

Symbolic Sparse Cholesky Factorisation Using Elimination Trees

Jeroen van Grondelle

Frontpage illustration:
M.C. ESCHER,
“Holunderbaum”,
Woodcut, 1919.

Symbolic Sparse Cholesky Factorisation Using Elimination Trees

Jeroen van Grondelle

Master's thesis
Supervisor: Dr. Rob H. Bisseling
Department of Mathematics
Utrecht University
August 1999

Preface

This is my master's thesis, which I wrote at Utrecht University. It is the result of my graduation research in the period of July 1998 until August 1999 in the field of direct solution methods for linear systems.

This thesis deals with symbolic Cholesky factorisation. Cholesky factorisation of the matrix A determines a lower-triangular matrix L that satisfies $LL^T = A$ and is one of the steps in the process of solving symmetric linear systems. The structure of the Cholesky factor L is determined by a symbolic factorisation which is carried out before the actual numerical Cholesky factorisation.

In this thesis, the symbolic factorisation is split into two phases. First, the elimination tree of the matrix A is computed and then the structure of L is computed efficiently using this tree. This turns out to be faster than the old approach.

Chapter 1 gives an introduction to symbolic and numerical Cholesky factorisation and introduces a graph representation of sparse matrices. The factorisation algorithms are formulated within this graph model.

In Chapter 2, a general framework for sequential computation of elimination trees is proposed. A proof of correctness is given for this framework and several algorithms are developed within this framework.

In Chapter 3, an algorithm is developed that computes the elimination tree in parallel. Although no speedup is reached, it performs better than a parallel approach by Gilbert and Zmijewski.

Chapter 4 introduces algorithms for the symbolic factorisation based on the elimination tree. They take advantage of the information of the elimination

tree and eliminate the overhead of the conventional algorithm. Also, parallel algorithms are proposed.

In Chapter 5, experimental results are given for the parallel tree computation and parallel symbolic factorisation. Combined, they give a parallel symbolic factorisation algorithm that gives modest speedup.

All the experiments were carried out on the Cray T3E at HPaC in Delft, through a grant by the NCF.

There is a number of people I would like to thank here. First of all, I thank my thesis supervisor Rob Bisseling for all his advice. I especially appreciated all the “working sessions”, during which a lot of the ideas in this thesis emerged. I also want to thank my fellow students at room 606, Gerrie de Jong, Christiaan Kuiper, Mathilda Roest and Arne Oostveen, for the nice time we had while writing our theses. Furthermore, I thank André de Meijer, who maintains the computer systems at the institute. I really enjoyed working part-time with him at the system administration department last year.

Jeroen van Grondelle
Utrecht, August 1999.

Contents

1	Introduction	1
1.1	Cholesky factorisation	1
1.2	Numerical factorisation	1
1.3	Symbolic factorisation	2
1.4	Sequential algorithms	3
1.4.1	A graph representation of symbolic matrices	3
1.4.2	A basic algorithm	4
1.4.3	Fast symbolic Cholesky factorisation	5
2	Sequential construction of elimination trees	7
2.1	Introduction	7
2.2	Computing elimination trees	9
2.3	Proof of correctness	10
2.4	Strategies for computing the elimination tree	12
2.4.1	Computation by symbolic factorisation	12
2.4.2	Fully sorted ancestors	13
2.4.3	Partially sorted ancestors	14
2.4.4	Liu's algorithm	15
2.4.5	Liu's algorithm with path compression	16
2.4.6	A generalisation of Liu's algorithm	17
2.5	Experimental results	18
2.6	Liu's algorithms and processing order	19
2.6.1	Liu's original algorithm	20
2.6.2	Liu with path compression	20
2.7	Equivalence of Liu and sorted ancestors	22
3	Parallel construction of elimination trees	25
3.1	Gilbert and Zmijewski's method	25
3.2	A new wavefront method	27

3.3	Sparse header storage	29
3.4	Choosing parameters	31
3.4.1	Time analysis	31
3.4.2	Optimal choices for parameters α and β	34
3.4.3	Heuristics for estimating α and β	35
3.4.4	Implicit bucket sort	36
3.4.5	Communication load balance	36
3.5	Experimental results	37
4	Symbolic factorisation using the elimination tree	39
4.1	An alternative storage scheme for the elimination tree	39
4.2	A left-looking algorithm	40
4.3	Calculation based on row structure	42
4.4	Parallel symbolic factorisation	44
4.4.1	Parallel left-looking algorithm	45
4.4.2	Parallel row structure algorithm	46
4.4.3	Load balance	46
4.5	Sequential results	46
5	Parallel results	49
5.1	Computing trees	49
5.1.1	Gilbert & Zmijewski's method	49
5.1.2	The wavefront method	50
5.2	Comparing the methods	52
5.3	Symbolic factorisation	53
5.4	Effects of parameters	54
	Conclusions and future work	57
A	Sorting methods	59
A.1	Insertion sort	59
A.2	Quicksort	60
A.3	Comparison of insertion sort and quicksort	60
A.4	Partial sorting	61
B	The BSP model	63
B.1	A BSP computer	63
B.2	A BSP algorithm	64
B.3	The cost model	64
B.4	Benchmarks of the Cray T3E and IBM SP	66

<i>CONTENTS</i>	vii
C Test matrices	67
Bibliography	69
List of algorithms	71

Chapter 1

Introduction

1.1 Cholesky factorisation

Cholesky factorisation is a technique for solving linear systems $Ax = b$ where A is a positive definite symmetric matrix. These computations appear frequently in for instance the interior point method. This iterative alternative to the simplex method is used for linear programming, a widely used optimisation technique.

Definition 1.1.1 (Cholesky factorisation) *Given a symmetric positive definite matrix A , the Cholesky factor L is the lower-triangular matrix that satisfies*

$$LL^T = A \tag{1.1}$$

After factoring A , we can first solve $Ly = b$ and then $L^T x = y$. Because the upper and lower triangular systems are easy to solve, Cholesky factorisation is a convenient way of solving a symmetric linear system $Ax = b$.

1.2 Numerical factorisation

We can calculate such a matrix L as follows. If we assume that (1.1) holds, then

$$a_{ij} = \sum_{k=0}^{n-1} L_{ik}L_{kj}^T = \sum_{k=0}^j l_{ik}l_{jk} = \sum_{k=0}^{j-1} l_{ik}l_{jk} + l_{ij}l_{jj} \tag{1.2}$$

where $0 \leq j \leq i < n$. For $i = j$ this leads to

$$l_{jj} = \left(a_{jj} - \sum_{k=0}^{j-1} l_{jk}^2 \right)^{\frac{1}{2}} \quad (1.3)$$

and for all $0 \leq j < i < n$ to

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=0}^{j-1} l_{ik} l_{jk} \right). \quad (1.4)$$

Algorithm 1.1 is a right-looking algorithm, based on equations (1.3) and (1.4). It is formulated in terms of dense matrices and is called right-looking because it adds a column to all the columns it should be added to, which are all on its right. A left-looking algorithm takes a column and adds to it all the columns that should be added to it. These columns are all on its left.

Algorithm 1.1 A dense numerical Cholesky factorisation algorithm

Input: $A = \text{lower}(A_0)$

Output: $A, A = L$ such that $LL^T = A_0$

```

for  $k := 0$  to  $n - 1$  do
   $a_{kk} := \sqrt{a_{kk}}$ 
  for  $i := k + 1$  to  $n - 1$  do
     $a_{ik} := a_{ik} / a_{kk}$ 

  for  $j := k + 1$  to  $n - 1$  do
    for  $i := j$  to  $n - 1$  do
       $a_{ij} := a_{ij} - a_{ik} a_{jk}$ 

```

1.3 Symbolic factorisation

Throughout this thesis, we will be factoring large, sparse matrices. Sparse matrices have many zero coefficients. In general, at most twenty percent of the entries of a sparse matrix will be nonzeros. Taking advantage of the sparsity of matrices allows us to compute Cholesky factors using far fewer

floating-point operations (flops) as we would need factoring dense matrices of the same size. Also we can store sparse matrices using less memory than their dense counterparts would use.

When factoring these matrices, we will see that Cholesky factors are often much denser than the original matrices. These new nonzeros, or fill-in, are for instance generated by adding two columns with different nonzero positions. Because we use data structures that only store the nonzeros, it is useful to know the structure of the Cholesky factor before factoring it. Then we can reserve space in our data structure for the fill-in.

Symbolic factorisation determines the structure of the Cholesky factor. Because we are not interested in the numerical values of the entries in the factor, this can be done much faster than a full numerical factorisation.

1.4 Sequential algorithms for symbolic factorisation

In the previous section we mentioned symbolic factorisation. In this section we deduce a fast symbolic algorithm from the numerical algorithm in the previous section.

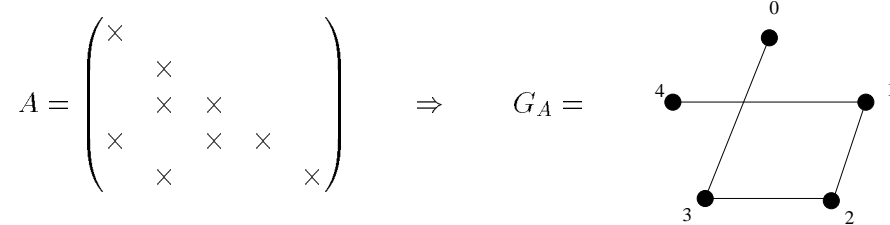
1.4.1 A graph representation of symbolic matrices

When dealing with symbolic factorisation, the algorithm can be formulated conveniently in the language of graph theory.

Definition 1.4.1 *An $n \times n$ matrix A induces the graph $G_A = (V_A, E_A)$, where $V_A = \{0, \dots, n-1\}$ and $E_A = \{(i, j) | 0 \leq i, j < n \wedge a_{ij} \neq 0\}$.*

E_A is called the set of edges and V_A the set of vertices. Because A is assumed symmetric, the graph need not be directed. Because of the symmetry of A , if $(i, j) \in E_A$, then $(j, i) \in E_A$. And because A is positive definite, all the diagonal elements of A are positive. As each vertex points to itself, these elements are generally omitted. Therefore we rewrite the definition of G_A for symmetric positive definite matrices A .

Definition 1.4.2 *A symmetric positive definite $n \times n$ matrix A induces the graph $G_A = (V_A, E_A)$, where $V_A = \{0, \dots, n-1\}$ and $E_A = \{(i, j) | 0 \leq j < i < n \wedge a_{ij} \neq 0\}$.*

Example 1.4.3**1.4.2 A basic algorithm**

Now we will transform algorithm 1.1 into a symbolic factorisation algorithm. To do this, we simply remove all operations from the algorithm that do not introduce new nonzeros or destroy existing nonzeros.

There are basically three operations in algorithm 1.1: A square root computation, a division by a_{kk} and the actual column addition. The first two operations clearly do not introduce new nonzeros, nor do they destroy them, so that the only operation we have to implement in symbolic factorisation is the column addition. Algorithm 1.2 implements this operation in graph notation.

Algorithm 1.2 A basic symbolic Cholesky factorisation algorithm

Input: $G_A = (V_A, E_A)$

Output: $G = (V, E)$ where $G = G_{L+L^T}$ with $LL^T = A$

```

for  $k := 0$  to  $n - 1$  do
  for all  $j : k < j < n \wedge (j, k) \in E$  do
    for all  $i : j < i < n \wedge (i, k) \in E$  do
       $E := E \cup \{(i, j)\}$ 
  
```

This algorithm has approximately the same complexity as the numerical factorisation, $\mathcal{O}(nc^2)^1$. We can reduce this runtime significantly, as we will

¹ c is the average number of nonzeros in each row.

show in the next section.

1.4.3 Fast symbolic Cholesky factorisation

In this section we will introduce a symbolic factorisation algorithm that is a factor c faster than the basic symbolic Cholesky factorisation algorithm. We will follow the treatment in [1]. But first we need a definition:

Definition 1.4.4 (Parent) *Every column is said to have a parent column. The parent of column k is defined as:*

$$\begin{aligned} \text{parent}(k) &= \min\{i : k < i < n \wedge l_{ik} \neq 0\} \\ &= \min\{i : (i, k) \in E\} \end{aligned} \quad (1.5)$$

Furthermore, $\text{parent}(k) = \infty$ if this minimum does not exist.

Corollary 1.4.5

$$\forall i \in \{0, \dots, n-1\} : i < \text{parent}(i) \quad (1.6)$$

Proof: The proof is trivial, since the parent of column i was defined as the row index of the first nonzero below the diagonal in column i . Apparently, this index will be greater than i . \square

We now use a theorem from Schreiber in [6] to reduce the amount of work. It is also formulated in [1].

Theorem 1.4.6 *Let A be an $n \times n$ sparse symmetric positive definite matrix with Cholesky factor L . Define the sparsity structure of column k , $0 \leq k < n$, of L by*

$$\text{Struct}(k) = \{i : k \leq i < n \wedge l_{ik} \neq 0\} \quad (1.7)$$

Let $0 \leq k < n$ and $\text{parent}(k) < \infty$. Then

$$\text{Struct}(k) \setminus \{k\} \subseteq \text{Struct}(\text{parent}(k)) \quad (1.8)$$

We do not prove this theorem here. A proof is given in [1].

Suppose we are about to perform step k of symbolic factorisation. According to the basic algorithm, we would have to add the edges (i, j) with

$$i > j \wedge i, j \in \text{Struct}(k) \setminus \{k\}$$

Suppose $Struct(k) = \{i_0, \dots, i_r\}$ with $i_0 = k < i_1 < \dots < i_r$. Now theorem 1.4.6 says that

$$Struct(k) \setminus \{k\} \subseteq Struct(parent(k)) = Struct(i_1)$$

So the edges (i, j) with

$$i > j \wedge i, j \in \{i_2, \dots, i_r\}$$

will be added at stage i_1 . So at stage k , we only have to add the edges (i, j) with

$$i \in \{i_2, \dots, i_r\} \wedge j = i_1$$

Applying this theorem repeatedly at each stage of the calculation reduces the runtime considerably. Algorithm 1.3 implements this technique.

Algorithm 1.3 The Fast Symbolic Cholesky factorisation algorithm

Input: $G_A = (V_A, E_A)$

Output: $G = (V, E)$ where $G = G_{L+L^T}$ with $LL^T = A$

for $k := 0$ **to** $n - 1$ **do**

$parent(k) = \min\{i : k < i < n \wedge (i, k) \in E\}$

for all $i : k < i < n \wedge (i, k) \in E$ **do**

$E := E \cup \{(i, parent(i))\}$

In each one of n steps, a column of c elements is added to its parent, therefore this algorithm has a runtime complexity of $\mathcal{O}(nc)$. This is a factor c faster than the basic algorithm from the previous subsection. From now on, when we refer to sequential symbolic Cholesky factorisation, we mean the factorisation by algorithm 1.3.

Chapter 2

Sequential construction of elimination trees

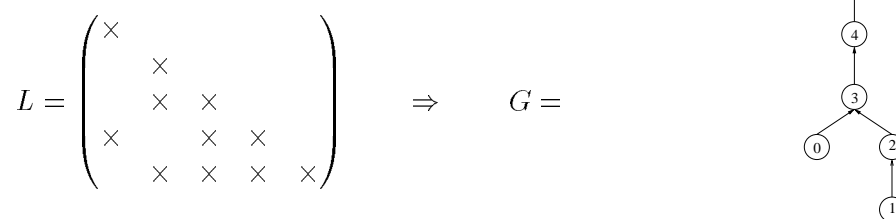
In Chapter 1 we saw that there exists a parent-child relation between columns. At the end of the sequential factorisation algorithm, the parent of each column is known. In this section, we will take a closer look at these relations and try to compute them in advance. We will use this information in Chapter 4 to speed up the computation of the symbolic Cholesky factor.

2.1 Introduction

Definition 2.1.1 (Elimination forest) *The elimination forest, associated with the Cholesky factor L , is the directed graph $G = (V, E)$ with $V = \{0, \dots, n - 1\}$ containing all column numbers of L and*

$$E = \{(i, j) \in V \times V : i = \text{parent}(j)\} \quad (2.1)$$

Example 2.1.2



Note that this is the symbolic Cholesky factor of example 1.4.3 and the forest happens to contain only one tree.

From now on we will assume that the elimination forest contains only one tree and refer to that tree as the elimination tree. This assumption is reasonable since if necessary, a simple preprocessing step divides the matrix into sub matrices that have a unique elimination tree.

Apart from the parent relation, the elimination tree contains a more general relation between columns.

Definition 2.1.3 (Ancestor relation) *Column i is said to be an ancestor of column j in the matrix A , if there is a path from j to i in the elimination tree of A .*

Obviously, the parent relation is also an ancestor relation.

We now formulate the central theorem on which the tree construction algorithms will be based:

Theorem 2.1.4 *Given p_1 and p_2 ancestors of column i and $p_1 < p_2$. Then p_2 is an ancestor of p_1 .*

Proof: There exists a path through the elimination tree from node i to the root. Since such a path is unique in a tree and there are also paths from i to both p_1 and p_2 and from these to the root, p_1 and p_2 have to be on this path from node i to the root. Consequently, there is either a path from p_1 to p_2 or from p_2 to p_1 . A path from p_2 to p_1 would imply that $p_2 < p_1$, which would contradict the assumption $p_1 < p_2$. Consequently, there has to be a path from p_1 to p_2 , implying that p_2 is an ancestor of p_1 . \square

This theorem can be generalised to the case where column i has r ancestors.

Theorem 2.1.5 *Given p_1, \dots, p_r a strictly monotonically increasing sequence of ancestors of column i . Then:*

$$p_i \text{ is ancestor of } p_{i-1}, \forall i \in \{2, \dots, r\}$$

Proof: This theorem can be proved by applying Theorem 2.1.4 $r - 1$ times, to the pairs (p_{i-1}, p_i) . \square

2.2 Computing elimination trees

There is a very close relation between the set of ancestors of a column and the nonzeros of this column. This is a result of the next theorem.

Theorem 2.2.1 *A nonzero a_{ij} of A with $i > j$ implies that i is an ancestor of j .*

Proof: This is a corollary to the theorem Schreiber proves in [6]. This theorem states that every nonzero l_{ij} in L implies that i is an ancestor of j . Our theorem is a direct result, since all nonzeros of A are also nonzeros in L . \square

Consequently, all the nonzeros in a column of A are ancestors of that column.

When we are computing the elimination tree, we are in fact looking in each column for the smallest element in the set of ancestors of that column. Theorems 2.1.4 and 2.1.5 introduce ways to move ancestors that are not the smallest, from their set to the ancestor set of a column with higher index. This suggests an algorithm for computing the elimination tree. At each step, we take two ancestors, r and s with $r < s$, from an ancestor set and use Theorem 2.1.4 to move s to the ancestor set of column r . We repeat this procedure, until we have at most one element left in all ancestor sets. Using Theorem 2.2.1, this procedure can be easily reformulated in matrix terms.

Definition 2.2.2 (Method for computing the elimination tree) *Given a matrix A , we compute its elimination tree as follows. At each step of the algorithm, we pick two nonzeros in a column, say a_{ij} and $a_{i'j}$ with $i < i'$. Using Theorem 2.1.4, we move nonzero $a_{i'j}$ to column i . This step is repeated until each column has at most one remaining nonzero below the diagonal.*

Note that the moves are different from addition in the sense that we are only interested in the structure of the matrix. If a nonzero is moved to a position that is already occupied, only one nonzero remains at that position.

Obviously, this method defines a whole class of algorithms. The definition does not specify the order in which the ancestors should be processed. You can look at the method as a game with some rules: You can move ancestors from one set to another according to Theorems 2.1.4 or 2.1.5. The goal is

to finish the game in as few moves as possible.

Later, we will discuss some strategies for this game, but in the next section we will prove that every algorithm following the rules computes the elimination tree correctly.

2.3 Proof of correctness

In the previous sections we have introduced a class of algorithms to compute the elimination tree. In this section we prove they are correct. To obtain the proof, we need one new theorem.

Theorem 2.3.1 *Let A be a sparse, symmetric matrix and suppose column k contains nonzeros a_{rk} and a_{sk} with $r < s$. The matrix A' is obtained from A by moving a_{sk} to column r based on theorem 2.1.4, so that A' contains nonzero a_{sr} . Then A and A' have the same elimination tree.*

Proof: To prove the theorem, we will look at the elimination tree as the set of parents. These parents are defined as the first nonzero elements below the diagonal of the Cholesky factor L . Our reference tree is the one deduced from L and we will compare it to the tree deduced from L' . We will compute the sparsity structures of L and L' with the Fast Symbolic Cholesky factorisation algorithm and show that they lead to the same elimination tree.

We now analyse the computation of L' from A' and compare it to the original computation of L from A . In the computation, we recognise three stages.

The first stage consists of the computation of the columns 0 to $k - 1$. Because they have not been altered, nothing changes in the first k steps of the algorithm.

The second stage consists of the steps k to $r - 1$. Although L and L' have different columns k , the parent information in these columns will not differ in the end. The only columns that can be affected by the difference between the columns k are the columns on the path of k to r on the elimination tree of L . Note however that both A and A' have a nonzero on row r of column k . Since this nonzero is added to all columns on this path, all possibly affected columns have parent smaller than or equal to r . Hence the possible differences on row s in these columns, with $s > r$, do not affect the parent

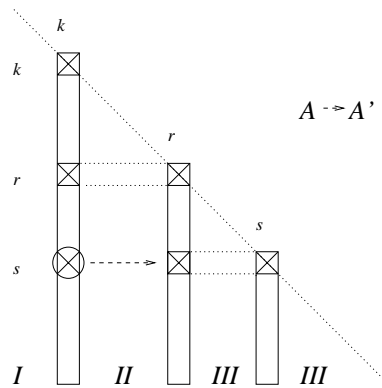


Figure 2.1: Creation of A' from A and the stages of their factorisation

information.

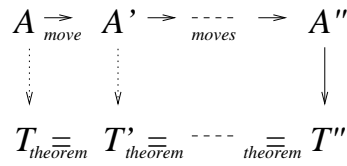
At column r , the possible difference is eliminated by the initial nonzero move. The only difference between A and A' in column r could be that a_{sr} has not become nonzero in A' in the factorisation process. But since a_{sr} was made nonzero by the initial nonzero move that created A' , column r is equal in A and A' . Consequently, from this point on the two computations will be identical.

We have proved that in all three stages of the computations the first nonzero below the diagonal of L' is equal to that of L , which implies by definition that their elimination trees are equal. \square

We can now prove the correctness of the algorithms:

Theorem 2.3.2 *The algorithms from definition 2.2.2 compute the elimination tree correctly.*

Proof: Notice that all the algorithms we have described transform their input matrix A into a matrix A'' with at most one nonzero below the diagonal in each column. This matrix A'' obviously has itself as Cholesky factor and therefore computing its elimination tree is trivial. Because A and A'' have the same elimination tree by repeated application of theorem 2.3.1, see Figure 2.2, all the algorithms in the new class of algorithms do actually compute the elimination tree correctly. \square

Figure 2.2: Computation of the elimination tree of A through A''

2.4 Strategies for computing the elimination tree

The tree computation method that was introduced in the previous section is actually a class of methods. We proved that all algorithms that move nonzeros according to the rules, correctly compute the elimination tree. Which pairs of nonzeros are picked and in what order is not specified. We will now suggest a few possible methods.

2.4.1 Computation by symbolic factorisation

We have seen that when a symbolic factorisation is done, the elimination tree is automatically computed. We will now show that this computation by the Fast Symbolic Cholesky algorithm can be formulated in terms of our moves. Adding column k to column $\text{parent}(k)$ is then viewed as applying theorem 2.1.4 to all nonzeros of column k greater than $\text{parent}(k)$. The theorem implies they are all ancestors of $\text{parent}(k)$ and can thus be stored in column $\text{parent}(k)$.

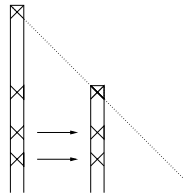


Figure 2.3: Theorem 2.1.4 used for symbolic factorisation

An implementation of this method is given in Algorithm 2.1. Given a linked list column representation of the matrix, adding two columns is a $\mathcal{O}(1)$ joining-operation. Removing duplicates costs $\mathcal{O}(c)$. This leads to $\mathcal{O}(nc)$ costs for the algorithm.

Algorithm 2.1 Tree computation by symbolic factorisation

Input: $A = (\mathcal{C}, n)$ where \mathcal{C}_j is the set of nonzeros i in column j with $i < j$

Output: $parent$

```

for  $k := 0$  to  $n - 1$  do
   $parent(k) := \min\{i : i \in \mathcal{C}_k\}$ 
   $\mathcal{C}_{parent(k)} := \mathcal{C}_{parent(k)} \cup \mathcal{C}_k \setminus \{parent(k)\}$ 

```

This may seem fast, but we will see later why this algorithm does not perform as well as the others.

2.4.2 Fully sorted ancestors

In the previous method, all nonzeros are moved to their parent column. But since only the smallest nonzero will remain in place eventually, only one move will be a final one and all but one nonzero will have to be moved again. Theorem 2.1.5 allows us to move nonzeros further to the right.

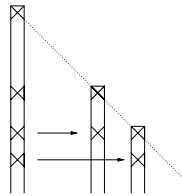


Figure 2.4: Theorem 2.1.5 moves nonzeros further to the right

Although this technique drastically reduces the number of moves, the set of ancestors has to be available in a sorted manner. This can be accomplished in several ways. The nonzeros could for instance be inserted in the ancestor set in an ordered manner. This is expensive however. Insertion of one element costs $\mathcal{O}(c)$, so processing a column would cost $\mathcal{O}(c^2)$. The total tree construction would cost $\mathcal{O}(nc^2)$

An alternative is sorting an ancestor set just before it is processed, as is implemented in Algorithm 2.2. All nonzeros are added to ancestor sets without

sorting or duplicate-checking. At stage k , the duplicates of column k are removed in $\mathcal{O}(c)$ and the column is sorted in $\mathcal{O}(c \log c)$ time. So the tree is computed in $\mathcal{O}(nc \log c)$ time.

Algorithm 2.2 Sorted-ancestors

Input: $A = (\mathcal{C}, n)$ where \mathcal{C}_j is the set of nonzeros i in column j with $i < j$

Output: $parent$

```

for  $k := 0$  to  $n - 1$  do
  sort( $\mathcal{C}_k$ )
  {  $\mathcal{C}_k$  is now available as  $\{c_0, \dots, c_r\}$ , with  $c_0 < c_1 < \dots < c_r$  }

   $parent(k) := c_0$ 
  for  $i := 1$  to  $|\mathcal{C}_k| - 1$  do
     $\mathcal{C}_{c_{i-1}} := \mathcal{C}_{c_{i-1}} \cup \{c_i\}$ 

```

Note that the full sort method seems to have a higher order runtime. But the comparison is more complex. Due to high fill-in, computation by factorisation suffers from higher values of c and is therefore slower than Sorted Ancestors.

Also, different sorting methods are preferred for different set sizes. See Appendix A for the methods used.

2.4.3 Partially sorted ancestors

While the previous method is very efficient in its number of moves, the sorting at each step of the algorithm makes it quite inefficient in time.

A possible alternative is partially sorting the set of ancestors. We have implemented a partial bucket sort which runs in linear time. It does not guarantee a completely sorted set of course. After partial sorting, Theorem 2.1.4 is applied to the last two elements of the set. The largest of the two is moved, whereas the smallest remains in the list. This is then repeated until the list contains at most one element.

The strategy aims at a trade-off: The speedup through faster sorting will compensate the increased number of moves required to compute the tree.

The partial bucket sort is described in appendix A.4.

2.4.4 Liu's algorithm

In [4], Liu proposes a tree construction algorithm. This algorithm processes the nonzeros of A by increasing row index. We will show here that it can be formulated in terms of moves. We will also show that Liu's algorithm can be generalised, such that row-wise processing is no longer necessary.

Definition 2.4.1 (Liu's algorithm) *Liu's algorithm builds a forest as follows: The nonzeros of A are processed per row. For each nonzero a_{ij} with $i > j$ in row i , node i is made the root of the subtree containing node j , if node i is not already found in this subtree. This is visualised in Figure 2.5.*

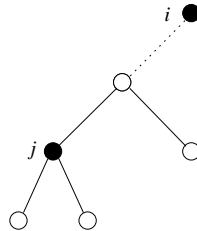


Figure 2.5: Liu's algorithm constructing a subtree

This algorithm is based on the fact that $a_{ij} \neq 0$ implies that i is an ancestor of j in the elimination tree. Because the nonzeros are processed per row, only ancestors smaller than or equal to i may have been inserted in the elimination tree so far. Therefore node i is either made the root of the subtree containing j at this step or it is already root. To do this, we start at node j and walk up through the tree until we find the root. If i is root, we are finished. Otherwise, we make i the root of this subtree. This algorithm is formulated in Algorithm 2.3.

This algorithm can easily be formulated within the moving scheme in matrix terms. We look at A with all the elements made invisible. The rows are now processed by increasing row index. Within a row, the nonzeros are made visible one by one. When a nonzero is visible and there is another off-diagonal nonzero in that column, Theorem 2.1.4 is applied and the nonzero with highest index is moved to the right. This procedure is repeated until

Algorithm 2.3 Liu's algorithm

Input: $A = (\mathcal{R}, n)$ where $\mathcal{R}_j = \{c_0, \dots, c_{|\mathcal{R}_j|-1}\}$ is the set of nonzeros i in row j with $i < j$

Output: *parent*

```
for  $j := 0$  to  $n - 1$  do
   $parent(j) := \infty$ 
  for  $i := 0$  to  $|\mathcal{R}_j| - 1$  do
     $r := c_i$ 
    while  $parent(r) \neq \infty$  do
       $r := parent(r)$ 
    if  $r \neq j$  do
       $parent(r) := j$ 
```

there is at most one nonzero below the diagonal in any column. Then the next nonzero of the row is processed.

2.4.5 Liu's algorithm with path compression

In Liu's algorithm, the most time-consuming operation is finding the root of the subtree that contains a certain node. Because the node we add is always the new root, we are only interested in finding the current root.

The original algorithm walks up the tree, beginning at node j , until it finds the root. Liu's algorithm with path compression maintains a second forest, the virtual forest, that is only used for root finding. The first forest, or real forest, is used to compute the elimination tree. In this forest the paths are constructed as in the original algorithm. In the virtual forest, the algorithm maintains shortcuts to the root; every element in the virtual forest points to an element on the path towards the root and as close to it as possible.

On inserting nonzero a_{ij} , the algorithm starts at node j of the virtual forest. While walking up through this forest, it sets the pointers of each node it passes to i , of which we know it is already root or will be root after the next step. When it finds the root, it inserts the proper relation to the real forest. A pseudo implementation is given in Algorithm 2.4

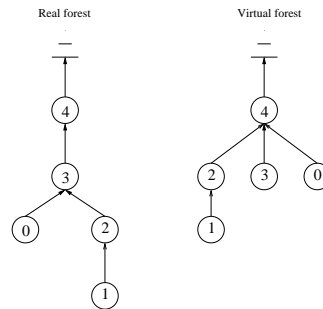


Figure 2.6: Example of a possible virtual forest

Algorithm 2.4 Liu's algorithm with path compression

Input: $A = (\mathcal{R}, n)$ where $\mathcal{R}_j = c_0, \dots, c_{|\mathcal{R}_j|-1}$ is the set of nonzeros i in row j with $i < j$

Output: *parent*

```

for  $j := 0$  to  $n - 1$  do
   $parent(j) := \infty$ 
   $virtual(j) := \infty$ 
  for  $i := 0$  to  $|\mathcal{R}_j| - 1$  do
     $r := c_i$ 
    while  $virtual(r) < j$  do
       $t := virtual(r)$ 
       $virtual(r) := j$ 
       $r := t$ 
    if  $virtual(r) = \infty$  do
       $virtual(r) := j$ 
       $parent(r) := j$ 

```

The extra information that is stored in the virtual forest can also be implemented using the moving scheme. This is done in Section 2.7

2.4.6 A generalisation of Liu's algorithm

We can generalise Liu's algorithm so that it no longer needs row-wise processing of nonzeros. If we look at the path from j to the root as a sorted

list of ancestors, we can apply Theorem 2.1.5 to this list with i inserted such that the list remains sorted. Suppose the path induces the ancestor list $\{p_1, \dots, p_n\}$, then applying the theorem to the list $\{p_1, \dots, p_k, i, p_{k+1}, \dots, p_n\}$ implies that p_{k+1} is an ancestor of i and i is an ancestor of p_k . Therefore, we can insert i in the tree on the path from j to the root between p_k and p_{k+1} .

Definition 2.4.2 (Method generalising Liu’s algorithm) *The general method based on Liu’s algorithm processes the nonzeros of A in an arbitrary order. On processing a_{ij} , node i is inserted in the path from j to the root, such that the nodes on this path remain sorted in increasing order.*

The difference in runtime between Liu’s original algorithm and the generalised version is not very large, but the path compression does not work anymore for the generalised version. As Liu without path compression is not very competitive in runtime, this generalised version will not be very useful.

2.5 Experimental results

We have implemented four of the proposed strategies¹. We have measured the runtime of these four algorithms for several matrices on one node of a Cray T3E. The results are in Table 2.1.

Matrix	Fully sorted	Partially sorted	Liu	Liu with PC
1138 BUS	23.28	44.10	32.23	8.43
1138 BUS (mmd)	12.78	15.88	5.82	6.75
900 BUS	48.36	113.07	55.14	34.52
900 BUS (mmd)	40.48	74.39	38.68	14.64
BCSSTK18	1077.58	2255.24	9422.04	297.93
BCSSTK26	172.71	295.75	421.26	53.42
BLCKHOLE	20.80	29.88	11.89	10.42
LSHP3466	27.75	33.87	14.98	15.60
BCSPWR10	181.18	561.35	2292.74	55.61

Table 2.1: Wall clock time in ms on a Cray T3E

¹Both the conventional strategy and Liu’s generalised algorithm were not implemented, because they are equal in complexity to the Fast Symbolic Cholesky factorisation.

We have also measured the number of moves required to compute the elimination tree. These results are in Table 2.2.

Matrix	Fully sorted	Partially sorted	Liu	Liu with PC
1138 BUS	2201	2945	44134	2201
1138 BUS (mmd)	889	923	2040	889
900 BUS	4118	8225	76531	4118
900 BUS (mmd)	3845	4977	49300	3845
BCSSTK18	88665	166600	14831191	88665
BCSSTK26	14695	20545	631953	14695
BLCKHOLE	1354	1603	7870	1354
LSHP3466	1363	1471	6556	1363
BCSPWR10	19722	41929	3584004	19722

Table 2.2: Number of moves

Note that these statistics have to be interpreted with some caution. The moves in the different algorithms may have different costs. And sorting operations are not represented in Table 2.2.

Liu's algorithms process nonzeros by increasing row index. The algorithms do not specify the order of processing within a row. The results shown above are measured by processing nonzeros in one row by increasing column index. But in Section 2.6 we will show that the number of moves is invariant under the order of processing within the rows.

Table 2.2 clearly indicates that the number of moves used by the Sorted Ancestors algorithm and Liu's algorithm with path compression is the same. In Section 2.7, we will prove this equivalence.

2.6 Liu's algorithms and processing order

Liu's original algorithm and its variant using path compression do not specify the order in which elements of a row should be processed. In this section we show that the number of moves made by the algorithm is invariant under this order.

2.6.1 Liu's original algorithm

We look at the algorithm in terms of matrices. Every nonzero has to be moved a number of times before it is moved to a column where it is the only nonzero below the diagonal. Suppose we are moving nonzero a_{ij} , then this nonzero will be moved to the column that belongs to the last node before node i on the path from node j to i .

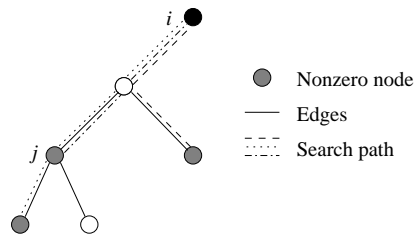


Figure 2.7: Every nonzero generates an independent path

The last move either adds the nonzero to this column, if node i is not the root yet, or finds that it is already nonzero, if node i is already root. Since this both counts as a move, the number of moves required to process a nonzero does not depend on the order of processing. Processing a_{ij} costs exactly the length of the path from j to i in the elimination tree, as is visualised in Figure 2.7.

2.6.2 Liu with path compression

To prove that the number of moves for Liu's algorithm with path compression is invariant is more complex. The number of moves required for processing row k depends not only on the structure of row k but also on the state of the virtual forest after the processing of row $k - 1$. So first we show that the state of the virtual forest after processing row $k - 1$ is independent of the order in which row $k - 1$ was processed.

Theorem 2.6.1 *The state of the virtual forest at the end of processing a row is independent of the order in which the row was processed*

Proof:

To prove this, we use an inductive argument. Suppose the theorem holds after processing row $k - 1$. We will now show what processing row k does

to the virtual forest.

While processing row k , for every nonzero a_{kj} we start at node j in the virtual forest and follow the path until we find the root. All edges passed on the way are set to point to the new root, which is node k . So afterwards, all the edges reached through starting at any node that corresponds to a nonzero points to k and all the other edges are unaltered. Clearly, the order in which the edges are reached has no effect.

Obviously, the theorem holds for row 0 and therefore the theorem is proved by induction. \square

Consequently, after completing processing row $k - 1$, the state of the virtual forest is completely determined by the nonzeros processed so far, rather than the order in which this was done. Looking closer at the processing of row k , we can now prove the next theorem.

Theorem 2.6.2 *The number of moves that Liu's algorithm with path compression makes while processing a row is independent of the order in which the nonzeros are processed.*

Proof: To process row k , we start at every node that corresponds to a nonzero and follow the path through the virtual forest until we find the root. Because the virtual forest is adjusted at each step, it is hard to count the moves. Instead of adjusting the virtual forest at each step, we can leave it unaltered but mark the nodes we pass. A marked node always points to node i . So we start at j and walk up the virtual forest until we find a marked node. From there, it is one move to the root.

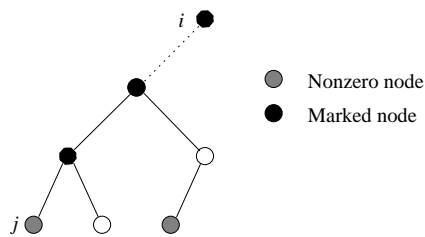


Figure 2.8: Marking passed nodes

After processing row k all nodes that are on paths from nonzero-nodes to

the root are marked. Because all the edges are followed only once, we can compute the number of moves. It is equal to the number of edges in the subtree generated by the nonzero nodes and their ancestors, plus the number of nonzeros in row k . The latter term accounts for the jumps from marked nodes to the root in the virtual forest. This sum is obviously not dependent on the processing order. \square

2.7 Equivalence of Liu and sorted ancestors

The results in Table 2.2 in Section 2.5 clearly suggest that the number of moves required for Liu's algorithm with path compression is equal to the number of moves required for the sorted ancestors algorithm. In this section we will prove this.

To prove this, we will look closer at the matrix interpretation of Liu's algorithm. We show how the virtual forest and the actual forest are stored in one matrix and what the effects are when a nonzero is moved. We will use this to show that each nonzero is moved in exactly the same way in both algorithms.

Theorem 2.7.1 *Liu's algorithm with path compression and the sorted ancestors algorithm move a nonzero a_{ij} in A to the same column.*

We look at the matrix interpretation of Liu's original algorithm again. All nonzeros of A are made invisible. After that, A is processed by rows. In each row, all nonzeros are made visible one by one. Once a nonzero becomes visible, it is moved as far as possible to the right, based on the nonzero above the current row. Of course, each column will contain either one or no nonzeros above the current row. If a column contains a nonzero, this has to be the parent.

Liu's algorithm with path compression maintains one extra pointer per column in the virtual forest. This pointer can be stored in the matrix in the same manner as the parent is stored, namely in the presence of a virtual forest nonzero, which lies below the parent nonzero or equals it. As a consequence, each column can now contain either one, two or no off-diagonal nonzeros. An example is shown in Figure 2.9.

In order to see why the same moves are made by the two algorithms, it is

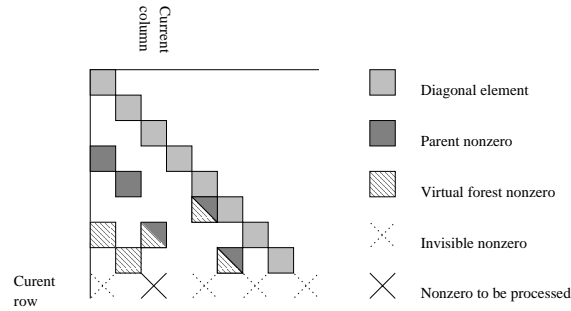


Figure 2.9: A matrix interpretation of Liu's algorithm with path compression

important to realise what information is stored in the virtual forest. On processing a row, the virtual forest is updated by making all passed nodes point to the node that corresponds to the index of the current row. The matrix equivalent of this operation is that the virtual forest nonzero in the current column is moved to the location of the nonzero that is being processed, as can be seen in Figure 2.10

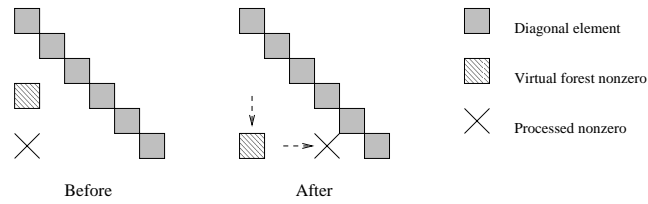


Figure 2.10: A move by Liu with path compression

As a consequence, the virtual forest stores the row index of the last element that is processed in each column. Since Liu processes the rows by increasing row index, the largest row index before the current row is stored. Thus, a nonzero a_{ij} is moved to the column that corresponds to the nonzero with largest row index that precedes nonzero a_{ij} in column j . This is the same as in the Sorted-ancestors algorithm. \square

Obviously, the fact that the number of moves required by both algorithms is the same follows directly from this theorem.

Chapter 3

Parallel construction of elimination trees

In this chapter we look at algorithms for computing the elimination tree in parallel. In the first section we study a technique proposed by Zmijewski and Gilbert in [9]. In section 3.2, we develop a new, more scalable algorithm.

3.1 Gilbert and Zmijewski's method

In [9], the elimination forest is calculated in parallel. The matrix A is distributed over the processors. Each processor i runs Liu's algorithm on its part of A and generates a vector $parent_i$. These are communicated and merged. This can be done pairwise, in $\log_2 p$ steps.

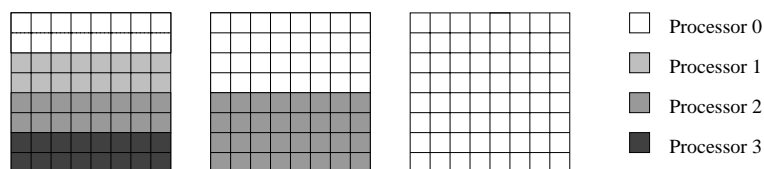


Figure 3.1: Zmijewski's merging scheme

The merging is done by putting the elements of the vectors to merge back in a matrix and run Liu again. This can best be explained if we look at the moving-game from Section 2.2 again. Suppose we have distributed rows in blocks. Then, every processor can move nonzeros based on the other nonzeros in its block. So eventually, every column in its block will contain

Algorithm 3.1 Merging partial forests

Input: A_{pid} is the allocated block of A
 p, pid {processor info}
 c {block size}

Output: $parent$

{ run Liu + path compression on A_{pid} }
 $parent := LiuTreeCalc(A_{pid})$

{ merge pairwise }
 $step := 1$
 $done := false$

while $step < p$ **do**
 $step := step * 2$

{ communicate parent vectors }
if $notdone$ **do**
 for $i := 0$ **to** $n - 1$ **do**
 if $parent[i] \neq \infty$ **do**
 send $(i, parent[i])$ to $P_{pid \text{ div } step}$
 $done := true$

sync;

{ process received nonzeros }
if $pid \bmod step = 0$ **do**
 for all (i, p_i) received **do**
 move nonzero $a_{p_i, i}$ into column i of A_{pid}
 $parent := LiuTreeCalc(A_{pid})$

either one or no nonzero. But globally, each column of the entire matrix can contain up to p nonzeros. Merging means that one processor gets two or more blocks to play in by communicating the nonzeros from one processor to the other, as shown in Figure 3.1. Then it can continue to play the game, since there are now columns with more than one nonzero.

The prove of correctness of this algorithm directly follows from the general proof in Section 2.3. Another proof is given in [9], but this is far more complex.

Because for large p many processors have no parent candidate, they would send the root as a candidate. An obvious optimisation is to only communicate non-root elements.

Still, communication grows almost linearly in p . And with communication being the bottle-neck of parallel algorithms, good speedup cannot be expected from this method. The experimental results of Chapter 5 confirm this expectation.

3.2 A new wavefront method

The poor scalability of the communication patterns of the algorithms of the previous sections suggests looking for a completely different approach. In this section we will introduce an algorithm that processes the matrix by blocks. We will show that processing these blocks in a wavefront manner leads to a more scalable algorithm.

Assume that the rows of the matrix are distributed block-cyclically over p processors with block size β . We will then be processing $\beta \times \beta$ -blocks. Assume Processor 0 processes the upper-left block. It moves nonzeros as in the sequential Sorted Ancestors algorithm. At the end of the processing, each column has either one or no nonzeros left. If there is one, we know this is the parent of that column. This value can then be communicated to the other processors. If the column is empty, the parent, if any, will be found in a row of one of the lower blocks.

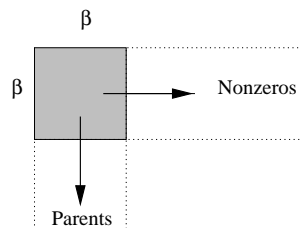


Figure 3.2: Communication patterns between blocks

The block to its right can not yet be processed at this stage, since Processor 0 could move nonzeros into that block in the current step. Besides, since the matrix is row distributed, this block is also processed by Processor 0 so processing it at this stage would lead to a load imbalance.

The block below the upper-left block will be processed by Processor 1. But not in this step, because Processor 0 will send parents at the end of processing the upper-left block. Processor 1 needs these to move nonzeros to the right that are the first of a column in its block but not the actual parent.

So after processing a block, in the next step we can process the block to the right and the block below. Starting in step 0, when Processor 0 processes the upper-left block, this leads to a wavefront through the blocks as shown in Figure 3.3.

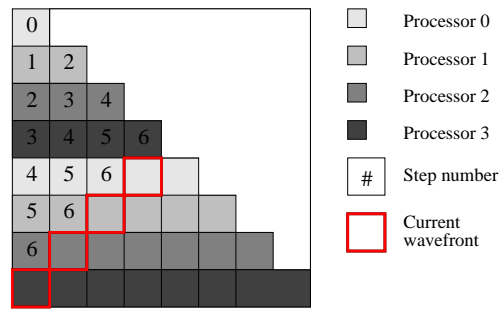


Figure 3.3: Processing the blocks in a wavefront

Algorithm 3.2 gives a pseudo-code implementation of this algorithm. The `put` command is an interprocessor communication command as defined in the BSP-model we use to model a general parallel architecture. This model is described in Appendix B. `put 5 in a at P_*` roughly means store a value 5 in variable a at all processors.

As we can see in Figure 3.4, the number of wavefronts is

$$2 \left\lceil \frac{n}{\beta} \right\rceil - 1.$$

Algorithm 3.2 Parallel wavefront method

Input: \mathcal{A} with $A_i \in \mathcal{A}$ are the locally allocated blocks of β rows
 p, pid {processor info}
 β {block size}

Output: $parent$

```

for  $step := 0$  to  $2 * \lceil \frac{n}{\beta} \rceil - 1$  do
  for  $i := 0$  to  $|\mathcal{A}| - 1$  do
    { process local block  $A_i$  }
     $start := \max\{0, (-1 * (i * p + pid) + step) * \beta\}$ 
     $end := \min\{(-1 * (i * p + pid) + step + 1) * \beta, n - 1\}$ 
    for  $j := start$  to  $end$  do
      process column  $j$  of block  $A_i$  using Sorted-Ancestors
      if smallest element  $r$  of column  $j$  exists and  $r < parent[j]$  do
        put  $r$  in  $parent[j]$  at  $P_*$ 

```

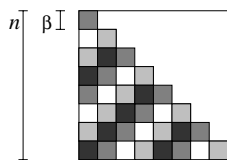


Figure 3.4: The number of wavefronts equals $2 \left\lfloor \frac{n}{\beta} \right\rfloor - 1$

3.3 Sparse header storage

In the implementations of algorithms, so far we have stored the nonzeros in a linked list per column. A pointer to each column was stored in a full array of size n . A consequence of storing the matrix in blocks of rows is that many columns in such a block might be empty and hence many of these pointers are null.

This leads to inefficient storage and, more importantly, to plenty of computation overhead. While processing a block, we run through its full header and process the nonempty columns. If most of the columns are empty, this is very inefficient. With block size β small, the total header length per processor, $\frac{\lceil n/\beta \rceil \cdot n}{p}$, can get very large and the time to traverse the full header can become dominant over the actual computations.

A solution for this problem is to store the nonzero column indices in a sparse manner. The operations that have to be performed on this set are inserting a new nonempty column, checking whether a column is nonempty and traversing the set in order of increasing column index. The most important candidates are the linked list and binary tree.

Operation	Full header	Binary Tree	Linked list	FH/LL
Inserting element	$\mathcal{O}(1)$	$\mathcal{O}(2 \log m)$	$\mathcal{O}(m)$	$\mathcal{O}(\frac{m\alpha}{n})$
Checking header	$\mathcal{O}(1)$	$\mathcal{O}(2 \log m)$	$\mathcal{O}(m)$	$\mathcal{O}(1)$
Traversing	$\mathcal{O}(n)$	$\mathcal{O}(c)$	$\mathcal{O}(m)$	$\mathcal{O}(m + \frac{n}{\alpha})$

Table 3.1: Runtime for important operations

In Table 3.1, the runtimes for the three most important operations on the header are compared for four data structures. In the table, m is the number of nonempty columns in one block of rows.

Note that these are average performance indications. The binary tree might suffer from imbalances, as shown in Figure 3.5. The average depth of a tree is $\log m$, and thus insertion costs $\mathcal{O}(\log m)$. But this assumes a random order in which the nonempty columns of the matrix are inserted. If they are inserted more or less in an increasing order, an imbalanced tree would be built with larger depth and, consequently, larger insertion time. This problem would have to be addressed by for instance a rebalancing scheme.

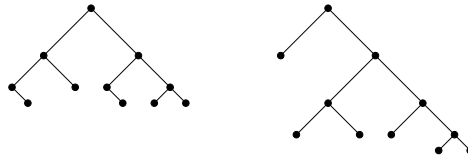


Figure 3.5: A balanced vs. an imbalanced binary tree

Although the insertion time for a straight forward linked list is rather large, this can be greatly improved by storing the elements in a bucket-sorted way. Instead of one list, a number of lists are maintained, one for each bucket.

This is done in the hybrid approach, which is a combination of a full header and a linked list. Although this is memory consuming, the best operations of both storage types can be combined. Each column is inserted only once

in the expensive sparse structure and from then on is accessed instantly in the full header. This reduces the checking time to $\mathcal{O}(1)$. The columns are processed in the order they appear in the sparse storage. Operation run-times for this hybrid storage scheme are also in Table 3.1

We have implemented a bucket-sorted linked list in combination with a full header. This implementation is visualised in Figure 3.6. The parameter defining the size of the buckets is α and in the implementation, $\alpha|\beta$ is assumed for convenience.

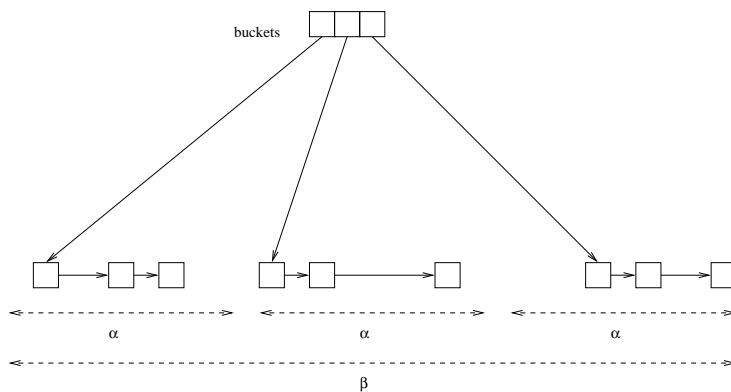


Figure 3.6: A bucket-sorted linked list

3.4 Choosing parameters

The algorithm has two parameters that influence its performance. The size of the blocks is denoted by β and the size of the buckets in which the nonempty columns are stored is α . In Section 3.4.1 we give a prediction of the runtime of those steps in the algorithm that depend on these parameters. In Section 3.4.2, optimal values for α and β are deduced using these runtime estimates. These values can not always be computed beforehand, so in Section 3.4.3 practical choices are given for α and β .

3.4.1 Time analysis

In this section, we deduce estimates for the runtime for several steps in the algorithm. These estimates are based on the BSP cost model, which is

Algorithm 3.3 Parallel wavefront method with sparse header storage

Input: \mathcal{A} with $A_i \in \mathcal{A}$ are the locally allocated blocks of β rows
 p, pid {processor info}
 β {block size}

Output: $parent$

```

for  $step := 0$  to  $2 * \lceil \frac{n}{\beta} \rceil - 1$  do
  for  $i := 0$  to  $|\mathcal{A}| - 1$  do
    { process local block  $A_i$  }
     $blocknr := step - (i * p + pid)$ 
    for all  $j \in nonzeroCols(blocknr, i)$  do
      process column  $j$  of block  $A_i$  using Sorted-Ancestors
      if smallest element  $r$  of column  $j$  exists and  $r < parent[j]$  do
        put  $r$  in  $parent[j]$  at  $P_*$ 

```

described in Appendix B. These are the steps that we will analyse:

1. Header overhead

An important fraction of the runtime is the time spent in running through the linked lists in which the nonzero columns are stored. Smaller blocks and buckets will lead to more empty linked lists, thus introducing overhead.

2. Generating new nonzero columns

Another fraction of the runtime that depends on the header storage is the time it takes to insert new column numbers. Small buckets will lead to shorter linked lists and thus to faster insertion.

3. Synchronisation time

This time is proportional to the number of wavefronts, which is of the order n/β .

4. Imbalance of workload

If one processor processes more blocks in a wavefront than another, the latter is likely to waste time waiting for the first processor to finish its computations.

Note that the total communication volume does not systematically depend on the choice of the parameters. Each parent has to be sent to all processors

some time after it is found. Therefore the communication is left out of this analysis. Still, communication load balance is influenced by the choice of β . This issue is addressed in Section 3.4.5.

We will now formulate runtime estimates, in terms of order statistics of flops. We use the symbols T_1, \dots, T_4 for the runtimes of the respective cost components.

1. The number of buckets each processor checks for nonzero columns during the algorithm is equal to the number of buckets there are in a row times the number of row blocks the processor has stored locally:

$$T_1 = \frac{n}{\alpha} \frac{n}{\beta p} = \frac{n^2}{\alpha \beta p}.$$

2. The cost T_2 is equal to the number of new columns created times the expected length of the linked list the column is inserted in, times the number of local row blocks.

Let m denote the number of nonzero columns in a block. Then the length of the linked lists is $\alpha \frac{m}{n}$ on average. If we assume that the number of new columns is far greater than the initial number, the total cost can be estimated by:

$$T_2 = m \alpha \frac{m}{n} \frac{n}{\beta p} = \alpha \frac{m^2}{\beta p}.$$

3. The algorithm performs one global synchronisation for each wavefront. T_3 is expressed in flops by the BSP-parameter l . The number of wavefronts is $2 * \left\lceil \frac{n}{\beta} \right\rceil - 1$, so an estimate for the synchronisation time is:

$$T_3 = 2 \frac{n}{\beta} l.$$

4. Since the blocks of a wavefront are distributed over the processors in a cyclic manner, any processor can have at most one block more to process than the other processors. Since the number of blocks in a wavefront changes for each wavefront, it is safe to assume that in each wavefront such a processor exists. The load imbalance this induces is proportional to the number of wavefronts and to the workload for one block. We assume this is proportional to the number of nonzeros in a $\beta \times \beta$ -block:

$$T_4 = 2 \frac{n}{\beta} \beta^2 \frac{nc}{n^2} = 2\beta c.$$

3.4.2 Optimal choices for parameters α and β

From the time estimates in the previous section, we can deduce optimal choices for α and β .

Let us first look at the parameter α . This parameter plays an important role in header operations. Note that in T_1 and T_2 , the header-related time estimates, α has different effects. Header overhead will decrease for larger α . This is because fewer buckets will be empty, thus generating less overhead. Linked list insertion will be more expensive though, since larger α leads to longer linked lists and thus to more expensive insertion.

We can deduce an optimal choice for α by minimizing

$$f(\alpha) = T_1 + T_2 = \frac{1}{\beta p} \left(\frac{n^2}{\alpha} + \alpha m^2 \right)$$

We do this by solving $f'(\alpha) = 0$. If we ignore the constant value $\frac{1}{\beta p}$, we get

$$\begin{aligned} f'(\alpha) &= 0 \\ \Leftrightarrow \frac{-n^2}{\alpha^2} + m^2 &= 0 \\ \Leftrightarrow \alpha &= \frac{n}{m} \end{aligned}$$

The variable m still denotes the number of nonzero columns. Obviously, this choice of α leads to linked lists that contain only one element on average.

The parameter β also influences the performance of the algorithm. Large β will lead to poor load balance, because it takes long for all processors to start processing blocks. Band matrices with high bandwidth are also poorly distributed in that case. Small β improves load balance but increases the number of steps the algorithm has to make and consequently the number of global synchronisations. The total communication volume is not influenced by the choice of β however.

To get a numeric estimate for β we can minimise $T_3 + T_4$. Again, this is equal to solving $g'(\beta) = 0$ with $g(\beta) = T_3 + T_4$:

$$\begin{aligned} g'(\beta) &= \frac{-2n}{\beta^2} l + 2c \\ \Leftrightarrow \beta^2 &= \frac{nl}{c} \\ \Rightarrow \beta &= \sqrt{\frac{nl}{c}} \end{aligned}$$

where c is the number of nonzeros in a column while processing.

So we now have deduced optimal values for α and β . But both expressions are dependent upon values that are not known in advance.

3.4.3 Heuristics for estimating α and β

The estimates that were deduced in the previous section are optimal choices for α and β . They can not be used in general however, since they depend on statistics that are not known in advance. In this section we will formulate heuristics for the choice of α and β that are based on the qualitative analysis in the previous section.

The optimal estimate for α showed that the optimum between header checking and header insertion was reached when all the linked lists were kept really small. The optimum was one element on average per linked list. So α depends on the fill-in factor, which is not known in advance. Experiments show that a good general value for α is 10 to 25. But fuller matrices might require a lower value.

We have seen that β influences the synchronisation time and the load balance. Large β gives low synchronisation time, but poor load balance. Our strategy will be to choose β as large as possible, so that reasonable load balance is still guaranteed.

We will first look at the number of blocks in a wavefront at each step of the computations. If the matrix A is partitioned in b row blocks, then the central wavefront has $\frac{b}{2}$ blocks. Both the wavefronts at one quarter and three quarters of the computation have $\frac{b}{4}$ blocks, as shown in Figure 3.7.

An absolute lower bound on the number of blocks is $b = 2p$. This guarantees that at some stage during the computation, all processors are processing blocks in parallel. If we choose $b = 4p$, all the processors work in parallel during half the steps. But still the load imbalance is quite significant, since during that stage each processor has either one or two blocks to process. This means that some processors are still idle half the time. A good choice for b could thus be $b = 8p$, which means $\beta = \frac{n}{8p}$. In 75% of the wavefronts, all the processors have work.

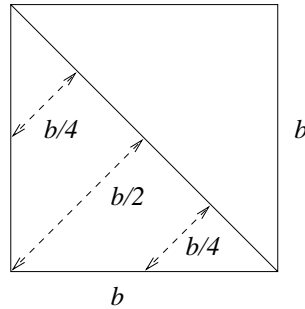


Figure 3.7: The length of the wavefronts

3.4.4 Implicit bucket sort

An aspect of the row block distribution that we have not yet discussed is an implicit bucket sort that is already performed while distributing the nonzeros of the input matrix. So on processing its blocks, a processor only has to sort approximately $\frac{c}{p}$ row indices belonging to one column. But because these are already sorted in b buckets of size β , the sorting is performed faster. Assume we are sorting with an insertion sort, which has worst case runtime $\mathcal{O}(n^2)$, but is suitable for small n . Sorting $\frac{c}{p}$ numbers costs

$$\mathcal{O}\left(\frac{c^2}{p^2}\right)$$

Sorting b buckets of $\frac{c}{pb}$ numbers costs

$$\mathcal{O}\left(b\frac{c^2}{p^2b^2}\right) = \mathcal{O}\left(\frac{c^2}{p^2b}\right)$$

This is a factor b faster. And because smaller β leads to a higher b , this is another motivation to choose smaller blocks.

This phenomenon is not accounted for in the estimates deduced in the previous sections.

3.4.5 Communication load balance

In the current implementation, the communication load does not depend on the choice of the block size. Each newly found parent is communicated to all the other processors. So the communication cost within the BSP cost

model is $np g$ and because it is constant, it is left out of the time analysis. However, only the processors that contain blocks that are beneath the current wavefront really need to receive newly found parents. Using this fact, communication in the last p wavefronts can be reduced. If we assume that β is chosen following the guidelines in Section 3.4, this reduction is not significant. Therefore this technique is not implemented.

A newly found parent is communicated using a one-phase broadcast. This means that the processor that found the parent sends it to all the other processors. This leads to $p - 1$ consecutive communications by one processor. Within the BSP cost model, this is called a $(p - 1)$ -relation.

Of course, it is more expensive to send $p - 1$ numbers than to receive one. So if the parents that are found within a superstep are not evenly spread over the processors, we have a communication load balance problem. For larger β , this is likely to occur. Since the parent is the first nonzero in a column beneath the diagonal, it is likely to be found close to the diagonal. It is safe to say that the blocks within a wavefront that are closer to the diagonal will contain more parents and will induce more communications.

Obviously, large β leads to this kind of imbalance. Communication load balance will be better if more than one block is assigned to each processor in as many wavefronts as possible. Then, each processor has a block that is close to the diagonal and some blocks that are further away from it. So this also sets an upper bound on the size of β that can be chosen.

3.5 Experimental results

Because the performance of this algorithm is very closely related to that of the actual factorisation algorithms that are discussed in the next chapter, the combined experimental results of both the tree computation and the factorisation are discussed in a separate chapter, Chapter 5.

Chapter 4

Symbolic factorisation using the elimination tree

In this chapter, we will study the symbolic Cholesky factorisation. In Chapter 1, we have seen that the Fast Symbolic Cholesky Factorisation algorithm computes the symbolic factor and also the elimination tree as a by-product. We will now assume that we know the elimination tree in advance and use it to speed up the calculation of the symbolic Cholesky factor L .

4.1 An alternative storage scheme for the elimination tree

The result of the tree search algorithms is a vector *parent* of length n , such that the parent of column i is $parent(i)$. Thus, the parent of an arbitrary column can be determined in $\mathcal{O}(1)$ time. The children of an arbitrary column though can only be determined in $\mathcal{O}(n)$ time. In this section, we propose an alternative storage scheme of the elimination tree that enables retrieving both parent and children in $\mathcal{O}(1)$ time.

Let us look at the vector *parent* differently and write it as a $2 \times n$ matrix F such that $F(0, i) = i$ and $F(1, i) = parent(i)$.

Example 4.1.1

$$parent = (3 \ 2 \ 3 \ 4 \ 5) \quad \Rightarrow \quad F = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Now the columns of F define the edges in the elimination tree associated with F . Permuting the columns does not affect the information stored in F . Therefore we can introduce a child-aware representation of the elimination tree as follows.

Definition 4.1.2 (Child-aware representation) \hat{F} is the matrix that is obtained from F by permuting the columns such that:

1. $\hat{F}(1, i) \geq \hat{F}(1, j)$ if $i > j$
2. $\hat{F}(0, i) > \hat{F}(0, j)$ if $i > j \wedge \hat{F}(1, i) = \hat{F}(1, j)$

In other words, columns are sorted by non-decreasing parent and columns with the same parent are sorted by increasing original index. By maintaining a vector of length n with for every column i a pointer to the columns of matrix \hat{F} with its first child, we can retrieve all the children of column i in $\mathcal{O}(c_i)$ time, where c_i is the number of children of column i .

Example 4.1.3

$$F = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 2 & 3 & 4 & 5 \end{pmatrix} \Rightarrow \hat{F} = \begin{pmatrix} 1 & 0 & 2 & 3 & 4 \\ 2 & 3 & 3 & 4 & 5 \end{pmatrix}$$

Theorem 4.1.4 (Creating \hat{F}) The child-aware representation \hat{F} can be calculated from F in $\mathcal{O}(n)$ time.

Proof: We calculate \hat{F} as follows. First, we run through the second row of F and determine the number of children for each parent. Now we can allocate exactly the required number of columns for each parent in the matrix \hat{F} . Thus, we can run through the columns of F again and put them in the appropriate place in \hat{F} . This guarantees that the second row of \hat{F} is sorted. Because the first row of F is sorted and we add the columns to \hat{F} in increasing order, the children with the same parent are guaranteed to be sorted too. Since this calculation scheme consists only of loops of length n , its computation time will be $\mathcal{O}(n)$. \square

4.2 A left-looking algorithm

We can use this property of the new representation to speed up the sequential Symbolic Factorisation Algorithm 1.3. We will show that adding all

children of a parent to this parent at the same time has an advantage over the old way. It allows us to factorise symbolically in $\mathcal{O}(nz(L))$ time.

Suppose we add 5 children C_1, \dots, C_5 to column P which has p elements. Suppose the 5 child columns have c_1, \dots, c_5 nonzeros, respectively, and suppose all nonzeros occur in one column only. We analyse the costs of adding the columns separately, possibly with other operations in between.

Adding C_1 to P costs $\mathcal{O}(c_1 + p)$. Adding C_2 to P now costs $\mathcal{O}(c_2 + p')$, where p' is the new number of nonzeros of P after adding C_1 . So adding C_2 actually costs $\mathcal{O}(c_2 + c_1 + p)$. After adding all columns, the total costs are $\mathcal{O}(c_5 + \dots + 4c_2 + 5c_1 + 5p)$.

If we use the expanding addition technique and add all the columns C_1, \dots, C_5 at the same time, the total costs would be $\mathcal{O}(c_5 + \dots + c_1 + p)$.

Though these costs are upper bounds, they do suggest a reduction in computation costs.

The problem is in what order to add the columns so that all children are added at the same time and no column is added before its children are added to it.

Theorem 4.2.1 *By adding columns to their parent in the order the columns occur in the first row of the matrix \hat{F} ,*

1. *all columns with the same parent are added consecutively, and*
2. *all children are added before their parent is added.*

Proof:(1) This is true, for the second row of \hat{F} is sorted. So all the columns of F having the same parent form a consecutive column block in \hat{F} .

(2) Take two arbitrary columns that represent parent and child. Let the child be column i of \hat{F} and the parent be column j of \hat{F} . Because they are parent and child, we have $\hat{F}(1, i) = \hat{F}(0, j)$. By Corollary 1.4.5, $\hat{F}(0, j) < \hat{F}(1, j)$. Hence,

$$\hat{F}(1, i) = \hat{F}(0, j) < \hat{F}(1, j).$$

And therefore, by Theorem 4.1.2 (1), $i < j$. So the child appears in \hat{F} before the parent does and is therefore added first. \square

This obviously leads to an algorithm that adds columns in the order proposed in Theorem 4.2.1. We will call this algorithm the left looking algorithm. Algorithm 4.1 implements efficient addition of multiple columns to a single column.

Algorithm 4.1 Adding columns to the same parent efficiently

Input: C_i with $i \in I$, index sets of columns to be added to column p ,

P is index set of nonzeros in column p

Output: P

{ expanding column P to full array }

for all $j \in P$ **do**

$fullarray[j] := p$

{ adding columns $i \in I$ }

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

for all $j \in C_i$ **do**

if $fullarray[j] \neq p$ **do**

$P := P \cup \{j\}$

$fullarray[j] := p$

Note that in Algorithm 4.1, the array *fullarray* has to be initialised, for instance to -1 , before it is first used. It need not be reset during the algorithm though, because different parents have different p 's. Another approach is to use a boolean array as *fullarray*, but this would require resetting the array after each group of columns is added to their parent.

4.3 Calculation based on row structure

In this section, we introduce another algorithm to compute the symbolic factorisation. We will call it the row-structure (RS) algorithm. It will later turn out to perform even better than the algorithm introduced before. There is a strong relation between the structure of a row k in L and the elimination tree T as we will show in the following theorems. But first we need a definition.

Definition 4.3.1 *Given a sparse matrix L , we denote the (lower triangular) structure of row k by:*

$$Lrow(k) = \{j : l_{kj} \neq 0 \wedge 0 \leq j < k\}$$

Theorem 4.3.2 (Schreiber [6]) *If $j \in Lrow(k)$, then there is a path from node j to node k in T and for every node $t \neq k$ on this path, $t \in Lrow(k)$.*

Theorem 4.3.3 (Liu [4]) *If $j \in Lrow(k)$, then there exists a node $d \leq k$ such that $a_{kd} \neq 0$ in A and d is a descendant of j in T .*

These two theorems can be used to prove the next theorem, which is the basis of our row-structure algorithm.

Theorem 4.3.4 (Calculate $Lrow(k)$ from $Arow(k)$ and T) *Row k of the symbolic factor L can now be computed as follows. Run through $Arow(k)$ and for every nonzero a_{jk} , start in the elimination tree T at the corresponding node j and run through T to node k . Every node on this path, except node k , corresponds to a nonzero in $Lrow(k)$. All nonzeros of $Lrow(k)$ are generated this way.*

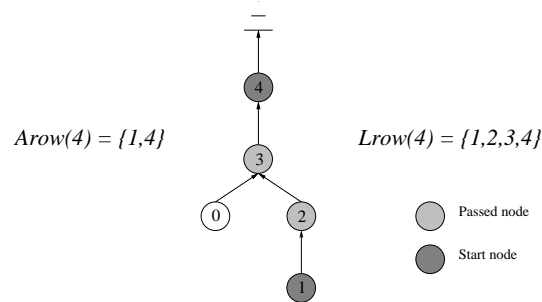
Proof: All the nodes passed are in $Lrow(k)$, because $j \in Arow(k) \subset Lrow(k)$ and because of Theorem 4.3.2.

To show that all the elements of $Lrow(k)$ are computed in this way, we need Theorem 4.3.3. Pick an arbitrary element j of $Lrow(k)$. Theorem 4.3.3 says there is an element $d \leq j$ in $Arow(k)$. The algorithm will take this d as a starting point at some stage and since Theorem 4.3.3 also says that d is a descendant of j in T , node j will be passed. \square

Theorem 4.3.4 introduces a method to compute the structure of L . This method is illustrated in Figure 4.1 and implemented in Algorithm 4.2.

Algorithm 4.2 is inefficient because it computes most of the elements of $Lrow(k)$ several times. But the algorithm can be implemented much more efficiently by storing the passed nodes in a sparse data structure and using a temporary full array.

Using this array, we can test at each step whether the algorithm has been at that node before during step k and if so, abort the walk through the tree at that point. This technique guarantees that each node of $Lrow(k)$ is touched

Figure 4.1: Calculating row 4 of L from row 4 of A and T **Algorithm 4.2** Symbolic factorisation algorithm based on row-structure*Input:* The nonzero structure of A , the elimination tree T *Output:* The nonzero structure of L .

```

for  $k := 0$  to  $n - 1$  do
  for all  $j \in Arow(k)$  do
    while  $j \neq k$  do
       $Lrow(k) := Lrow(k) \cup \{j\}$ 
       $j := parent[j]$ 

```

only once. This idea is implemented in Algorithm 4.3

Algorithm 4.3 has runtime complexity

$$\mathcal{O}(nz(L) + nz(A) + n).$$

Here, $nz(L)$ accounts for all the nodes that are passed once, $nz(A)$ for terminating each run either at the root k of the subtree or at a node already passed, and it costs n to initialise the full array. This is the same complexity as the symbolic factorisation, but simpler basic operations make a more efficient implementation possible.

4.4 Parallel symbolic factorisation

In this section we will introduce a parallel algorithm for symbolic factorisation. The most important choice is that of the data distribution. The parallel tree computation algorithm introduced in Section 3.2 distributes

Algorithm 4.3 Improved implementation of the row-structure algorithm

Input: The nonzero structure of A , the elimination tree T

Output: The nonzero structure of L .

```

for  $k := 0$  to  $n - 1$  do
  for all  $j \in Arow(k)$  do
    while  $j \neq k \wedge fullarray[j] \neq k$  do
       $Lrow(k) := Lrow(k) \cup \{j\}$ 
       $fullarray[j] := k$ 
       $j := parent[j]$ 

```

the matrix A over the processors by rows. It seems a logical choice to use the same distribution for the symbolic factorisation.

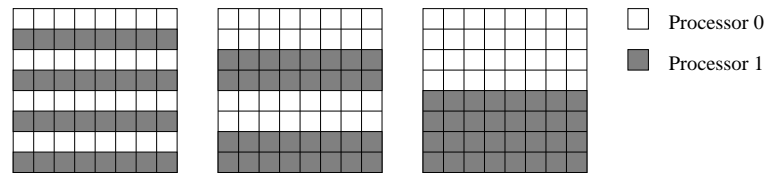


Figure 4.2: Cyclic, block-cyclic and block distribution of rows

This turns out to be a good choice. Both the left-looking algorithm and the row-structure algorithm can be trivially parallelised given this data distribution and provided a complete copy of the elimination tree is available.

4.4.1 Parallel left-looking algorithm

Given that every processor has a number of rows of the matrix A allocated to it and the elimination tree T , a processor can compute the corresponding rows of L locally without communication.

The algorithm is actually equal to the sequential algorithm. Add all the columns to their parents and optimise this by doing it family-wise. Because entire rows are allocated to a processor, adding columns is a local operation. The row block distribution does however result in locally empty column parts and these have to be treated efficiently to avoid extra overhead.

4.4.2 Parallel row structure algorithm

It is trivial to show that this algorithm can be parallelised without introducing communication. The sequential algorithm was defined by showing that a row of L could be computed from a row of A and the elimination tree T . Since rows are local to a processor and T is available at every processor, obviously calculating rows of L is a local operation.

The only part of the algorithm that does not scale is initialising of the full array. So the runtime will be

$$\mathcal{O}\left(\frac{nz(L) + nz(A)}{p} + n\right)$$

4.4.3 Load balance

This parallelisation will work for all row distributions, whether block, block-cyclic or cyclic distributions. The only point of attention should be the load balance.

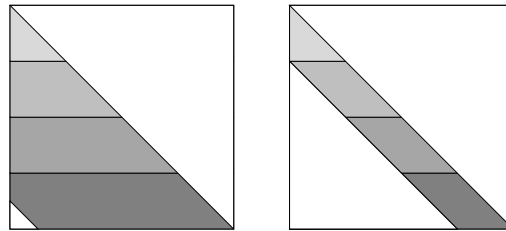


Figure 4.3: Load balance problems with high-bandwidth matrices

A block distribution has some disadvantages. If we are factoring a high bandwidth matrix, the nonzeros are unevenly distributed over the processors. The rows at the first processor are much shorter than the rows on the last processor. Both the block-cyclic and cyclic distributions improve load balance in these cases.

4.5 Sequential results

In this section we compare three implementations of the sequential symbolic factorisation code. We compare the Fast Symbolic Cholesky (FSC)

code from Chapter 1, the left-looking code from Section 4.2 and the row-structure code from Section 4.3. We have run these codes on one node of a Cray T3E, so that we can compare these values to the parallel results in the next section. The factorisation times are in Table 4.1.

Matrix	FSC	left-looking	RS
1138BUS	25.69	24.20	11.49
bcsstk18	1523.09	1479.92	1086.26
bcsstk26	53.34	50.52	37.94
s2rmt3m1	404.51	381.87	271.44

Table 4.1: Factorisation runtime on a Cray T3E in ms

Clearly, both the left-looking and the RS method are faster than the FSC code. But we have to keep in mind that to run these two codes, the elimination tree has to be known in advance. In Table 4.2, the total runtime is given, including the calculation of the elimination tree by Liu's method with path compression.

Matrix	FSC	left-looking	RS
1138BUS	25.69	25.62	13.61
bcsstk18	1523.09	1515.99	1116.39
bcsstk26	53.34	55.58	42.28
s2rmt3m1	404.51	413.56	300.80

Table 4.2: Total runtime on a Cray T3E in ms

The RS method is faster than the FSC code, even if you include the runtime for calculating the elimination tree.

In conclusion, it is worthwhile to split the symbolic factorisation in a tree generation phase and a factorisation phase.

Chapter 5

Parallel results

In this chapter, we present results of parallel implementations of the algorithms that were introduced in the previous chapters. We look at the two methods for computing elimination trees and at the actual factorisation phase. We also look at the combined runtime of these two phases and at the effect of different choices of α and β . These effects are compared to the expected behaviour as formulated in Section 3.4.

5.1 Computing trees

5.1.1 Gilbert & Zmijewski's method

Our implementation of Gilbert & Zmijewski's method distributes the matrix in a row block distribution. It uses Liu's algorithm with path compression to calculate a local tree.

In Section 3.1, the trees were merged pairwise. Our implementation enables a more general merging scheme. Experiments have shown that merging four trees per merging step is more efficient. This way, the number of synchronisations is reduced, as can be seen in Figure 5.1.

Note that at the end of Gilbert & Zmijewski's algorithm, only one processor has the full elimination tree stored locally. Since the elimination tree has to be known by all processors in a later operation, a broadcast of the tree is necessary. This is included in the runtimes in Table 5.1.

In Figure 5.2, the speedup for Gilbert & Zmijewski's method is plotted for

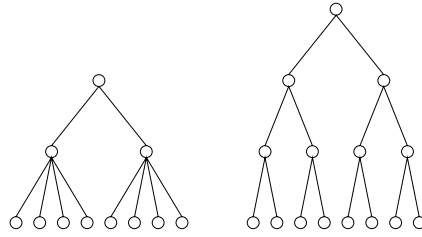


Figure 5.1: Two merging strategies

Matrix	$p = 1$	$p = 2$	$p = 4$	$p = 8$
900BUS	14	44	92	204
1138BUS	7.2	40	88	212
BCSSTK16	388	448	404	524
BCSSTK17	576	672	672	776
BCSSTK18	204	352	356	428
BCSSTK26	40	84	134	248
S2RMT3M2	292	344	364	452
S3RMT3M3	272	324	356	456

Table 5.1: Runtimes for Zmijewski's method on the Cray T3E in ms.

several matrices. Obviously, performance is poor. In general, adding processors increases the runtime, instead of decreasing it. Note that this is the parallel code running on one processor. The sequential code is even several times faster.

5.1.2 The wavefront method

In this section we present experimental results for the wavefront method introduced in Section 3.2. In Table 5.2, runtimes are given for several matrices.

In Figure 5.2, the speedup is visualised. Clearly, there is no considerable speedup for any of the tested matrices. Still, the speedup results are much better than the results for Gilbert & Zmijewski's method.

Matrix	$p = 1$	$p = 2$	$p = 4$	$p = 8$
900BUS	28	32	33	35
1138BUS	27	27	30	33
BCSSTK16	500	444	452	456
BCSSTK17	780	704	700	716
BCSSTK18	288	308	308	328
BCSSTK26	64	72	72	76
S2RMT3M2	372	340	344	356
S3RMT3M3	416	412	404	400

Table 5.2: Runtimes for the wavefront method on the Cray T3E in ms.

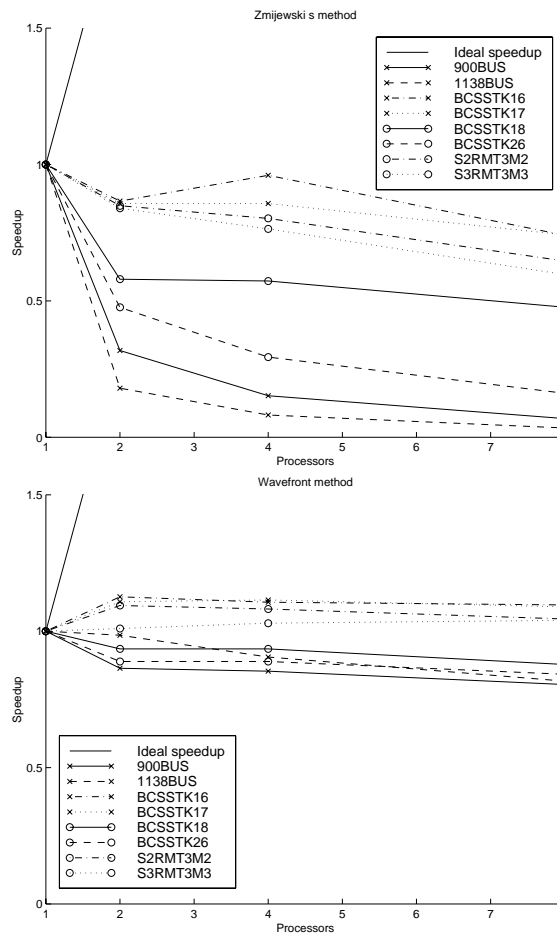


Figure 5.2: Speedup for Zmijewski and the wavefront method

5.2 Comparing Zmijewski and the wavefront method

From the data in the previous section, we have seen that the wavefront method shows better speedup characteristics. Zmijewski's code is faster on a small numbers of processors however. For many matrices this is just true for the one processor case, which is not very interesting. For some it is also faster on two or even four processors. In Figure 5.3 the runtimes of both methods are compared for certain large matrices.

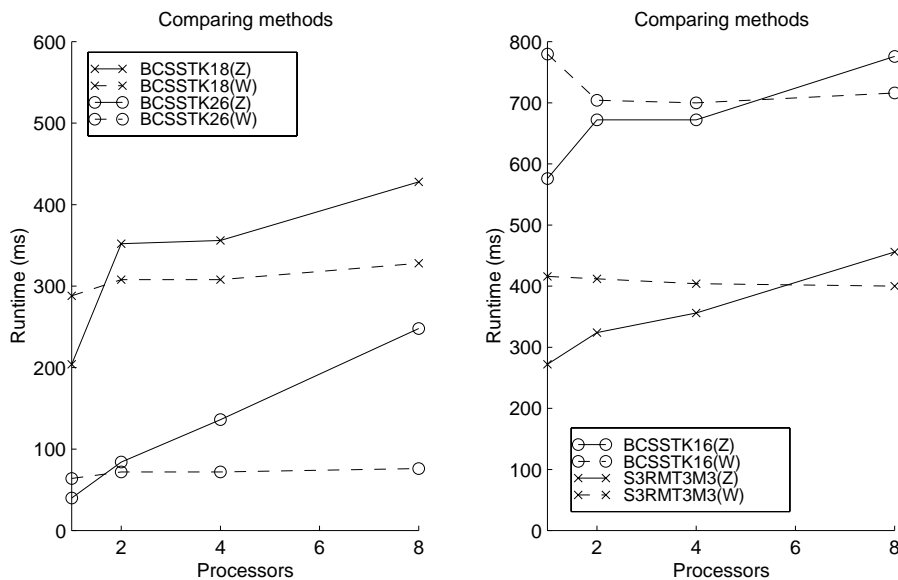


Figure 5.3: Comparing runtimes for Zmijewski and the wavefront method

Also note that in the algorithms we discuss all processors know the parent at the end. This is necessary to compute the symbolic Cholesky factor. If only one processor needs to know the elimination tree, Zmijewski's algorithm has the advantage that it can skip the communications at the end. This is not a likely situation however, since the elimination tree is hardly ever the final result that is required.

5.3 Symbolic factorisation

In this section we use the factorisation method from Section 4.4.2 in combination with the parallel wavefront method to see whether we can speed up the complete factorisation.

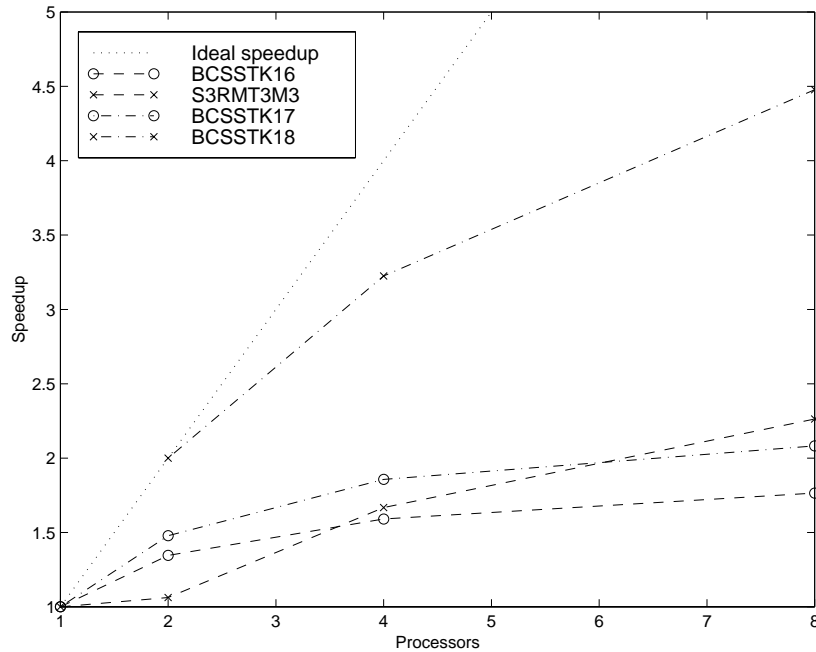


Figure 5.4: Speedup of complete symbolic Cholesky factorisation

In Figure 5.4, the speedup for the complete symbolic Cholesky factorisation is plotted. Clearly, there is modest speedup for all the matrices. The algorithm does not scale well and a maximum is reached fast. This can be explained by looking at the speedup behaviour for both the parallel tree computation algorithm and the actual symbolic factorisation algorithm. The factorisation algorithm scales very well and exhibits linear speedup. However, the tree computation shows no speedup and therefore limits the speedup for the combined algorithm.

In Figure 5.5, the total factorisation runtime is split into the runtime for the tree computation and the runtime for the actual factorisation. The total runtime is plotted with a solid line, the tree computation with a dashed

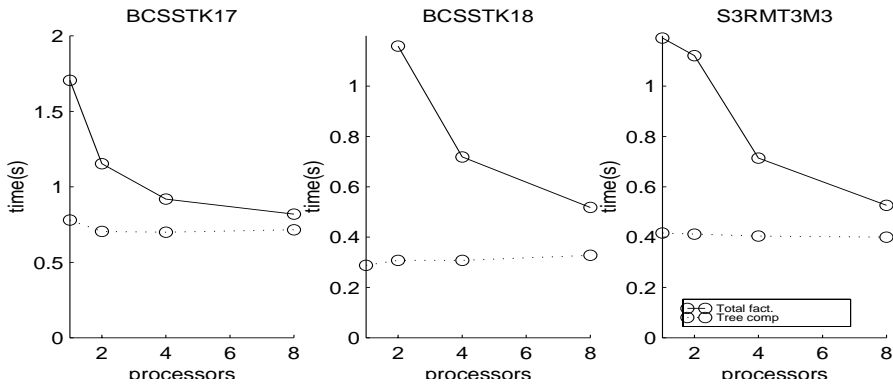


Figure 5.5: Cumulative plot of tree computation and factorisation

line. Clearly, the total runtime approaches the tree computation runtime fast.

5.4 Effects of parameters

In Section 3.4, we have given a theoretical analysis of the choice of the parameters α and β . In this section, we give some experimental results on this matter.

In Table 5.3, the parameter values are given that were used to obtain the results from the previous two sections.

Matrix	Optimal β (α)			
	$p = 1$	$p = 2$	$p = 4$	$p = 8$
BUS900	900(25)	200(20)	200(10)	100(10)
BUS1138	1200(25)	500(10)	200(10)	100(10)
BCSSTK17	1000(100)	100(25)	100(25)	80(10)
BCSSTK18	1000(100)	500(10)	200(25)	100(10)
BCSSTK26	1000(25)	200(25)	80(10)	50(10)
S2RMT3M2	100(20)	100(10)	80(10)	80(10)
S3RMT3M3	1000(25)	400(10)	300(10)	300(10)

Table 5.3: Parameter values for data in Section 5.1.2

Note that the optimal values were found using a rather large step size for the parameters. In Figure 5.6, the total factorisation time is plotted for a detailed set of parameter values for the matrix BCSSTK17.

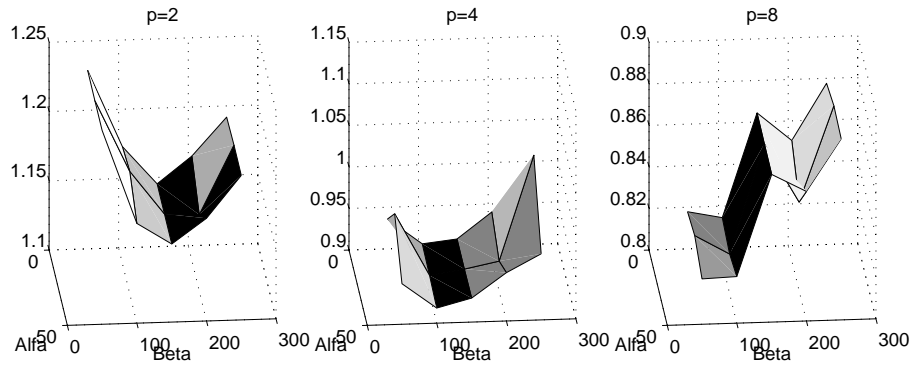


Figure 5.6: Factorisation time of BCSSTK17 for different α and β

Clearly, both the optimal α and β decrease as p increases. For larger problems, the optimal β decreases linearly in p , which was predicted in Section 3.4.

Conclusions and future work

Conclusions

In this thesis we have tried to improve the speed of symbolic Cholesky factorisation of sparse matrices by using elimination trees.

Liu's algorithm with path compression proved to be the fastest algorithm to sequentially compute the elimination tree. The new Sorted-Ancestors algorithm that is introduced in this paper requires the same number of operations but is a factor 3 to 5 slower in the sequential case, due to more complex operations.

The Sorted ancestors algorithm is however more suitable for parallelisation. The introduced parallel wavefront algorithm based on it does not exhibit much speedup compared to the sequential algorithm, but it does not show slowdown either. Its computation time is more or less independent of the number of processors. It has the advantage of being a distributed algorithm, so it is scalable with respect to memory.

The symbolic factorisation algorithm introduced in Chapter 4.3 computes the symbolic factor based on the elimination tree very efficiently. Combined with Liu's algorithm with path compression, it computes the symbolic Cholesky factor faster than the conventional algorithm. Also, the algorithm can be parallelised without introducing communication and shows linear speedup. Together with the parallel Sorted Ancestors algorithm, this results in a parallel symbolic Cholesky factorisation algorithm that gives modest speedup.

The overall conclusion is that in both the sequential and the parallel case, the symbolic Cholesky factorisation of sparse matrices can be speeded up by calculating the elimination tree in advance.

Future work

Further research should focus on improving the speedup of the parallel elimination tree computation, since this part of the algorithm limits the speedup of the combined symbolic Cholesky factorisation. We give two suggestions for further research.

In this thesis, all processors have full knowledge of the elimination tree after computing it in parallel. However, not all processors use all that information for performing the symbolic factorisation. An improvement of the proposed algorithm could be to distribute the elimination tree over the processors on a need-to-know basis. A reduction of the communication volume could be obtained and performance could hence be improved.

Another improvement could be reached by improving the move of the first nonzero of each column in a row block. These are now moved based on parent information, which leads to suboptimal moves. Better moves would require more information on larger ancestors and thus require more communication. But possibly an improvement can be reached.

Appendix A

Sorting methods

In several algorithms, we use methods to fully or partially sort arrays of integers. There are several techniques to do this. To choose the right method, one needs to know in which circumstances they perform well.

A.1 Insertion sort

A straightforward sorting method is insertion sort. One by one, the elements of the source array are inserted in the destination array, by starting at the end of the array and moving all elements greater than the element to be inserted to the right to make space for this element.

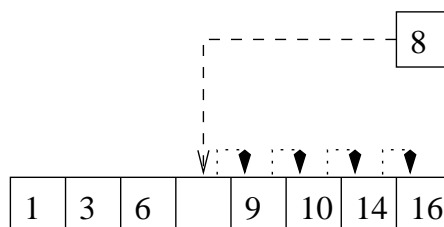


Figure A.1: One step in the insertion sort of an array

This technique has a runtime of $\mathcal{O}(n^2)$ for sorting n elements. It can be implemented to sort in place and will show to be very fast on small arrays. In fact, the sequential implementation of full sorted tree computation showed to be fastest using this technique.

A.2 Quicksort

An alternative that is used frequently is quicksort. This also has a worst-case runtime of $\mathcal{O}(n^2)$, but on average it takes $\mathcal{O}(n \log(n))$.

Quicksort sorts an array $A[p \dots r]$ by dividing it into two sub arrays $A[p \dots q]$ and $A[q \dots r]$, such that each element of the first sub array is smaller or equal to the pivot element q and every element from the second array is larger than q . It then sorts these two arrays by a recursive call to quicksort on these two arrays.

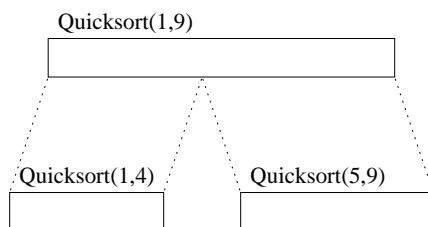


Figure A.2: Quicksort recursively sorts sub arrays

We will not discuss quicksort in detail. For more information on the choice of the pivot element q and quicksort's implementation of the partitioning of the arrays, see [2].

A.3 Comparison of insertion sort and quicksort

To compare the two sorting methods, we have sorted arrays of random integers. We sorted arrays of length 4 to 200 with step size 2. For each length, 400 arrays were generated and sorted and the mean runtime was determined. The computations were performed on one node of a Cray T3E. The results are in Figure A.3.

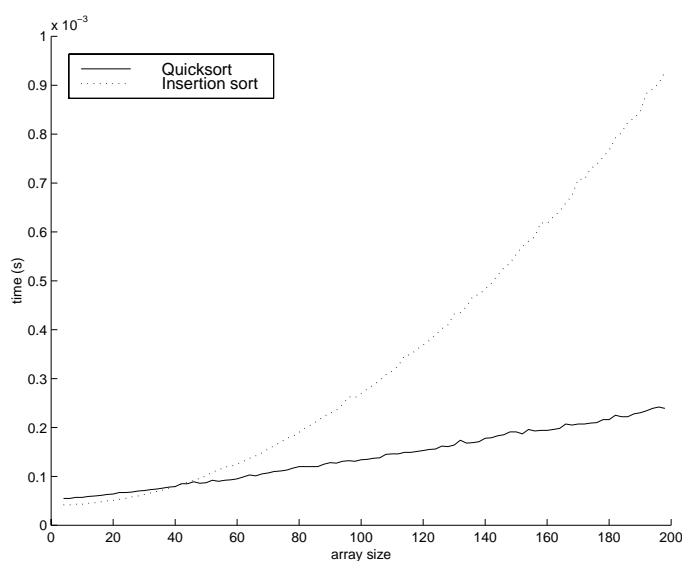


Figure A.3: Comparing Quicksort and Insertion sort

Clearly, quicksort is in general the best sorting technique. However, for the elimination tree construction algorithm in Section 2.4.2, insertion sort was used: because the matrices we used were sparse, even in quite large matrices the number of nonzeros in a column was never big. And as we can see in Figure A.3, sorting of up to about 40 elements is still done faster with insertion sort.

A.4 Partial sorting

The two algorithms we have seen so far sort every array of numbers, regardless of the range they are in or whether they are integers or doubles. They are based only on the possible pairwise comparison of any two numbers of the set. There are sorting algorithms that benefit from extra knowledge about the set to sort in linear time. We use a combination of bucket sort and counting sort to partially sort an array of integers in linear time.

We use the property of the array that all the integers in the array are in the $[k \dots n]$ integer interval. We divide this interval in equal subintervals called buckets.

Now we run through the source array to count for every bucket how many elements it will contain. Based on this information, space is allocated for each bucket in the destination array and the elements are placed there.

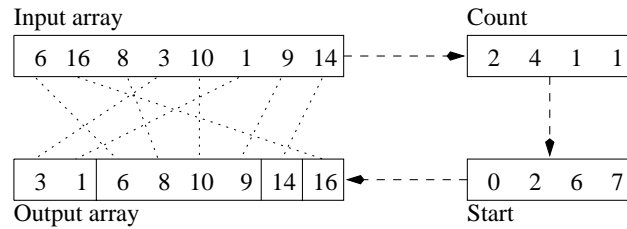


Figure A.4: Partial sort of an array using four buckets of size five.

For a full sort each bucket could now be sorted in place using for instance an insertion sort.

Appendix B

The BSP model

Nowadays, there is a wide spectrum of different parallel computers on the market and they all have their own possibilities and techniques. To avoid the use of all sorts of platform and hardware specific mechanisms, we have developed our code within the Bulk Synchronous Parallel (BSP) model. This model was proposed in 1990 by Valiant in [8]. It defines a BSP computer, a BSP algorithm and a cost model.

B.1 A BSP computer

The BSP model models a generic distributed memory parallel computer. It consists of a set of processors, each with an amount of local memory. These processors are connected through a communication network, of which the model assumes very little.

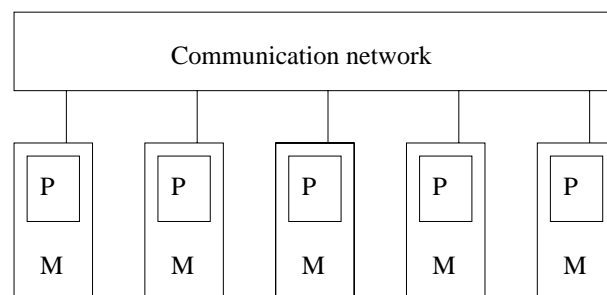


Figure B.1: A BSP computer

Through this network, a computer can access memory that is not local. Of course, this costs more time than accessing its own local memory. The model assumes that this time is equal for all processors, whereas this might actually not be the case, since this depends on the network topology.

In an implementation of the BSP-model, called BSPlib, there are two ways to access remote memory. The first is Direct Remote Memory Access (DRMA). This method is used when the sender knows what part of memory he will be writing to. All the processors register parts of their local memory to which the other processors can write and read from. In a communication superstep, all processors put data in remote memory and after a synchronization, they are directly accessible.

The other mode is called Bulk Synchronous Message Passing (BSMP). This technique is used when the amount of transmitted information is not known in advance or differs for different processors. In a communication superstep, all processors can send data packets to various processors and after a synchronization, all the processors can read the packets sent to them from a buffer.

B.2 A BSP algorithm

The model also defines a structure for the algorithms implemented on the BSP computer. An algorithm is said to consist of a series of supersteps, where after each superstep all the processors perform a global synchronization. These supersteps are alternatively of the computation type and the communication type. During computation supersteps, processors perform actions on local data. These supersteps are followed by communication supersteps, in which the processors send and receive data from other processors. The model does not provide pairwise synchronization mechanisms. This is why the BSP computer is called bulk synchronous. Each processor runs through exactly the same number of supersteps. The superstep concept of a BSP-algorithm is illustrated in Figure B.2.

B.3 The cost model

The BSP model also provides us with a cost model that predicts the running time of the algorithm. Basically, the costs of all supersteps are computed and added. The total cost is a number of flops, so the running time can be

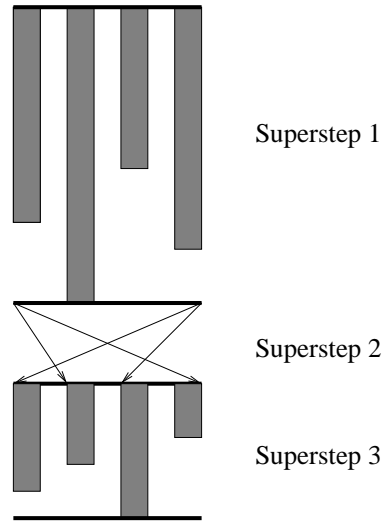


Figure B.2: A BSP algorithm

computed by dividing by the flop rate s of the computer.

The cost of a computation superstep is determined by the maximum number of flops performed by a processor in that superstep. To calculate the costs, we add the cost of a global synchronisation, denoted by l . Assuming w flops are performed by the processor with the maximum amount of work,

$$T_{comp} = w + l.$$

The cost of a communication superstep is computed by examining the amount of sent and received data. We denote the maximum number of words sent by any processor by h_s and the number of words received by h_r . Then

$$h = \max\{h_s, h_r\}.$$

The cost of such a communication superstep is

$$T_{comm} = hg + l,$$

where g is the number of flops one word sent costs and l is the cost of a global synchronisation.

Given this model, we can predict the running time of an algorithm on a specific platform if we know the following parameters for this platform: p , l , g and s .

B.4 Benchmarks of the Cray T3E and IBM SP

An implementation of the BSP model is available for a variety of platforms, ranging from massively parallel supercomputers to clusters of workstations. See [3] for more details on BSPlib, the BSP library. In our experiments we made use of two parallel computers. We experimented on the 76-node IBM SP at Peca/SARA in Amsterdam and the 80-node Cray T3E at HP α C/TU Delft in Delft. In this appendix we give benchmark results for the BSP variables of both machines.

Table B.1 contains some benchmarks on the IBM-SP and Cray T3E. The figures are computed with the `bsp_probe` utility. They are updates for the results given in [7], which were also determined with this utility.

Note that g and l are expressed in flops and that flop rates have also changed for both machines.

Machine	s (Mflops/s)	p	g_{local}	g_{global}	l
SP2(Switch)	59	2	4.88	11.00	5636
Cray T3E	190	2	0.86	2.49	428
		4	0.82	1.52	534
		8	0.81	1.48	655
		16	1.02	1.51	966
		32	1.77	1.74	1199

Table B.1: Benchmark results of `bsp_probe`

In Table B.2, we give some alternative benchmarks for both machines. They are measured with the `bench` utility from `BSPPACK_EDU`.

Machine	s (Mflops/s)	p	g	l
SP2 (Switch)	80	2	73.6	19726.7
		4	71.2	30989.9
		8	112.3	56340.1
Cray T3E	38	2	35.8	632.4
		4	38.7	829.1
		8	41.0	1373.6
		16	39.3	2200.4

Table B.2: Benchmark results of `bench`

Appendix C

Test matrices

We have used a set of sparse test matrices to test the algorithms developed in this thesis. Some of these are from the Harwell-Boeing set of matrices. Others are from separate sets. They are all available at the Matrix Market at <http://math.nist.gov/MatrixMarket/>

Table C.1 gives some statistics on the used test matrices. bw denotes the bandwidth of the matrix, c denotes the number of nonzeros per column.

Matrix name	n	$nz(A)$	c	bw
900BUS	900	7744	8.6	32
1138BUS	1138	4054	3.6	1031
BCSPWR10	5300	21842	4.1	5190
BCSSTK16	4884	290378	59	141
BCSSTK17	10974	428650	39	522
BCSSTK18	11948	149090	12	1244
BCSSTK26	1922	30336	16	234
BLCKHOLE	2132	14872	7	1806
LSHP3466	3466	23896	7	3435
S2RMT3M1	5489	217681	40	192
S3RMT3M3	5357	207123	39	5303

Table C.1: Properties of the used test matrices

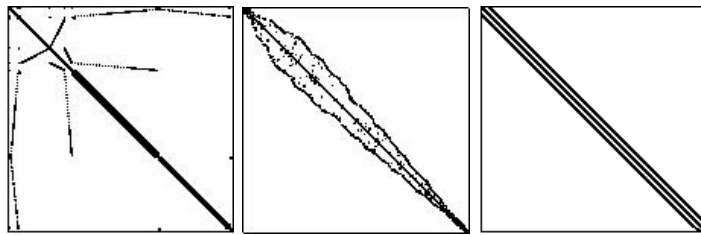


Figure C.1: Test matrices with high, medium and low bandwidth

Bibliography

- [1] ROB H. BISSELING, *Parallel Scientific Computation with BSP*, to appear.
- [2] T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge Massachusetts, 1989.
- [3] JONATHAN M.D. HILL, BILL MCCOLL, DAN C. STEFANESCU, MARK W. GOUDREAU, KEVIN LANG, SATISH B. RAO, TORSTEN SUEL, THANASIS TSANTILAS, ROB H. BISSELING, *BSPlib: The BSP Programming Library*, *Parallel Computing* 24 (1998) 1947-1980.
- [4] JOSEPH W. H. LIU, *A Compact Row Storage Scheme for Cholesky Factors Using Elimination Trees*, *ACM Trans. on Math. Software* 12(2), pp. 127-148, 1986.
- [5] WILLIAM H. PRESS, SAUL A. TEUKOLSKY, WILLIAM T. VETTERLING, BRIAN P. FLANNERY, *Numerical Recipes in C, The Art of Scientific Computing, Second Edition*, Cambridge University Press, 1992.
- [6] R. SCHREIBER, *A new implementation of sparse Gaussian elimination*, *ACM Trans. on Math. Software* 8, pp. 256-276, 1982.
- [7] D.B. SKILLICORN, J.M.D. HILL AND W.F. MCCOLL, *Questions and Answers about BSP*, *Scientific Programming* 6(3), pp. 249-274, 1997.
- [8] L. G. VALIANT, *A Bridging Model for Parallel Computation*, *Communications of the ACM*, 33(8), pp. 103-111, 1990.
- [9] EARL ZMIJEWSKI AND JOHN R. GILBERT, *A parallel algorithm for sparse Cholesky factorization on a multiprocessor*, *Parallel Computing* 7, pp. 199-210, 1988.

List of algorithms

1.1	A dense numerical Cholesky factorisation algorithm	2
1.2	A basic symbolic Cholesky factorisation algorithm	4
1.3	The Fast Symbolic Cholesky factorisation algorithm	6
2.1	Tree computation by symbolic factorisation	13
2.2	Sorted-ancestors	14
2.3	Liu's algorithm	16
2.4	Liu's algorithm with path compression	17
3.1	Merging partial forests	26
3.2	Parallel wavefront method	29
3.3	Parallel wavefront method with sparse header storage	32
4.1	Adding columns to the same parent efficiently	42
4.2	Symbolic factorisation algorithm based on row-structure	44
4.3	Improved implementation of the row-structure algorithm	45