NONZ], Degree[N];

struct Place{
    int CliqueName;
    struct Place *Next;
};

struct Place *Location[N];

struct Vertex{
    int NextNode,PrevNode,Degree,SNL;
};
```

```

struct Vertex Vertices[N];

/*****End Datastructure*****/

/* arrays used for flagging */
int n[N],CliqueCount[NONZ],NodeCount[N],UC[NONZ],RN[N],h1[N],m[N];

int Order[N],LgthOrder,LengthRN,LengthUC,lengthn,i,Neighbor[N],lengthm;

int E,V,mindeg;

/*****Functions*****/

void InitArray(int *Length, int E){
    int i;
    for(i=0;i<E;i++)
        Length[i]=0;
}

void InitDegree(int *Degree, int V){
    int i;
    for(i=0;i<V;i++)
        Degree[i]=-1;
}

void InitVertices(struct Vertex *Vertices, int V){
    int i;
    for(i=0;i<V;i++){
        Vertices[i].NextNode=i;
        Vertices[i].PrevNode=i;
        Vertices[i].Degree=0;
        Vertices[i].SNL=i;
    }
}

int MinDegNode(int *mindeg,int *Degree){
    return Degree[*mindeg];
}

void Update(int *Degree,struct Vertex *Vertices,int *mindeg,int i,

```

```

        int NewDeg){

/* when reading nonzeros from matrix A, this funtion increases
   the degree of i by one and changes the array Degree if
   necessary */

int l,k;

/* check if node i is a representative of the nodes with the
   same degree as i in the array i. If so then it has to
   be changed, since were gonna increase the degree of i by one.
   We can find a new candidate, via the double cyclic linked list.
   If there is no candidate it gets the default value -1 */

l=Vertices[i].NextNode;
k=Vertices[i].PrevNode;
Vertices[k].NextNode=l;
Vertices[l].PrevNode=k;
if(Degree[Vertices[i].Degree]==i){
    if(l==i)
        Degree[Vertices[i].Degree]=-1;
    else{
        Degree[Vertices[i].Degree]=l;
    }
}
/* change degree of node i and check if there is a
   representative for node with that degree. If so insert
   node i in that double cyclic linked list. If not make
   node i the representative of node with such a degree */

if(NewDeg<0)
    Vertices[i].Degree++;
else{
    Vertices[i].Degree=NewDeg;
    if(NewDeg<*mindeg)
        *mindeg=NewDeg;
}
if(Degree[Vertices[i].Degree]==-1){
    Degree[Vertices[i].Degree]=i;
    Vertices[i].PrevNode=i;
    Vertices[i].NextNode=i;
}
else{
    l=Degree[Vertices[i].Degree];
    k=Vertices[l].NextNode;
    Vertices[i].NextNode=k;
    Vertices[l].NextNode=i;
    Vertices[i].PrevNode=l;
    Vertices[k].PrevNode=i;
}
}

```

```

    }
}

void Reading(int *Length,int *Degree,struct Vertex *Vertices,
            struct Place **Location,int **Cliques,int *mindeg){

/* reads the nonzeros of A from the input line as pairs (i,j) which
   are the positions at which A has nonzeros. These nonzeros are
   inserted in the datastructure */

int i,j,number,size;
char line[30];
struct Place *z;

/* number is the number of cliques created so far */
/* we will start with number 1 because we also use
   negative cliquenumbers */
number=1;
*mindeg=N;
while(fgets(line,sizeof(line),stdin)!=NULL){
    sscanf(line,"%d %d",&i,&j);
    if(i!=j){
        /* update Degree and degree nodes */
        Update(Degree,Vertices,mindeg,i,-1);
        Update(Degree,Vertices,mindeg,j,-1);
        /* create new clique of size 2*/
        size=2*sizeof(int);
        Cliques[number]=(int *)malloc(size);
        Cliques[number][0]=i;
        Cliques[number][1]=j;
        Length[number]=2;

        /* insert cliquename in Location array on positions i and j */
        z=(struct Place *)malloc(sizeof(struct Place));
        z->CliqueName=number;
        z->Next=Location[i];
        Location[i]=z;
        z=(struct Place *)malloc(sizeof(struct Place));
        z->CliqueName=number;
        z->Next=Location[j];
        Location[j]=z;
        number++; /* increase by one for the next new clique*/
    }
}
}

#include "merge.c"
#include "degrupda.c"

```

```

#include "remove.c"
#include "masselim.c"
#include "multiple.c"

/*****Program Body*****/

main(int argc, char **argv){

char line[30];
int i,j;
struct Place *r;

fgets(line, sizeof(line), stdin);
fgets(line, sizeof(line), stdin);
sscanf(line, "%d %d", &V, &E);
InitArray(CliqueCount, (E+1));
InitArray(h1, V);
InitArray(Length, (E+1));
InitArray(NodeCount, V);
InitArray(Neighbor, V);
InitDegree(Degree, V);
InitVertices(Vertices, V);
Reading(Length, Degree, Vertices, Location, Cliques, &mindeg);

LengthRM=0;
LengthUC=0;
lengthn=0;
LgthOrder=0;
mindeg=0;

while( (Degree[mindeg]<0) && (mindeg<V) )
    mindeg++;

for(i=0; i<V; i++){
    if(Vertices[i].Degree==0){
        Vertices[i].Degree=2*V;
        Order[LgthOrder]=i;
        LgthOrder++;
    }
}

while(LgthOrder<V){
    m[0]=Degree[mindeg];
    lengthm=1;

    MultipleElimination(&lengthm, m, Length, Cliques, Vertices, Location, mindeg, 1,
        Degree, h1, V);
}

```

```

for(i=0;i<lengthm;i++){
    n[0]=m[i];
    lengthn=1;

    MassElimination(&lengthn,n,h1,Cliques,Length,Location);

    Merge(lengthn,n,Cliques,Length,CliqueCount,NodeCount,Neighbor,Vertices,
        Location,UC,&LengthUC,RN,&LengthRN,h1) ;
}

DegreeUpdate(&LengthUC,UC,Length,Cliques,NodeCount,h1,Degree,Vertices,
    Location,&mindeg,CliqueCount);

RemoveNodes(RN,LengthRN,Order,&LgthOrder,Location,Cliques,Vertices,
    CliqueCount,Length,Degree);

while( (Degree[mindeg]<0) && (mindeg<V) )
    mindeg++;
LengthUC=0;
LengthRN=0;
lengthn=0;
lengthm=0;

if(mindeg==V-LgthOrder-1){
    for(i=0;i<V;i++){
        if(Vertices[i].Degree!=2*V){
            Order[LgthOrder]=i;
            LgthOrder++;
        }
    }
}

printf("%d\n",LgthOrder);
printf("%d %d\n",V,E);
for(i=0;i<LgthOrder;i++){
    printf("%d \n",Order[i]);
}
}

Merge

void Put(int *k,int l,int *Neighbor,int *h1,int *klength,int *m){

/* flags the nodes from clique k in array Neighbor, whose
    flagged elements are nodes for the new clique */

```

```

int i;

for(i=0;i<l;i++){
    /* check if it's not one of the nodes which we are eliminating */
    if(h1[k[i]]==0){
        /* check if the node is a new node in the new clique */
        if(Neighbor[k[i]]==0){
            m[*klength]=k[i];
            (*klength)++;
            Neighbor[k[i]]=1;
        }
    }
}

int *MakeNewClique(int *m,int klength,int *Neighbor){

int *l,size;

size=klength*sizeof(int);
l=(int *)malloc(size);
for(i=0;i<klength;i++){
    l[i]=m[i];
    Neighbor[m[i]]=0;
}
return l;
}

void Merge(int lengthn,int *n,int **Cliques,int *Length,int *CliqueCount,
           int *NodeCount,int *Neighbor,struct Vertex *Vertices,
           struct Place **Location,
           int *UC,int *LengthUC,int *RN,int *LengthRN,int *h1){

/* array n contains lengthn nodes which we want to eliminate and
which are indistinguishable. First all cliques which contain at
least one of these nodes will form a new clique.
The new clique will be inserted in the position of one of the
old cliques and this position will be stored in the array
UC (UpdateCliques) which contains numbers of cliques whose nodes
need a degree update LengthUC is the number of elements in UC.
RN (RemoveNodes) is a array of nodes which are to be eliminated from the
current elimination graph */

struct Place *p;
int klength,*k,l,i,j;
int m[N];

```

```

klength=0;
for(i=0;i<lengthn;i++)
    h1[n[i]]=1; /* flag nodes which are to be eliminated */

/* traverse all cliques which contain nodes from n, in such
   a way that we don't visit the same clique twice. For this
   we use the array CliqueCount to flag cliques */

for(i=0;i<lengthn;i++){

    p=Location[n[i]];
    while(p!=NULL){

        /* check if clique already has been visited */
        if(CliqueCount[p->CliqueName]==0){
            /* Check if node n[i] is a supernode */
            if(Vertices[n[i]].Degree>0){

                CliqueCount[p->CliqueName]=1;
                k=Cliques[p->CliqueName];
                l=Length[p->CliqueName];
                Put(k,l,Neighbor,h1,&klength,m);
            }
            else
                /* This means that this clique is visited by a non-supernode */
                CliqueCount[p->CliqueName]=3;
        }
        else{
            if(CliqueCount[p->CliqueName]==3){
                if(Vertices[n[i]].Degree>0){
                    CliqueCount[p->CliqueName]=1;
                    k=Cliques[p->CliqueName];
                    l=Length[p->CliqueName];
                    Put(k,l,Neighbor,h1,&klength,m);
                }
            }
        }
        p=p->Next;
    }
}

if(klength>0){
    k=MakeNewClique(m,klength,Neighbor);
    /* j will be the name of the new clique */
    j=Location[n[0]]->CliqueName;
    /* j is flagged 2 in CliqueCount to make sure that if we erase old
       cliques, we don't erase j. All cliques wich are flagged 1 or 3
       in CliqueCount are old */
}

```

```

CliqueCount[j]=2;
free(Cliques[j]);
Cliques[j]=k;
Length[j]=klength;
CliqueCount[j]=2;

/* insert in every Location of the nodes of the new clique the
   clique number j as a negative number. So if we do a degreeupdate
   we know that this is the new clique */
for(i=0;i<klength;i++){

    /* we don't have to update the lists of nodes which aren't supernodes
       because we update their representatives */
    if(Vertices[k[i]].Degree>0){
        p=(struct Place *)malloc(sizeof(struct Place));
        p->CliqueName=-j;
        p->Next=Location[k[i]];
        Location[k[i]]=p;
        NodeCount[k[i]]=1;
    }
}

/* Update set UC and RN */
UC[(*LengthUC)]=j;
(*LengthUC)++;
}
for(i=0;i<lengthn;i++){
    RN[(*LengthRN)]=n[i];
    (*LengthRN)++;
    h1[n[i]]=0;
}
}

```

### Degree Update

```

void CheckClique(int c,int l,int **Cliques,int *mlength, int *m,int *h1){

/* checks if a clique with cliquename c and length l has some
   new nodes which we haven't yet counted, search of the new degree
   of a node. For this we use array m and array h1 for flagging. If
   there is a new node then it is put in m en mlength is increased by
   one */

int i, *k;

k=Cliques[c];
for(i=0;i<l;i++){
    if(h1[k[i]]==0){
        h1[k[i]]=1;
    }
}
}

```

```

        m[(*mlength)]=k[i];
        (*mlength)++;
    }
}
}

void DegreeUpdate(int *LengthUC,int *UC,int *Length,int **Cliques,
                int *NodeCount,int *h1,int *Degree,
                struct Vertex *Vertices,struct Place **Location,
                int *mindeg,int *CliqueCount){

/* will make a degree update of the nodes contained in cliques
whose clique number is contained in array UC of length LengthUC,
furthermore, if a newdegree is smaller than mindeg then mindeg will
change to that value. The cliques contained in UC can have some
overhead with eachother (we hope so) so we use a flagging array
NodeCount to check this. This can save a lot of work.*/

int m[N],l,i,j,c,*k,mlength,t;
struct Place *z,*p,*x;

for(i=0;i<(*LengthUC);i++){
    l=Length[UC[i]];
    k=Cliques[UC[i]];
    for(j=0;j<l;j++){
        mlength=0;
        /* check if we haven't visit this node in an other clique which
        needed an degree update */

        if(NodeCount[k[j]]==1){
            NodeCount[k[j]]=0;
            /* check if node k[j] is a supernode */
            if(Vertices[k[j]].Degree>0){
                /* we don't want to count the node itself in it's degreeupdate */
                h1[k[j]]=1;
                z=Location[k[j]];
                p=Location[k[j]];
                /* traverse all clique names which contain node k[j] and
                check if the clique has to be removed/emptied. If so
                eliminate this clique name from the list Location[k[j]].
                The cliques we don't remove we count the number of
                different nodes in the cliques. For this we use
                an array m and mlength. mlength will become the new degree*/
                while(p!=NULL){
                    c=p->CliqueName;
                    /* c is a new clique */
                    if(c<0){
                        p->CliqueName=-c;
                        CheckClique(-c,Length[-c],Cliques,&mlength,m,h1);
                    }
                }
            }
        }
    }
}
}

```

```

        if(p==Location[k[j]])
            p=p->Next;
        else{
            z=p;
            p=p->Next;
        }
    }
else{
    /* clique c don't have to be removed so count the
       number of different nodes it contains using m and
       mlength and array h1 for flagging */
    if(CliqueCount[c]==0){
        CheckClique(c,Length[c],Cliques,&mlength,m,h1);
        if(p==Location[k[j]])
            p=p->Next;
        else{
            z=p;
            p=p->Next;
        }
    }
    else{
        /* clique c has to be removed/emptied from list
           Location[k[j]]. check first if c is a the beginning
           of the list or not */
        if(p==Location[k[j]]){
            p=p->Next;
            z->Next=NULL;
            free(z);
            z=p;
            Location[i]=z;
        }
        else{
            x=p;
            p=p->Next;
            z->Next=p;
            x->Next=NULL;
            free(x);
        }
    }
}
}
Update(Degree,Vertices,mindeg,k[j],mlength);
h1[k[j]]=0;
for(t=0;t<mlength;t++)
    /* reset flagged array h1 for other node which need
       a degree update */
    h1[m[t]]=0;
}
}
}

```

```

}
}

```

### Remove Nodes

```

void RemoveNodes(int *RN,int LengthRN,int *Order,int *LgthOrder,
                 struct Place **Location,int **Cliques,
                 struct Vertex *Vertices,int *CliqueCount,int *Length,
                 int *Degree){

```

```

/* Nodes put in array RN are to be removed and put in array Order
because Order will give us a collumn ordering for the Choleski
matrix. Furthermore we remove all cliques containing one
of these nodes. Because we don't want to remove cliques twice
we use the array CliqueCount in which cliques are flagged if
they have to be removed, unless if the clique is flagged 2.
This means that this clique was replaced by a newly created
clique */

```

```

int i,l,k,node,d;
struct Place *p;

```

```

for(i=0;i<LengthRN;i++){
    node=RN[i];
    Order[(*LgthOrder)]=node;
    (*LgthOrder)++;
    p=Location[node];
    while(p!=NULL){
        /* remove all cliques in list Location[node] */
        l=p->CliqueName;
        /* check if we haven't removed it already */
        if(CliqueCount[l]!=0){
            /* if clique is flagged two DON'T ERASE IT!! */
            if(CliqueCount[l]!=2){
                free(Cliques[l]);
                Cliques[l]=NULL;
                Length[l]=0;
            }
            CliqueCount[l]=0;
        }
        p=p->Next;
    }
    /* erase list of node which is removed */
    free(Location[node]);
    Location[node]=NULL;
    /* check if node was a representative in Degree of nodes with
       the same degree, and adjust array Degree and array Vertices */

    /* if node is not a supernode then Vertices[node].Degree will

```

```

        have the negative value of the representative (supernode) so
        we have to find the degree of node in a different way */

if (Vertices[node].Degree < 0)
    d = Vertices[-(Vertices[node].Degree)].Degree;
else
    d = Vertices[node].Degree;

if (Degree[d] == node) {
    if (Vertices[node].NextNode == node)
        /* node was the only node with this degree */
        Degree[d] = -1;
    else {
        /* node was not the only node with this degree
           so we need an other representative and have
           to remove node from this list */
        l = Vertices[node].PrevNode;
        k = Vertices[node].NextNode;
        Degree[d] = k;
        Vertices[k].PrevNode = l;
        Vertices[l].NextNode = k;
    }
}
else {
    /* node wasn't a representative but still has to be removed from
       the list it is contained in */
    l = Vertices[node].PrevNode;
    k = Vertices[node].NextNode;
    Vertices[k].PrevNode = l;
    Vertices[l].NextNode = k;
}
Vertices[node].PrevNode = node;
Vertices[node].NextNode = node;
}
for (i = 0; i < LengthRN; i++)
    Vertices[RN[i]].Degree = 2 * V;
}

```

### Multiple Elimination

```

void MultipleElimination(int *lengthm, int *m, int *Length, int **Cliques,
                        struct Vertex *Vertices, struct Place **Location,
                        int mindeg, float tol, int *Degree, int *h1, int V) {

```

```

/* m[0] is the node with minimum degree such that m[0] is a representative
of a supernode. MultipleElimination will
find a set M containing node m[0] such that all nodes have minimum degree
but none of them are neighbours */

```

```

int nextvertex,i,*k,flag,d,n;
struct Place *p;

/* h1[] contains flags on the nodes which are already in set M */
h1[m[0]]=1;
n=m[0];

d=mindeg;
while((d<=(tol*mindeg)) && (d<V)){

    nextvertex=Vertices[n].NextNode;
    while(nextvertex!=n){
        if(Vertices[nextvertex].Degree>0){
            p=Location[nextvertex];
            flag=0;
            /* check if neighbours of nextvertex aren't already in set M */
            while( (p!=NULL) && (flag==0) ){
                k=Cliques[p->CliqueName];
                for(i=0;i<Length[p->CliqueName];i++)
                    if(h1[k[i]]==1)
                        flag=1;
                p=p->Next;
            }
            if(flag==0){
                /* inserting nextvertex in set M */
                h1[nextvertex]=1;
                m[*lengthm]=nextvertex;
                (*lengthm)++;
            }
            else
                flag=0;
        }
        nextvertex=Vertices[nextvertex].NextNode;
    }

    d++;
    while( (Degree[d]<0) && (d<=(tol*mindeg)) && (d<V))
        d++;
    n=Degree[d];
}
/* unflagging h1[] */
for(i=0;i<(*lengthm);i++)
    h1[m[i]]=0;
}

```

### Mass Elimination

```

void MassElimination(int *lengthn,int *n,int *h1,int **Cliques,
                    int *Length,struct Place **Location){

```

```

/* n[0] is a node with minimum degree and you want to find
the neighbours of n[0] with the same degree and with the same
adjacency list. If n[0] were the representative of a supernode
then all the indistinguishable nodes with n[0] as representative
are already put in array n[] */

int i,j, *k, *z,flag,x,y;
struct Place *p, *q;

/* we don't want node 0 as supernode because we use negative
node numbers to refer to the supernode */
if(n[0]!=0){
    p=Location[n[0]];

    /* flagging neighbours of n[0] and n[0] in h1[] */
    h1[n[0]]=1;
    while(p!=NULL){
        k=Cliques[p->CliqueName];
        for(i=0;i<Length[p->CliqueName];i++){
            if(h1[k[i]]==0)
                h1[k[i]]=1;
        }
        p=p->Next;
    }
    p=Location[n[0]];
    /* traversing all neighbours again for mass elimination check */
    while(p!=NULL){
        k=Cliques[p->CliqueName];
        for(i=0;i<Length[p->CliqueName];i++){
            /* check if neighbour has the same degree */
            /* if we use multiple elimination we can do a couple
of merges before doing a degree update if a node i is
put in a new clique because of a merge we insert the
negative cliquename at the beginning of the list
Location[i]. So if we are scanning nodes for
masselimination nodes that contain a negative
cliquename at the begining of the list are not candidates
for that because they were also in the adjacency list
of another node from the multiple elimination set and
therefore can not have the same adjacency list as another
node from the multiple elimination set */
            if((Vertices[k[i]].Degree==Vertices[n[0]].Degree) && (k[i]!=n[0])
                && (Location[k[i]]->CliqueName>0)){
                /* if so flag it, we don't want to visit the same
neighbour twice*/
                if(h1[k[i]]!=2){
                    h1[k[i]]=2;
                    q=Location[k[i]];
                    flag=0;
                    /* check if node k[i] has the same adjacency list as n[0] */
                    while( (q!=NULL) && (flag==0) ){

```

```

        z=Cliques[q->CliqueName];
        for(j=0;j<Length[q->CliqueName];j++){
            if(h1[z[j]]==0){
                flag=1;
            }
        }
        q=q->Next;
    }
    if(flag==1)
        flag=0;
    else{
        n[*lengthn]=k[i];
        (*lengthn)++;
        /* make k[i] indistinguishable */
        Vertices[k[i]].Degree=-n[0];
        /* insert k[i] in SNL list and check if it is
        a supernode representative, if so put all
        the nodes from this supernode also in the SNL list
        We don't have to adjust array Degree because
        this node will be eliminated this round, so
        this will be done by remove*/
        x=Vertices[k[i]].SNL;
        Vertices[k[i]].SNL=Vertices[n[0]].SNL;
        Vertices[n[0]].SNL=k[i];
        while(x!=k[i]){
            y=Vertices[x].SNL;
            Vertices[x].Degree=-n[0];
            Vertices[x].SNL=Vertices[n[0]].SNL;
            Vertices[n[0]].SNL=x;
            x=y;
        }
    }
}
}
}
}
p=p->Next;
}
p=Location[n[0]];
/* unflag neighbours of n[0] and n[0] in h1[] */
h1[n[0]]=0;
while(p!=NULL){
    k=Cliques[p->CliqueName];
    for(i=0;i<Length[p->CliqueName];i++)
        if(h1[k[i]]!=0)
            h1[k[i]]=0;
    p=p->Next;
}
}
}
}

```

## Appendix B

# The parallel minimum degree program

### Main program

```
#include<stdio.h>
#include<stdlib.h>

double bsp_time();
double bsp_dtime();
#define SZINT sizeof(int)
#define N 2
#define M 3 /* N & M is the gridsize */
#define SIZE 4500 /* is the maximum order of the matrix */
#define NONZ 120000 /* is the maximum number of nonzeros */
#define PROC 0
/* Global declaration */

/* used for putting subcliques in the hash array Cliques */

struct Name{
    int SCliqueName;
    int SCLength;
    struct Name *Next;
    int *SClique;
};

struct Name *Cliques[(NONZ/(N*M)+1)];

struct Place{
    int CliqueName;
    int flag;
    struct Place *Next;
};
```

```

struct Place *Location[SIZE/M+1];

struct DegLoc{
    int Degree,Node;
    struct DegLoc *Next;
};

struct DegLoc *Degree[SIZE/(N*M)+1];

struct Vertex{
    int NextNode,PrevNode;
    int Degree;
};

struct Vertex Vertices[SIZE/(N*M)+1];

int CliqueCount[NONZ/(N*M)+1];

int h1[SIZE/M+1],h2[NONZ/M+1],h3[(SIZE/M+1)][N];

int E,V; /* sizes of the input */

int Clength,h1Length,h2Length; /*used to keep track of the amount of data in
                                arrays CliqueCount and h1*/

int flag1,flag2; /*these are used when we are reading data*/

int Order[SIZE/(N*M)+1]; /* used to store the order of the columns */
                        /* such that i'th column of the permutation
                           matrix is located in processor i%(MN) on
                           position Order[i/(MN)]*/

int cand,mr,ncn,lgthorder,gmmdn,degree,totl;

int row[M][2],column[N][2];

/* time testing */

int super[32];
float sstime[32];

#include "form.c"

```

```

#include "read.c"
#include "chosemmd.c"
#include "bspmerge.c"
#include "bspdegup.c"
#include "bspremov.c"

main(int argc,char **argv){

void Reading(int s,int *lmmdn);
int i,j,p,s,lmmdn, *k;
char line[30];
struct Name *d;
struct DegLoc *z;

bspstart(argc,argv,(N*M),&p,&s);
/* p=number of processors
   s=processornumber */
if(p!=(N*M)){
    printf("less processors obtained than requested\n");
    bspfinish();
}

/* our input will be read in by processor 0 and will have to be distributed
to the right processor, by processor 0. Because were working with cliques
which are dynamically created we can't write the data direct in to the
array Cliques. Therefore we use another technique. Every processor
has 2 arrays we use for flagging during the execution of the algorithm.
When we are reading data we will use thes arrays to temporarily store
data which is ment to be for array cliques. These 2 arrays will be filled
during the reading of the data. If there is a processor who's array
is almost full then all arrays in every processor if flushed. e.g. all
the data in these array is moved to the right position */

bpsstep(1);
    lgthorder=0;
    if(s==0){
        fgets(line,sizeof(line),stdin);
        sscanf(line,"%d %d",&V,&E);
        /* broadcast V and E to other processors for initialisation procedure*/
        for(i=1;i<(N*M);i++){
            bspstore(i,&V,&V,SZINT);
            bspstore(i,&E,&E,SZINT);
        }
    }
    for(i=0;i<31;i++){
        super[i]=0;
        sstime[i]=0;
    }

```

```

bspssstep_end(1);

/* initialisation */

bsp_time();

bspssstep(2);
for(i=0;i<(V/(N*M)+1);i++){
    Vertices[i].Degree=0;
    Vertices[i].NextNode=i;
    Vertices[i].PrevNode=i;
    /* in processor s, index i in array vertices equals node i*M*N+s */
}
if(s==0){
    for(i=0;i<(E/(N*M)+1);i++)
        CliqueCount[i]=0;
    for(i=0;i<(E/M+1);i++)
        h2[i]=0;
    for(i=0;i<(V/(N*M)+1);i++)
        Degree[i]=NULL;
    for(i=0;i<(V/M+1);i++)
        h1[i]=0;
}
for(i=0;i<N;i++)
    for(j=0;j<(V/M+1);j++)
        h3[j][i]=-1;
bspssstep_end(2);

form(bsp_dtime(),2);

Reading(s,&lmmdn);

if(s==PROC){
    printf("%d\n",0);
    printf("%d %d\n",V,E);
}

ChooseMMD(s,lmmdn);
while( (degree!=(V-lgthorder-1)) && (lgthorder<V) ){

bsp_time();

    bspssstep(13);
    if(s==PROC)
        printf("%d\n",gmmdn);
    bspssstep_end(13);
}

```

```

form(bsp_dtime(),13);

    if(degree>0){
    Merge(s,gmmdn);
    BSPDegreeUpdate(s,&lmmdn);
    }
    Remove(s,&lmmdn);
    ChooseMMD(s,lmmdn);
    }

i=0;
while( (i<(V/(N*M)+1)) && ((i*M*N+s)<V) ){
    if(Vertices[i].Degree>-1)
        printf("%d\n",i*M*N+s);
    i++;
}

if(s==0){
    for(i=2;i<32;i++){
        printf("superstep=%d average time=%f number=%d total time=%f\n",i,sstime[i],super[i],super[i]*sstime[i]);
    }
}
bspfinish();
}

```

## Reading

```

void MoveSClique(int pid,int i,int number,int rank,int *flag2){

/* moves data read by processor 0 to processor pid and puts it
in array cliquecount. it can only happen that we read a subclique
of size 1 or 2. In cliquecount we put subcliques of size 1 we use
their negative number. So if number<0 we know that i is a subclique
of size one is number>0 we know that i is part of a subclique of
size 2, rank tells us if number>0, if i is the first part or the
second part of the clique. So if we read array cliquecount
in a processor not equal to 0 then the first entry is the
cliquenumber which also tells the length of the subclique of that
clique in that processor */

/* in processor 0 cliquecount keeps track of the number of elements
in cliquecount of other processors. CliqueCount[i] gives us the
number of elements put in this array during the reading of the
nonzeros. Using this we can see when there is a array in a processor
which is full. After that we can flush all arrays and start again
with empty arrays */

int pos;

```

```

pos=CliqueCount[pid];
if(rank==1){
    bspstore(pid,&number,&CliqueCount[pos],SZINT);
    CliqueCount[pid]++;
}
pos=CliqueCount[pid];
bspstore(pid,&i,&CliqueCount[pos],SZINT);
CliqueCount[pid]++;
/* check if we are still within the length of the array. because we
can read in one step at most 3 elements in a particular array we
have to take that into account */
if(CliqueCount[pid]>(E/(N*M)+1-3))
    (*flag2)=1;
}

void InsertSClique(int *c,int l,int number){

/* inserts a subclique of length 1 or 2 (l is this length) in the
hash array Cliques using the cliquenumber*/

int pos,*k,size,i;
struct Name *z,*p;

z=(struct Name *)malloc(sizeof(struct Name));
z->SCliqueName=number;
z->SCLength=l;
size=l*sizeof(int);
k=(int *)malloc(size);
for(i=0;i<l;i++)
    k[i]=c[i];
z->SClique=k;
z->Next=NULL;
pos=number/N;
pos%=(E/(N*M)+1);
p=Cliques[pos];
if(p==NULL)
    Cliques[pos]=z;
else{
    z->Next=p->Next;
    p->Next=z;
}
}

void UpdateLocation(int node,int CliqueName){

/* updates the cliques node is contained in. This is done
locally. So we know that the entry Location of node
is contained in this processor */

```

```

int pos;
struct Place *p,*z;

pos=node/M;
z=(struct Place *)malloc(sizeof(struct Place));
z->CliqueName=CliqueName;
z->flag=0;
z->Next=Location[pos];
Location[pos]=z;
}

void MoveLocation(int pid,int node,int CliqueName,int *flag2){

/* When we are reading the nonzeros from the input we also
have to keep track of the cliques node are contained in. For
this we use arrays Location which we have distributed over
the processors. Since this structure is a dynamic structure
we can not direct access this structure if we are reading input
with processor 0. And we want to manipulate array location
in another processor. What we do is that we store the data
we want to put in array location in that other processor
in a static array h1 in that processor and occasionally
flush this array (e.g. put the input in array location).
Furthermore, in array h1 in processor 0 we keep track of
the length of data put in arrays h1 in other processors.
h1[i] in processor 0 corresponds with the length of h1 in
processor i. If one of the length gets to large we flush
them all and start again with length 0 */

int l;

l=h1[pid];
bspstore(pid,&node,&h1[l],SZINT);
h1[pid]++;
l=h1[pid];
bspstore(pid,&CliqueName,&h1[l],SZINT);
h1[pid]++;
/* check if array is full */
if(h1[pid]>(V/M+1-2))
    (*flag2)=1;
}

void UpdateVertices(int node,int tag){

/* node/(N*M) is position in array vertices in that processor */

```

```

Vertices[node/(N*M)].Degree++;
}

void MoveVertices(int pid,int node, int *flag2){

/* updates the degree of node node. when reading the nonzeros
by processor 0. Because we distributed array Vertices
we have to fetch the degree of node if it is not in processor zero
increase it by one and store it again in that processor. This
can be done directly because this is a static structure */

int pos;

pos=h2[pid];
bspstore(pid,&node,&h2[pos],SZINT);
h2[pid]++;
if(h2[pid]>(E/N-1))
(*flag2)=1;
}

void Reading(int s,int *lmmdn){

/*this procedure reads in the nonzeros of the matrix and distributes
is over the processors as described in the paper. */

int number,pid,pos,i,j,c[2],k,node,cliquename,l;
int pos1,pos2,pos3,pid1,pid2,pid3,degree,NextNode;
char line[30];
struct DegLoc *d;

flag1=0;
number=1; /* this represents the number of cliques. We start with one
because we also will use negative cliquenames and we don't
have a negative zero */
while(flag1==0){

bsp_time();

bspstep(3);
if(s==0){
flag2=0;
while(flag2==0){
if(fgets(line,sizeof(line),stdin)!=NULL){
sscanf(line,"%d %d",&i,&j);
/* check if it is not a diagonal entry */
if(i!=j){
/* check if both i and j are projected to the same column */
if(i%M==j%M){

```

```

/* check if processor number is processor 0 if so then
   put these data direct in the array Cliques */
pid=(number%M)*M+i%M;
if(pid==0){
    c[0]=i;
    c[1]=j;
    InsertSClique(c,2,number);
}
else{
    MoveSClique(pid,i,number,1,&flag2);
    MoveSClique(pid,j,number,2,&flag2);
}
}
else{
/* first put node i away then node j */
pid=(number%M)*M+i%M;
if(pid==0){
    c[0]=i;
    InsertSClique(c,1,number);
}
else{
    MoveSClique(pid,i,-number,1,&flag2);
}
}
pid=(number%M)*M+j%M;
if(pid==0){
    c[0]=j;
    InsertSClique(c,1,number);
}
else{
    MoveSClique(pid,j,-number,1,&flag2);
}
}
pid=(number%M)*M+i%M;
if(pid==0)
    UpdateLocation(i,number);
else
    MoveLocation(pid,i,number,&flag2);
pid=(i%(M*N));
if(pid==0)
    UpdateVertices(i,0);
else
    MoveVertices(pid,i,&flag2);
pid=(number%M)*M+j%M;
if(pid==0)
    UpdateLocation(j,number);
else
    MoveLocation(pid,j,number,&flag2);
pid=(j%(M*N));
if(pid==0)
    UpdateVertices(j,0);

```

```

        else
            MoveVertices(pid,j,&flag2);
        number++;
    }
}
else{
    flag1=1;
    flag2=1;
}
}
/* Or we have no more data to read or there is array which
is full and we will flush all arrays. For this we broadcast
the variable flag1 because if we are at the end of the input
file all processor must be able to go to the next superstep.
Furthermore we broadcast the length of array CliqueCount and
array h1 to the processors that contain it, so they will be
able to read the data into their datastructure */

for(i=1;i<(N*M);i++){
    bspstore(i,&CliqueCount[i],&Clength,SZINT);
    bspstore(i,&h1[i],&h1Length,SZINT);
    bspstore(i,&h2[i],&h2Length,SZINT);
    bspstore(i,&flag1,&flag1,SZINT);
    CliqueCount[i]=0;
    h1[i]=0;
    h2[i]=0;
}
}
bpsstep_end(3);

form(bsp_dtime(),3);
/* reading the data into the datastructure */

bsp_time();

bpsstep(4);
if(s!=0){
    k=0;
    while(k<Clength){
        if(CliqueCount[k]<0){
            number=-CliqueCount[k];
            c[0]=CliqueCount[k+1];
            InsertSClique(c,1,number);
            k+=2;
        }
        else{
            number=CliqueCount[k];
            c[0]=CliqueCount[k+1];
            c[1]=CliqueCount[k+2];
            InsertSClique(c,2,number);
        }
    }
}

```

```

        k+=3;
    }
}
k=0;
while(k<h1Length){
    node=h1[k];
    cliquename=h1[k+1];
    UpdateLocation(node,cliquename);
    k+=2;
}
k=0;
while(k<h2Length){
    UpdateVertices(h2[k],2);
    k++;
}
}
bpsstep_end(4);

form(bsp_dtime(),4);
}

```

/\* we have now read in the edges (as cliques of size 2) into our datstructure and we know the degree of every vertex. The only thing we don't know are the entries of the array degree. We use one of such array per processor. Because we don't want to use too much memory we use a short array per processor and we will use hashing. which means  $i$  equals  $i \bmod MN$  \*/

```
bsp_time();
```

```

bpsstep(5);
(*lmmdn)=0;
for(i=0;i<(V/(N*M)+1);i++){
    if( (i*M*N+s)<V ){
        degree=Vertices[i].Degree;
        if(degree<Vertices[*lmmdn].Degree)
            (*lmmdn)=i;
        pos=degree%(V/(N*M)+1);
        d=Degree[pos];
        while( (d!=NULL) && (d->Degree!=degree) )
            d=d->Next;
        if(d==NULL){
            d=(struct DegLoc *)malloc(sizeof(struct DegLoc));
            d->Next=Degree[pos];
            d->Degree=degree;
            d->Node=i; /* local node value */
            Degree[pos]=d;
        }
    }
}
else{

```

```

        l=d->Node;
        k=Vertices[l].NextNode;
        Vertices[i].NextNode=k;
        Vertices[l].NextNode=i;
        Vertices[i].PrevNode=l;
        Vertices[k].PrevNode=i;
    }
}
}
for(i=0;i<(E/N+1);i++)
    h2[i]=0;
for(i=0;i<(V/M+1);i++)
    h1[i]=0;
for(i=0;i<(E/(N*M)+1);i++)
    CliqueCount[i]=0;
bpsstep_end(5);

form(bsp_dtime(),5);
}

```

### Choose minimum degree

```

void ChooseMMD(int s,int lmmdn ){

/* processor s knows the local minimum degree node lmmdn and
we want to know the global minimum degree node gmmdn.
first we calculate the rowwise minimum degree node and then
the columnwise minimum degree node. s=(s/M,s%M) (2 dimensional
numbering). Thus first broadcasting to processors p in the same
row. (e.g. p/M=s/M) and then broadcasting to processors in the
same column (p%M=s%M). */

int i,pid;

bsp_time();

bpsstep(10);
if(lmmdn!=-1){
    degree=Vertices[lmmdn].Degree;
    lmmdn=lmmdn*M*N+s; /*change to global value */
}
else
    degree=-1;
/* send to processors in the same row */
for(i=0;i<M;i++){
    pid=(s/M)*M+i;
    bspstore(pid,&lmmdn,&row[s%M][0],SZINT);
    bspstore(pid,&degree,&row[s%M][1],SZINT);
}
}

```

```

    }
    bspstep_end(10);

    form(bsp_dtime(),10);

    bsp_time();

    bspstep(11);
    gmmdn=row[0][0];
    degree=row[0][1];
    /* choose minimum degree node per row */
    for(i=1;i<M;i++){
        if( ( row[i][1]<degree) && (row[i][1]>-1) ) || (degree<0) ){
            gmmdn=row[i][0];
            degree=row[i][1];
        }
    }
    for(i=0;i<N;i++){
        pid=i*M+s%M;
        bspstore(pid,&gmmdn,&column[s/M][0],SZINT);
        bspstore(pid,&degree,&column[s/M][1],SZINT);
    }
    bspstep_end(11);

    form(bsp_dtime(),11);

    bsp_time();

    bspstep(12);
    gmmdn=column[0][0];
    degree=column[0][1];
    /* choose global minimum degree node */
    for(i=1;i<N;i++){
        if( ((column[i][1]<degree) && (column[i][1]>-1)) || (degree<0) ){
            gmmdn=column[i][0];
            degree=column[i][1];
        }
    }
    bspstep_end(12);

    form(bsp_dtime(),12);
}

```

### Merge cliques

```

void SearchSC(int pos,int s,int i,int *length,int rowid){

/* this procedure will look into hash array Cliques[pos] if it
contains a subclique of clique i. It will put elements of

```

```

this subclique (if there is any) into array h3[][rowid]
if and only if there not already contained in this array.
To check this we use the array h1 */

int j,*k,l;
struct Name *c,*d;

c=Cliques[pos];
d=c;
if(c!=NULL){
  if(c->SCliqueName==i){
    k=c->SClique;
    for(j=0;j<(c->SCLength);j++){
      /* processor s only contains elements s%M, s%M+M, s%M+2M,...*/
      /* so node i equals index i/M in array h1 */
      if( (h1[(k[j]/M)]==0) && (k[j]!=gmmdn) ){
        h1[(k[j]/M)]=1;
        h3[*length][rowid]=k[j];
        (*length)++;
      }
    }
    /* removeing subclique*/
    d=c->Next;
    Cliques[pos]=d;
    c->Next=NULL;
    free(c);
  }
  else{
    /* check if processor s contains a subclique */
    c=c->Next;
    while( (c!=NULL) && ((c->SCliqueName)!=i) ){
      d=c;
      c=c->Next;
    }
    if(c!=NULL){
      k=c->SClique;
      for(j=0;j<(c->SCLength);j++){
        /* processor s only contains elements s%M, s%M+M, s%M+2M,...*/
        /* so node i equals index i/M in array h1 */
        if( (h1[(k[j]/M)]==0) && (k[j]!=gmmdn) ){
          h1[(k[j]/M)]=1;
          h3[*length][rowid]=k[j];
          (*length)++;
        }
      }
      /* removing subclique*/
      d->Next=c->Next;
      c->Next=NULL;
      free(c);
    }
  }
}

```

```

    }
  }
}

void Merge(int s,int gmmdn){

/* We want to merge cliques which contain node node. We will
do this in parallel. We do this by first merging subcliques of
cliques that contain node locally per processor and the result
of this we merge column wise. Thus the new clique will be distributed
in subcliques over the columns */

/* we have int h1[V/M+1] int h2[E/N+2] initialized on 0.
note: (V/M+1)>(M*N). Furthermore we have array int h3[V/M+1][N].
gmmdn is the next node to be eliminated, thus we have to merge the cliques
containing that node */

struct Place *p;
int pos,n,pid,i,j,length,rowid, *k,size,f;
struct Name *d;

/*first find the processors that contain the cliquenames of cliques
containing node gmmdn and send this information to the processors in
the correct row using a 2 phase broadcast. First send the cliquenames
to different processors in the row, after that each processor in that
row sends the cliquenames it received to all other processors in that
row. Furthermore sstep 14 and sstep 15 determine the new cliquename
ncn */

if(degree!=0){

bsp_time();

bpsstep(14);
/* search in which column the cliquenames of gmmdn are located */
if(s%M==gmmdn%M){
  pos=gmmdn/M;
  p=Location[pos];
  /* if p!=Null then send the cliquename to all other processors
in the column so that a global new cliquename can be determined. If
p=null then send a -1 */
  if(p!=NULL)
    f=p->CliqueName;
  else
    f=-1;
  for(i=0;i<N;i++){
    pid=i*M+s%M;
    if(pid!=s)
      bspstore(pid,&f,&column[s/M][0],SZINT);
    else

```

```

        column[s/M][0]=f;
    }
    /* send cliquenames to processor in the correct row
    (phase I)*/
    i=0;
    while(p!=NULL){
        n=p->CliqueName;
        pid=(s/M)*M+i%M; /* (s/M)*M (same row) */
        pos=(i/M)*M+pid%M;
        bspstore(pid,&n,&h2[pos],SZINT);
        p=p->Next;
        i++;
    }
}
bspstep_end(14);

form(bsp_dtime(),14);

/* determine which cliquename will be the cliquename of the new clique
and send this information to all processor in your row. Furthermore send
all information a processor contained about the cliquenames of cliques
that will be used for merging to all other processors in the same row */

bsp_time();

bspstep(15);
if(s%M==gmmdn%M){
    ncn=column[0][0];
    for(i=1;i<N;i++){
        if( ( column[i][0]<ncn) || (ncn==-1) ) && (column[i][0]!=-1) )
            ncn=column[i][0];
    }
    for(i=0;i<M;i++){
        pid=(s/M)*M+i;
        if(pid!=s)
            bspstore(pid,&ncn,&ncn,SZINT);
    }
}
i=0;
pos=(s%M)+i*M;
while( (h2[pos]!=0) && (pos<(E/N+1)) ){
    for(j=0;j<M;j++){
        pid=(s/M)*M+j;
        if(pid!=s)
            bspstore(pid,&h2[pos],&h2[pos],SZINT);
    }
    i++;
    pos=(s%M)+i*M;
}
bspstep_end(15);

```

```

form(bsp_dtime(),15);

/*each processor will now look into its hash array Cliques if it
contains subcliques if so merge al these subcliques together
and remove these subcliques. Each processor can contain
(V/M+1) elements at maximum. Thus we can use array h1 for marking
them, in which index i equals node s%M+i*M note:s=(s/M,s%M) */

bsp_time();

bpsstep(16);
rowid=s/M; /* row which processor s is contained in */
length=0;
i=0;
while(h2[i]!=0){
pos=h2[i]/N;
pos %=(E/(N*M)+1);
SearchSC(pos,s,h2[i],&length,rowid);
i++;
}
mr=ncn%M;
/* resetting array h1 in processors that are not in the row where
the new clique will be constructed */
if(s/M!=mr){
for(i=0;i<length;i++){
pos=h3[i][rowid]/M;
h1[pos]=0;
}
/* pid is the processor in the column where we send our subclique
to */
pid=mr*M+s%M;
for(i=0;i<length;i++){
bspstore(pid,&h3[i][rowid],&h3[i][rowid],SZINT);
h3[i][rowid]=-1;
}
}
bpsstep_end(16);

form(bsp_dtime(),16);

/* the processors in row mr have received the subcliques from
the other processors in the same column. These are stored in array
h3. They now gonna merge these to become a new subclique and will
store these subcliques in array Cliques. The name of the new
clique is ncn */

bsp_time();

bpsstep(17);

```

```

if(rowid==mr){
  for(i=0;i<N;i++){
    if(i!=rowid){
      /* check the other subcliques for new elements */
      j=0;
      while(h3[j][i]!=-1){
        if(h1[(h3[j][i]/M)]==0){
          h3[length][rowid]=h3[j][i];
          h1[(h3[j][i]/M)]=1;
          length++;
        }
        h3[j][i]=-1;
        j++;
      }
    }
  }
  /* create new subclique */
  if(length!=0){
    size=length*(sizeof(int));
    k=(int *)malloc(size);
    for(i=0;i<length;i++){
      k[i]=h3[i][rowid];
      h1[((h3[i][rowid])/M)]=0;
      h3[i][rowid]=-1;
    }
    d=(struct Name *)malloc(sizeof(struct Name));
    d->SCliqueName=ncn;
    d->SCLength=length;
    d->SClique=k;
    pos=ncn/N;
    pos %=(E/(N*M)+1);
    d->Next=Cliques[pos];
    Cliques[pos]=d;
  }
}
/* All cliquenames of cliques that contain the eliminated node
wil now be flagged in array CliqueCount. Processors that contain
the same cliquenames of this set stored in array h2 are located
in the same row. Every processor in the same row will flag part
of the cliquenames */
i=0;
while( (h2[s%M+i*M]!=0) && ((s%M+i*M)<(E/N+1)) ){
  pid=h2[s%M+i*M]%(N*M);
  pos=h2[s%M+i*M]/(N*M);
  j=1;
  bspstore(pid,&j,&CliqueCount[pos],SZINT);
  i++;
}
i=0;
while(h2[i]!=0){

```

```

        h2[i]=0;
        i++;
    }
    bspsstep_end(17);

    form(bsp_dtime(),17);

}
}

```

### Degree update

```

void SearchCP(int pos,int s,int i,int *length,int rowid){

/* this procedure will look into hash array Cliques[pos] if it
contains a subclique of clique i. It will put elements of
this subclique (if there is any) into array h3[][rowid]
if and only if there not already contained in this array.
To check this we use the array h1 */

int j,*k;
struct Name *c,*d;

c=Cliques[pos];
while( (c!=NULL) && ((c->SCliqueName)!=i) ){
    c=c->Next;
}
if(c!=NULL){
    k=c->SClique;
    for(j=0;j<(c->SCLength);j++){
        /* processor s only contains elements s%M, s%M+M, s%M+2M,...*/
        /* so node i equals index i/M in array h1 */
        if(h1[(k[j]/M)]==0){
            h1[(k[j]/M)]=1;
            h3[*length][rowid]=k[j];
            (*length)++;
        }
    }
}
}
}

```

```

void Update(int s,int *lmmdn){

/* check which processor contains information about the
cliquesnames of the node that need a degree update.
Search its list Location to see which cliques ought to
be removed. We do this by first using the flagged array
CliqueCount to see which cliques are removed. We also

```

```

insert the new cliquename always. A list doesn't have
to contain this new name, so we insert it anyway.
*/

int pos,pid,pid1,pos1,CName,i,j,length,k,l;
struct Place *p,*z;
struct DegLoc *d,*e;

bsp_time();

bpsstep(23);
if( (cand!=-1) && (s%M==cand%M) ){
    pos=cand/M;
    p=Location[pos];
    while(p!=NULL){
        CName=p->CliqueName;
        pid1=CName%(N*M);
        pos1=CName/(N*M);
        /* if p->flag becomes 1 than this cliquename has to be
        removed */
        bspfetc(pid1,&CliqueCount[pos1],&(p->flag),SZINT);
        p=p->Next;
    }
}
bpsstep_end(23);

form(bsp_dtime(),23);

/* we will now look which cliques ought to be removed and which one
we will use in making the degree update for the node. These cliques
we will send to the first processor in the correct row */

bsp_time();

bpsstep(24);
if( (cand!=-1) && (s%M==cand%M) ){
    i=0;
    p=Location[pos];
    z=p;
    while(p!=NULL){
        /*check if clique have to be removed */
        if( (p->flag)==1 ){
            if(p==z){
                p=p->Next;
                z->Next=NULL;
                free(z);
                z=p;
                Location[pos]=p;
            }
            else{

```

```

        z->Next=p->Next;
        p->Next=NULL;
        free(p);
        p=z->Next;
    }
}
else{
    /* send clique to the correct processor in the correct row */
    pid1=(s/M)*M+i%M;
    pos1=(i/M)*M+pid1%M;
    bspstore(pid1,&(p->CliqueName),&h2[pos1],SZINT);
    i++;
    if(p==z)
        p=p->Next;
    else{
        p=p->Next;
        z=z->Next;
    }
}
}
/* send also ncn to the right processor and insert this cliquename
in the list Location. If it were already contained in this list
we removed it, so we always have to insert it */
if(s/M==ncn%M){
    p=(struct Place *)malloc(sizeof(struct Place));
    p->Next=Location[pos];
    p->flag=0;
    p->CliqueName=ncn;
    Location[pos]=p;
    pid1=(s/M)*M+i%M;
    pos1=(i/M)*M+pid1%M;
    bspstore(pid1,&ncn,&h2[pos1],SZINT);
}
}
bspsstep_end(24);

form(bsp_dtime(),24);

/* we wil now send the cliquenames contained in the processors
in array h2 to all other processors in the same row */

bsp_time();

bspsstep(25);
i=0;
pos=(s/M)+i*M;
while( (h2[pos]!=0) && (pos<(E/N+1)) ){
    for(j=0;j<M;j++){
        pid=(s/M)*M+j;
        if(pid!=s)

```

```

        bspstore(pid,&h2[pos],&h2[pos],SZINT);
    }
    i++;
    pos=(s%M)+i*M;
}
bpsstep_end(25);

form(bsp_dtime(),25);

/* every processor will now look into array Cliques to see if
it contains subcliques. These will be merged locally. */

bsp_time();

bpsstep(26);
length=0;
i=0;
while(h2[i]!=0){
    pos=h2[i]/N;
    pos %=(E/(N*M)+1);
    SearchCP(pos,s,h2[i],&length,s/M);
    i++;
}
if(s/M!=ncn%N){
    /* resetting array h1 in processor that are not in the
row where the new degree will be calculated */
    for(i=0;i<length;i++){
        pos=h3[i][s/M]/M;
        h1[pos]=0;
    }
    /* pid is the processor in the column where we send our subclique
to */
    pid=(ncn%N)*M+s%M;
    for(i=0;i<length;i++){
        bspstore(pid,&h3[i][s/M],&h3[i][s/M],SZINT);
        h3[i][s/M]=-1;
    }
}
bpsstep_end(26);

form(bsp_dtime(),26);

/* the processors in row ncn%N have received the subcliques from
the other processors in the same column. These are stored

```

```

in array h3. They now gonna merge these to become a new*/

bsp_time();

bpsstep(27);
if(s/M==ncn%M){
  for(i=0;i<N;i++){
    if(i!=s/M){
      /* check the other subcliques for new elements */
      j=0;
      while(h3[j][i]!=-1){
        if(h1[(h3[j][i]/M)]==0){
          h3[length][s/M]=h3[j][i];
          h1[(h3[j][i]/M)]=1;
          length++;
        }
        h3[j][i]=-1;
        j++;
      }
    }
  }
  j=0;
  /* length represents the subdegree of the node adding up all subdegrees
  will result in the new degree of the node */
  /* reset array h1 and h3[][s/M] */
  while(h3[j][s/M]!=-1){
    h1[(h3[j][s/M]/M)]=0;
    h3[j][s/M]=-1;
    j++;
  }
  pid=(ncn%M)*M;
  bspstore(pid,&length,&row[s%M][0],SZINT);
}
/* reset array h2 */
j=0;
while(h2[j]!=0){
  h2[j]=0;
  j++;
}
bpsstep_end(27);

form(bsp_dtime(),27);

/* calculate new degree */

bsp_time();

bpsstep(28);

```

```

    if(s==(ncn%N)*M){
        mr=0;
        for(i=0;i<M;i++){
            mr +=row[i][0];
            /* because we also counted the node itself in its degree update */
        }
        mr -=1;
    }
    bspstep_end(28);

    form(bsp_dtime(),28);

    bsp_time();

    bspstep(29);
    if( (cand!=-1) && (s==cand%(M*N)) ){
        pid=((ncn%N)*M);
        /* cand has the value of the node that gets a degree update */
        bspfetc(pid,&mr,&mr,SZINT);
    }
    bspstep_end(29);

    form(bsp_dtime(),29);

    /* adjust array vertices and degree because degree maybe changed */

    bsp_time();

    bspstep(30);
    if( (cand!=-1) && (s==cand%(M*N)) ){
        /* local node value*/
        pos=cand/(M*N);
        /* check if degree is changed*/
        if(mr!=Vertices[pos].Degree){
            pos1=(Vertices[pos].Degree)/(V/(N*M)+1);
            d=Degree[pos1];
            e=d;
            /* search hash array for representative of nodes that
            had the same degree as cand before degree update*/
            while( (d!=NULL) && ((d->Degree)!=Vertices[pos].Degree) ){
                if(d!=Degree[pos1])
                    e=e->Next;
                d=d->Next;
            }
            if(d==NULL)
                printf("error\n");
            else{
                /* cand was a representative so we have to remove it*/
                /* all nodes in array Degree are local node values. cand
                is a global node value so we have to take care of that */
            }
        }
    }

```

```

if((d->Node)*M*N+s==cand){
    /* check if cand is the only node*/
    if(Vertices[pos].NextNode==pos){
        /*remove this pointer*/
        if(d==e){
            Degree[pos1]=d->Next;
            d->Next=NULL;
            free(d);
        }
        else{
            e->Next=d->Next;
            d->Next=NULL;
            free(d);
        }
    }
    else{
        k=Vertices[pos].NextNode;
        l=Vertices[pos].PrevNode;
        d->Node=k;
        Vertices[k].PrevNode=l;
        Vertices[l].NextNode=k;
    }
}
else{
    k=Vertices[pos].NextNode;
    l=Vertices[pos].PrevNode;
    Vertices[k].PrevNode=l;
    Vertices[l].NextNode=k;
}
}
pos1=mr%(V/(M*N)+1);
d=Degree[pos1];
while( (d!=NULL) && ((d->Degree)!=mr) )
    d=d->Next;
if(d==NULL){
    /*cand is the only node with this degree*/
    d=(struct DegLoc *)malloc(sizeof(struct DegLoc));
    d->Next=Degree[pos1];
    d->Degree=mr;
    d->Node=pos;
    Degree[pos1]=d;
    Vertices[pos].Degree=mr;
    Vertices[pos].NextNode=pos;
    Vertices[pos].PrevNode=pos;
}
else{
    l=Vertices[d->Node].PrevNode;
    Vertices[d->Node].PrevNode=pos;
    Vertices[pos].NextNode=d->Node;
    Vertices[pos].PrevNode=l;
}

```

```

        Vertices[l].NextNode=pos;
        Vertices[pos].Degree=mr;
    }
}
if( (Vertices[pos].Degree)<(Vertices[*lmmdn].Degree) )
    (*lmmdn)=pos;
}
bspsstep_end(30);

form(bsp_dtime(),30);

}

void BSPDegreeUpdate(int s,int *lmmdn){

/* every processor knows the new clique name ncn, thus every
processor knows if it is contained in the row where clique
ncn is located. Furthermore, array h1 and h2 are initialize on
0 and h3 on -1 */

int rowid,colid,pos,l,*k,pid,i,j,subl,node,h;
struct Name *d;

if(degree!=0){

bsp_time();

bspsstep(20);
rowid=s/M; /* row coordinate of processor s*/
colid=s%M; /* collumn coordinate of processor s*/
if(rowid==(ncn%M)){
/* find subclique of ncn and length of that subclique */
pos=ncn/N;
pos %=(E/(N*M)+1);
d=Cliques[pos];
while( (d!=NULL) && (d->SCliqueName!=ncn) )
    d=d->Next;
if(d!=NULL){
    l=d->SCLength;
    k=d->SClique;
}
else{
    l=0;
    k=NULL;
}
/*send length of subcliques to al other processors in the same row */
for(i=0;i<M;i++){
    pid=rowid*M+i;
    if(pid!=s)

```

```

        bspstore(pid,&l,&row[colid][0],SZINT);
    else
        row[colid][0]=1;
    }
}
bpsstep_end(20);

form(bsp_dtime(),20);

bsp_time();

bpsstep(21);
/* calculate total length of new clique and sublength, which is
the amount of nodes not located in this processor that
will have a degree update first */
if(rowid==(ncn%M)){
    totl=0;
    subl=0;
    for(i=0;i<M;i++){
        if(i<colid)
            subl+=row[i][0];
        totl+=row[i][0];
    }
    /* every processor will now know the length of the new clique.
every processor in the row which contains the new clique will
broadcast it to processors in it's row */
    for(i=0;i<N;i++){
        pid=colid+i*M;
        bspstore(pid,&totl,&totl,SZINT);
    }
}
cand=-1;
bpsstep_end(21);

form(bsp_dtime(),21);

/* every processor now knows the number of nodes that need a degree
update */
j=0;
for(i=0;i<totl;i++){

    bsp_time();

    bpsstep(22);
    if( (rowid==(ncn%M)) && (i>(subl-1)) && (i<(subl+1)) ){
        node=k[j]; /* node that will get a degreeupdate */
        /* signal processors in the column that contains the cliquenames
of node */
        for(h=0;h<N;h++){
            pid=(node%M)+h*M;

```

```

        if(pid==s)
            cand=node;
        else
            bspstore(pid,&node,&cand,SZINT);
    }
    pid=(node%(N*M));
    if(pid==s)
        cand=node;
    else
        bspstore(pid,&node,&cand,SZINT);
    j++;
}
bpsstep_end(22);

form(bsp_dtime(),22);

Update(s,lmmdn);
cand=-1;
}

}
}

```

### Remove minimum degree node

```

void Remove(int s,int *lmmdn){
/* we want to remove node gmmdn out of the set of nodes
that still have to be ordered and we want to reset arrays
CliqueCount, and remove the list of cliquenames of gmmdn*/

int pos,pos1,pid1,deg,k,l;
struct DegLoc *d,*e;
struct Place *p;

bsp_time();

bpsstep(31);
if(s==gmmdn%(M*N)){
/* local node value */
pos=gmmdn/(M*N);
deg=Vertices[pos].Degree;
pos1=deg%(V/(M*N)+1);
d=Degree[pos1];
e=d;
/* remember d->Node is the local node value */
while( (d!=NULL) && (((d->Node)*M*N+s)!=gmmdn) ){
if(d!=Degree[pos1])
e=e->Next;
}
}
}
}

```

```

    d=d->Next;
  }
if(d==NULL){
  k=Vertices[pos].NextNode;
  l=Vertices[pos].PrevNode;
  Vertices[k].PrevNode=l;
  Vertices[l].NextNode=k;
  Vertices[pos].NextNode=-2*V;
  Vertices[pos].PrevNode=-2*V;
  Vertices[pos].Degree=-2*V;
}
else{
  /* check if gmmdn is the only node with this deg */
  if(Vertices[pos].NextNode==pos){
    /* remove this pointer */
    if(d==e){
      Degree[pos1]=d->Next;
      d->Next=NULL;
      free(d);
    }
    else{
      e->Next=d->Next;
      d->Next=NULL;
      free(d);
    }
  }
  else{
    k=Vertices[pos].NextNode;
    l=Vertices[pos].PrevNode;
    d->Node=k;
    Vertices[k].PrevNode=l;
    Vertices[l].NextNode=k;
  }
  Vertices[pos].PrevNode=-2*V;
  Vertices[pos].NextNode=-2*V;
  Vertices[pos].Degree=-2*V;
}
/* find new minimum deg node in this processor */

if( (*lmmdn)==pos ){
  (*lmmdn)=-1;
  d=Degree[pos1];
  while( ((*lmmdn)==-1) && (deg<V) ){
    while( (d!=NULL) && (d->Degree!=deg) ){
      d=d->Next;
    }
    if(d!=NULL)
      (*lmmdn)=d->Node; /* local node value */
    else{
      deg++;
    }
  }
}

```

```

        pos1 =deg%(V/(N*M)+1);
        d=Degree[pos1];
    }
}
}
/* reset arrays CliqueCount */
if(s%M==gmmdn%M){
    pos=gmmdn/M;
    p=Location[pos];
    while(p!=NULL){
        pid1=(p->CliqueName)%(N*M);
        pos1=(p->CliqueName)/(M*N);
        l=0;
        bspstore(pid1,&l,&CliqueCount[pos1],SZINT);
        p=p->Next;
    }
    p=Location[pos];
    Location[pos]=NULL;
    free(p);
}
if( (lgthorder%(M*N))==s )
    Order[lgthorder/(M*N)]=gmmdn;
    lgthorder++;
    bspsstep_end(31);

form(bsp_dtime(),31);
}

```

### Time measurements

```

void form(float k,int i){

    sstime[i]=(ssstime[i]*super[i]+k)/(super[i]+1);
    super[i]++;
}

```

# Bibliography

- [1] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM, J. Sci. Comput.*, 16(6):1404–1411, November 1995.
- [2] R.H. Bisseling. Basic techniques for numerical linear algebra on bulk synchronous parallel computers. Technical Report 964, Mathematics and Computer Science Department, University of Utrecht, June 1996.
- [3] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures. *IFIP transactions A*, 5(1):509–514, 1994.
- [4] A. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.
- [5] G.H. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore and London, 1989.
- [6] Mark W. Goudreau, Jonathan M. D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel, and Thanasis Tsantilas. A proposal for the BSP Worldwide standard library. Technical report, Oxford Parallel, Oxford, UK, April 1996.
- [7] A.M. Erisman I.S. Duff and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, 1986.
- [8] J.W.H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, June 1985.
- [9] R. Miller. A library for bulk synchronous parallel programming. *General Purpose Parallel Computing*, pages 100–108, 1993.
- [10] T.A. Davis P. Amestoy and I.S. Duff. An approximate minimum degree ordering algorithm. Technical Report TR-94-039, Computer and Information Sciences Department, University of Florida, December 1994.
- [11] L.G. Valiant. A bridging model for parallel computation. *ACM communications*, 33:103–111, 1990.
- [12] M. Yannakakis. Computing the minimum fill in is np-complete. *SIAM J. Algebraic and Discrete Methods*, 2:77–79, 1981.