Two algorithms for the numerical factorization of large symmetric positive definite matrices for a hypercube multiprocessor

Rune Torkildsen
Program Systems Group
Department of Computer Systems
The Norwegian Institute of Technology
N-7034 Trondheim, Norway
e-mail: runetor@idt.unit.no

June 8, 1990

Abstract

This paper present two algorithms for the numerical factorization of large matrices. The matrices are assumed to be symmetrical positive definite which are typical matrices evolving from finite element method problems. The algorithms are designed to run on a hypercube multiprocessor.

1 Introduction

This paper gives a description of how the finite element method [FEM] may be implemented on a parallel multiprocessor with distributed local memory. The main concerns are devoted to the most time consuming process of this method which is the numerical factoriztion of the stiffness matrix **K**.

Two algorithms for the numerical factorization will be presented along with a brief theoretical comparative analysis. Both algorithms assumes that the finite element domain is discretized and distributed among the processors using the nested dissection method. This method makes it possible for the algorithms to utilize the structural parallelism, that is parallelism due to the matrix structure.

Solving a boundary-value problem by the finite element method will lead to the more general problem of solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ where **A** is an $n \times n$ matrix and **x** and **b** both are vectors with n elements.

The matrix **A** will for many boundary-value problems be a symmetric positive definite matrix. This assumption will be utilized when solving the system of linear equations. A symmetric positive definite matrix may be factorized such that $\mathbf{A} = \mathbf{LL^T}$ where **L** is a lower triangular matrix.

In [Geor81], George and Liu discuss four stages in the solution of large spares symmetric systems of equations:

1. *Ordering:* Find a 'good' symmetric permutation **P** of the coefficient matrix **A** which reduces the amount of fill-in which occurs during the factorization process.
2. *Symbolic factorization:* Determine the structure of the factor **L** and create a sparse data structure for this factor.
3. *Numerical factorization:* Compute the factor **L**.
4. *Triangular solution:* Solve the triangular systems $\mathbf{Ly} = \mathbf{Pb}$, $\mathbf{L^Tz} = \mathbf{y}$ and then set $\mathbf{x} = \mathbf{P}^T\mathbf{z}$.

The paper gives an brief indication of how step 1 and 2 may be done concurrently in addition to a more detailed description of the numerical factoriztion stage.

No description of how the triangular solution may be computed will be given in this paper. However, this stage can done in a similar but less complicated way as the factoriztion stage.

It has to be stressed that this paper is only meant as a presentation of ideas for implementation of the finite element method on a parallel computer. No implementation of the presented algorithms has yet been done. That means that no simulation results are available showing the efficiency of the algorithms.

An analytical approach is difficult due to the structural parallelism. That is, the algorithms may perform well for some systems, worse for others.

An outline of this paper is as follows: In section 2 a short introduction to the finite element method is given in order to clarify the notions and most basic concepts of the method.

Section 3 gives a description of how a symmetrical positive definite matrix **A** can be factorized using a parallel column-oriented version of the basic Cholesky factoriztion algorithm.

Section 4 considers how the nested dissection method could be utilized in order to find a good ordering for the matrix **A**. A relative numbering scheme based on this ordering will be introduced and will play an important role for the proposed algorithms.

Section 5 contains the two numerical factorization algorithms along with proposals for a data structure and message protocol. A brief comparative analysis of the two algorithms will be presented to the end of this section.

2 The finite element method

A brief description of the finite element method is presented is this section to clarify the notions and most basic concepts used throughout this paper.

The finite element is a general technique for constructing approximate solutions to boundary-value problems. The method involves dividing the domain of the solution into a finite number of simple subdomains, the finite elements, and using variational concepts to construct an approximation of the solution over the collection of finite elements. For example, consider the two-dimensional Laplacian equation:

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \qquad (2.1.)$$

The method attempt to solve such a problem by the determination of an approximate solution for ϕ at a finite number of discrete points in the domain. Figure 1 shows a typical finite element discretization of a two-dimensional problem.

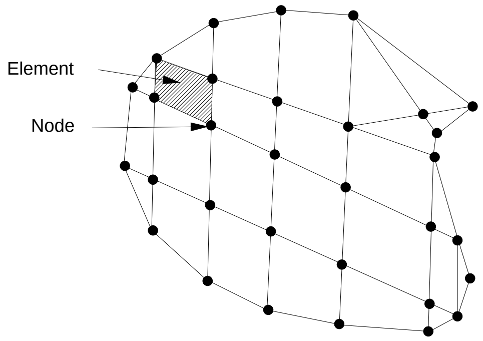


Figure 1. A two-dimensional finite element discretization of a domain.

A system of linear equations $\mathbf{K}\Phi = \mathbf{F}$ where the unknown ϕ_i comprise the desired approximate solution, will be constructed. The matrix **K** is normally referred to as the *stiffness* matrix and the vector **F** is referred to as the *load* vector. Φ is a vector consisting of the unknowns ϕ_i.

An element is as shown in figure 1, a subdomain made up by line segments between grid *nodes*. Each node is a part of at least one element and there exists a nonzero element in the stiffness matrix **K** connecting any two unknowns ϕ_i and ϕ_j whose node share a common element.

The stiffness matrix will therefore normally have a sparse structure. This sparseness will be exploited when solving the system of linear equations. An other important observation is that the computations of the matrix elements may be accomplished by calculating the contributions from each finite element and add them together in the global stiffness matrix. The computation of the element contributions is independent of each other. This may be expressed in the following manner:

$$\mathbf{K} = \sum_{e=1}^{E} \mathbf{K}^{e} \qquad (2.1.)$$

$\mathbf{K}^{e}$ is an $n \times n$ matrix with zero elements except the rows and columns corresponding to the nodes of the element **e**.

As an example, an assembly of a stiffness matrix along with two element matrices will be shown. The domain illustrated in this example consists of six elements and seven nodes numbered as indicated in the figure below.

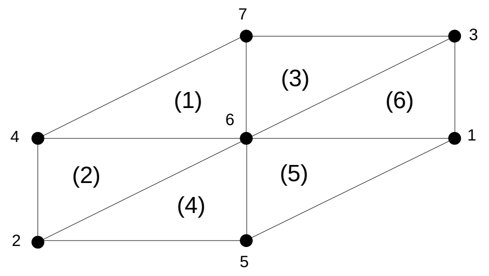


Figure 2. A domain discretized in 6 elements with 7 nodes.

The element matrices for element number 2 and 6 is shown below.

$$\mathbf{K}^{2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x & 0 & x & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x & 0 & x & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x & 0 & x & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{K}^{6} = \begin{bmatrix} x & 0 & x & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & 0 & x & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$\mathrm{K} = \sum_{e=1}^{6} \mathrm{K}^{e}$ gives the following stiffness matrix:

$$\mathbf{K} = \begin{bmatrix} x & 0 & x & 0 & x & x & 0 \\ 0 & x & 0 & x & x & x & 0 \\ x & 0 & x & 0 & 0 & x & x \\ 0 & x & 0 & x & 0 & x & x \\ x & x & 0 & 0 & x & x & 0 \\ x & x & x & x & x & x & x \\ 0 & 0 & x & x & 0 & x & x \end{bmatrix}$$

The elements of the load vector **F** may be obtained in a similar fashion.

A proper computation of the element matrices is important in determining the accuracy of the resulting solution. The governing differential equation is recasted in the form of a *variational equation*. This "weaker form" allows weaker conditions to the solution and its derivatives.

The space of acceptable solutions of the variational equation is assumed (in the discrete case) to be adequately represented by an *M*-dimensional space of basis functions where *M* is equal to the number of nodes. In the discrete problem (under the Galerkin approximation), these basis functions η_i are used to construct a trial solution. The approximate solution may now be expressed in the following manner:

$$\tilde{\phi} = \sum_{i=1}^{N} \phi_i \cdot \eta_i \tag{2.2.}$$

The form of a general entry in the stiffness matrix is determined by the governing variational principle of the problem. In general, a matrix element $\mathbf{K_{ij}}$ consists of an integral over the problem domain involving the basis functions η_i and η_j:

$$K_{ij} = \int\int f\left(\eta_i, \eta_j, \frac{\partial\eta_i}{\partial x}, \frac{\partial\eta_j}{\partial x}, \frac{\partial\eta_i}{\partial y}, \frac{\partial\eta_j}{\partial y}, \ldots\right) dx\,dy \tag{2.3.}$$

For three-dimensional problems this integral will be a volume integral.

The basis functions η_i are chosen such that they equal zero on every node except on node *i* where they equal 1. They are also chosen such that they have a simple linear or polynomial variation within an element. An example of functions with these characteristics are Legrande functions.

These criteria causes $\eta_i \neq 0$ only within those elements of which node *i* is a part. Hence the above integral may be restricted to the area of a single element to obtain the element stiffness contribution K^e_{ij}. For the Laplacian equation (2.1), the element stiffness matrix can be shown to have the form:

$$K^e_{ij} = \int\int\left(\frac{\partial\eta_i}{\partial x} \cdot \frac{\partial\eta_j}{\partial x} + \frac{\partial\eta_i}{\partial y} \cdot \frac{\partial\eta_j}{\partial y}\right) dx\,dy \tag{2.4.}$$

2.1 Characteristics and assumptions to the stiffness matrix

The rest of this paper will assume that **K** is a symmetric positive definite matrix[1)].

In numerous physical problems an exchange of the indices *i*, *j* in equation (2.4) will give the same answer. Hence, $K_{ij} = K_{ji}$ which is the definition of a symmetric matrix. It can also be shown that such matrices are positive definite.

Symmetrical positive definite matrices have the following characteristics which are utilized when solving the systems of equations $\mathbf{K\Phi} = \mathbf{F}$ by a direct method as the Gaussian elimination:

- **K** may be factorized such that $\mathbf{A} = \mathbf{LL^T}$ where **L** is a lower triangular matrix.
- The factorization process will in general be stable. Hence, there is no need for pivoting in the factoriztion phase.

3 Sparse Cholesky factorization

This section contains a general description of how a symmetric positive definite matrix **A** can be factorized such that $\mathbf{A} = \mathbf{LL^T}$. **A** is here used as a general $n \times n$ matrix whereas **K** is the stiffness matrix of an FEM problem.

A column-oriented version of the basic Cholesky factoriztion algorithm will be given. Also its potential for parallelism will be investigated twofold. First, its *natural parallelism* which is the inherent parallelism of the basic algorithm. Second, its *structural parallelism* which exploits the sparseness of the matrix.

1 A matrix is said to be positive definite iff the scalar $\nu\phi\nu^T > 0$ for all nonzero vectors ν.

The last part of this section describes the *elimination tree* which is used in order to identify the structural parallelism and the effect of different orderings for the matrix **A**. This part is based on the paper [Geor88] by George et. al. and is submitted here in order to clarify concepts in later sections.

3.1 The sequential column-Cholesky factorization algorithm

The sequential Cholesky factorization algorithm is shown in the figure below. Notice that **A** is overwritten by **L** in the code.

```
for j:=1 to n do
begin
    for k:=1 to j-1 do
        for i:=j to n do
            $a_{ij} := a_{ij} - a_{ik} \times a_{jk}$
    $a_{jj} := \sqrt{a_{jj}}$
    for k:=j+1 to n do
        $a_{kj} := a_{kj} / a_{jj}$
end
```

Figure 3. Sequential column-Cholesky factorization.

As described by Liu in [Liu86], a parallel version of this algorithm is well suited for medium- and large-grained multiprocessors. A parallel version of this algorithm can be obtained by identification av two subtasks:

cmod(j,k):	Modification of column *j* by column *k* ($k < j$) That is, $col_j := col_j - col_k * L_{jk}$, $k < j$
cdiv(j):	Modification of column *j* by its diagonal element.

Hence the algorithm may be rewritten in the following manner:

```
for j:=1 to n do
begin
    for k:=1 to j-1 do
        cmod(j,k)
    cdiv(j)
end
```

Figure 4. The cdiv and cmod operations.

As seen from the algorithm, the columns are eliminated[2)] sequentially starting with column number one. A different ordering of **A**[3)] will of course have no effect on the final solution. However when a particular ordering is found, it determines the order of elimination. As described in successive sections different orderings may have significant effect on the degree of parallelism.

2 When the *cdiv* operation for a column is completed, no more operations are necessary on this column. The column is then said to be *eliminated*.

3 To find an ordering of a matrix is the same as to permute the matrix by a permutation matrix **P**. In order to preserve the symmetry of the matrix **A**, the operation $\mathbf{PAP}^{\mathbf{T}}$ is performed.

3.2 *Natural parallelism* in the column-Cholesky algorithm

The natural parallelism of this algorithm may be observed by noticing the following dependencies between the two subtasks *cdiv* and *cmod*.

- *cdiv*(j) can not start before *cmod*(j,k) is done for all $k < j$.
- *cdiv*(j) has to be completed before it can be used to modify subsequent columns.

George et. al. illustrates in [Geor88] these precedence relations by a precedence graph as shown in the figure below.

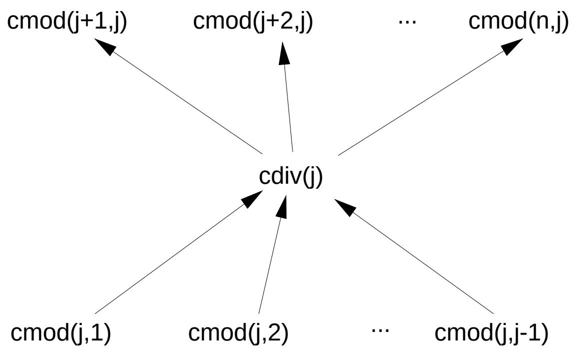


Figure 5. Subtask precedence graph for column-Cholesky.

As seen from the dependencies and the precedence graph the *cdiv* operations have to be done in a purely sequential manner. However, all the column modifications *cmod*(j,k) for all columns j, $j > k$, can be performed in parallel.

3.3 *Structural parallelism* in the column-Cholesky algorithm

The structural parallelism exploits the sparse structure of **A**. From the definition of *cmod*, it is easy to see that the column j is only modified by columns k where $L_{jk} \neq 0$. Also, after *cdiv*(j) is completed the column j will only affect subsequent columns where $L_{jk} \neq 0$.

Note, it is the structure of **L** and not the structure of **A**, that determines the criteria above.

Hence if a column j does not depend on any modifications from any column k, $k < j$, it may be eliminated. In a sparse structure several columns may satisfy this condition at the same time and may therefore be eliminated in parallel.

The structure of the Cholesky factor **L** will normally contain some additional nonzero elements. These extra nonzero elements are known as *fill-ins*. To minimize the number of *cmod* operations it is important to minimize the number of fill-ins.

To fully utilize the structural parallelism it is necessary to find a good ordering **P** for **A**. This ordering should also try to minimize the number of fill-ins. As said by George et.al. in [Geor88] these objectives turn out to be mutually complementary.

3.4 The *elimination tree* and the effect of ordering

In order to identify the structural parallelism corresponding to a specific ordering, elimination trees are useful. An elimination tree visualises the structure of **L** such that the column dependencies can be identified. In [Geor88], the elimination tree corresponding to the structure of L is defined in the following manner.

First define $\gamma[j]$:

$$\gamma[j] = \min\{i \mid L_{ij} \neq 0\,,\, i > j\}$$

That is, $\gamma[j]$ it is the row subscript of the first off-diagonal nonzero in column j of **L**. If column j has no off-diagonal nonzero, $\gamma[j]$ is set to j. ($\therefore\ \gamma[n] = n$).

An elimination tree has n nodes[4], one for each column, labelled from 1 to n. For every node j, if $\gamma[j] > j$, then node $\gamma[j]$ is the *parent* of node j and the node j is one of possibly several *child* nodes of node $\gamma[j]$. It is assumed that the matrix **A** is *irreducible*, so that n is the only node with $\gamma[j] = j$ and it is the *root* of the tree.

$$\therefore\ \gamma[j] > j \quad \mathit{for}\ 1 \leq j < n$$

If **A** is reducible the elimination tree will become a forest which consists of several trees.

The example in figure 2 will have the following Cholesky factor **L** and elimination tree. Note, ν denotes fill-ins in the Cholesky factor **L**.

$$\mathbf{L} = \begin{bmatrix} x & & & & & & \\ 0 & x & & & & & \\ x & 0 & x & & & & \\ 0 & x & 0 & x & & & \\ x & x & \nu & \nu & x & & \\ x & x & x & x & x & x & \\ 0 & 0 & x & x & \nu & x & x \end{bmatrix}$$

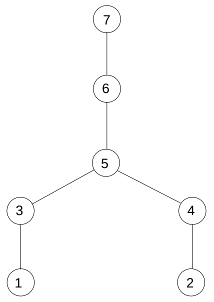


*Figure 6. A Cholesky factor **L** with corresponding elimination tree.*

The elimination tree provides information about the column dependencies. For example, *cdiv*(i) can not be executed before *cdiv*(j) has completed for all descendant nodes j of node i.

If node i and node j is on the same level in the tree, then their respective columns may be eliminated in parallel if all their descendant nodes already are eliminated.

In order to get a high degree of structural parallelism the matrix is ordered to minimize the height of the tree. As mentioned before, it is desirable that such an ordering also minimize the number of fill-ins. It is so that some of the well known fill-in reducible ordering algorithms like "*nested dissection*" and "*minimum degree*" generates fine[5] elimination trees. These ordering algorithms are described by George and Liu in [Geor81].

The example in the figure 7 shows a finite element grid with two different orderings. The one to the left corresponds to the "*nested dissection*" ordering algorithm, the one to the left to a band-oriented method[6]. The resulting structures of **L** and their corresponding elimination trees show clearly that some ordering schemes are better suited to exploit the structural parallelism than others. Notice that the number of fill-ins equals zero in both cases.

In several proposed numerical factorization algorithms the elimination tree is derived from the structure of **L** and utilized during the factorization process.

4 The node j corresponds to the column j in the Cholesky factor **L**. The word node will often be used throughout this paper.

5 A *fine* elimination tree is a tree with relative low height.

6 A band-oriented ordering algorithm tries to keep the structure of L closely into the diagonal. This is optimal with respect to storage overhead.

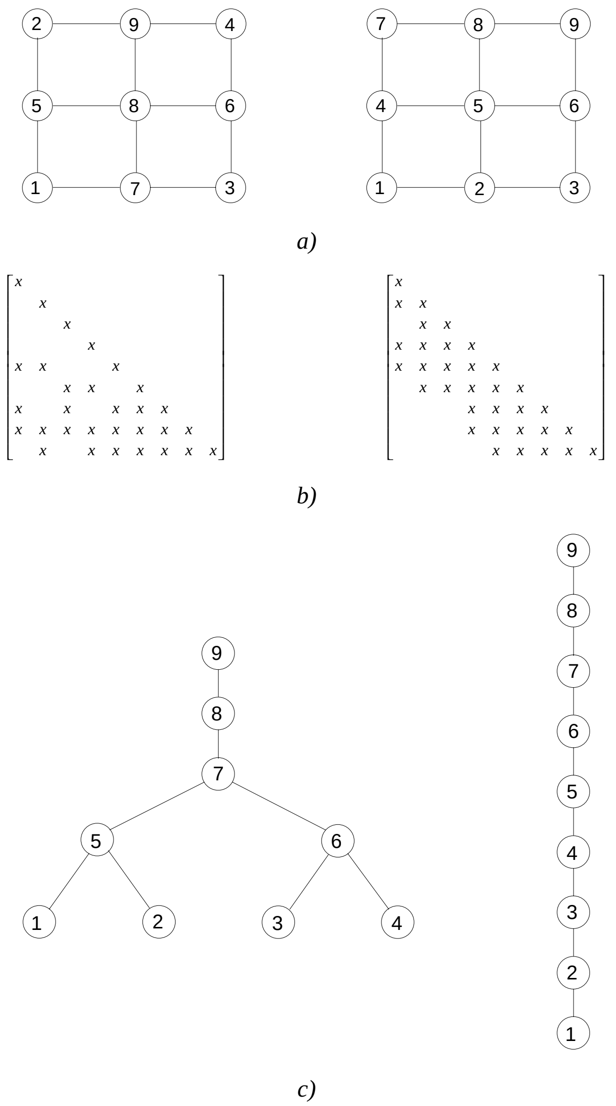


Figure 7. a): Two different ordering schemes for an FEM grid b): The corresponding structures of ***L*** *for the ordering schemes in a), c): The corresponding elimination trees.*

4 The nested dissection ordering scheme for a hypercube multiprocessor

This section describes how the nested dissection method is utilized for the proposed numerical factorization algorithms. An algorithm for the nested dissection method will not be given, however a brief explanation of how this may be done is included.

As mentioned in the previous section, the nested dissection method will provide elimination trees with relative low heights. This method therefore makes it possible to utilize the potential structural parallelism to some extent.

A relative numbering scheme well suited for a hypercube multiprocessor will be presented. This numbering scheme is derived from the nested dissection method and is useful in order to recognize the order of elimination.

4.1 The nested dissection algorithm

For an FEM domain, the nested dissection algorithm will, as the name indicates, first divide the domain in two subdomains and label the border nodes sequentially from *n* and downwards. Then the two subdomains will further be divided into two new smaller subdomains labelling the border nodes from the highest available number and downwards. This domain decomposition will continue in the same manner until no more decompositions is possible.

For a hypercube multiprocessor this can be described by the following algorithm.

> *For a hypercube of dimension* ***d****:*
>
> > *Divide the domain in two such that each part will have approximate the same size (in the number of nodes) and to minimize the number of border nodes.*
> >
> > *Submit each subdomain to a subcube of dimension* ***d****-1*

Figure 8. Decomposition of an FEM domain by the nested dissection technique.

This algorithm will be repeated in a recursive manner until each processor (d=0) has obtained its part. It is assumed here that the number of processors is less than the number of elements.

The numbering of the nodes may be done by numbering the border nodes from n and downwards where n is the number of nodes in the subdomain to be divided. It does not matter if nodes in different subcubes have the same node number since they can be uniquely identified from the subcube which they belong to.

To illustrate the nested dissection algorithm, an example containing a finite element grid with 7×7 nodes is shown below.

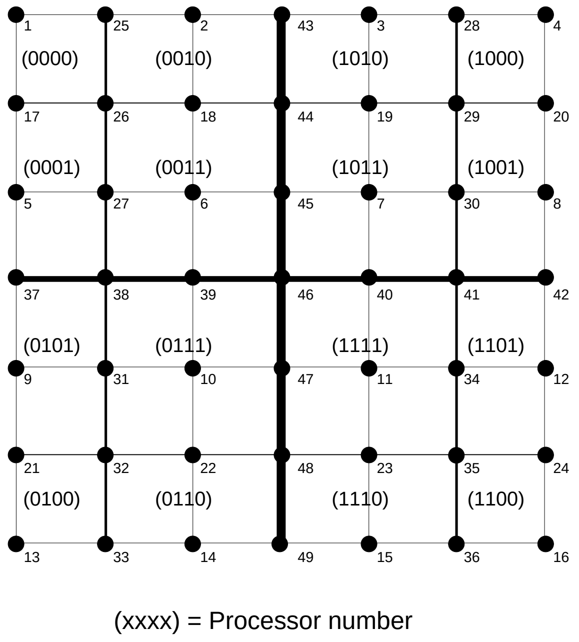


Figure 9. An element grid containing 36 elements and 7×7 nodes.

4.2 A relative numbering scheme

The relative numbering scheme is a way to number the border nodes utilizing the nested dissection algorithm described in the previous section. The numbering scheme will number or order the nodes relatively to the dimension of the subcube they belong to[7)].

Definition 1:

"A node is said to belong to a subcube of dimension ***d*** *if it lies on the border between two subcubes of dimension* ***d****-1 or the subcube is not further divided."*

A relative node number consists of two numbers ($d.i$) where d indicates the subcube dimension and i is the relative node number within this dimension. For example will node (2.3) be node number 3 in a subcube of dimension 2. The nodes 27, 30, 32 and 35 in figure 9 are all identified by this number but will reside different subcubes.

Relative numbering of the nodes in figure 9 gives the following result.

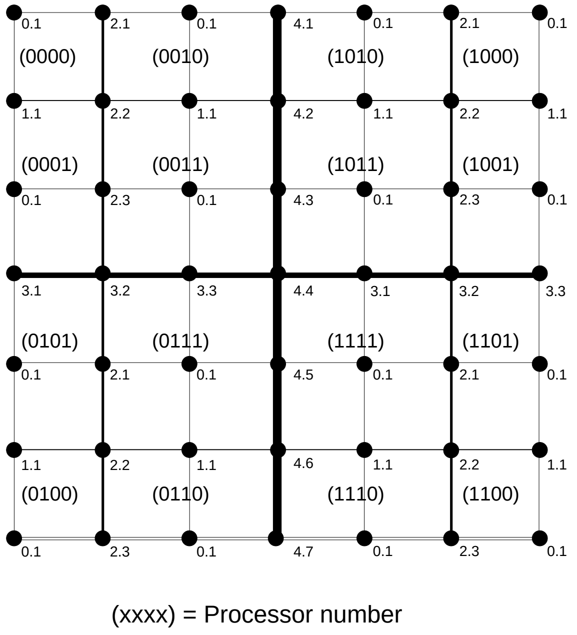


Figure 10. Relative numbering of a 7×7 grid.

The border nodes of a subcube are supposed to have a higher ordering numbers than its internal nodes in order to determine the order of elimination corresponding to the nested dissection method. That leads to the following definition.

Definition 2:

"A relative ordering number (a.b) is said to be higher than the relative ordering number (x.y) iff $(a > x) \vee ((a = x) \wedge (b > y))$*"*

The relative numbering scheme will play an important role in the proposed numerical factorization algorithms and data structures.

7 Physically, this means that the node belongs to the subdomain which is allocated to the processor.

5 Numerical factorization algorithms for a hypercube

This section describes two different parallel algorithms for numerical factorization of sparse symmetric positive definite matrices. The algorithms are designed in connection to finite element method problems which typically leads to problems of this type.

The algorithms are based upon the column-Cholesky algorithm and will exploit both the natural and the structural parallelism. To illustrate details of the algorithms and the purposed data structure, the 7×7 grid example from the previous section will be used. This element grid is spread over a hypercube of dimension 4 (16 processors).

Such grids are often used as examples since they are relatively simple to analyse and quite common in two-dimensional FEM problems.

Both algorithms assume that three preprocessing stages are completed before the numerical factoriztion can begin. These three stages, *initialization*, *numerical integration* and *symbolic factorization* are described in the next three subsections.

Then, a short presentation of a proposed data structure and message protocol will follow. The data structure and message protocol is mainly submitted here in order to show how the relative numbering scheme may be utilized.

The first algorithm presented is based on an algorithm given by George et. al. in [Geor88]. The second algorithm tries to improve on the first one in the way the communication is performed and by splitting the *cmod* operation into two parts. Both algorithms is designed for a hypercube multiprocessor.

Finally, a short comparative analysis between these two algorithms is done.

5.1 The initialization stage

In this stage the finite element domain will be decomposed and distributed according to given domain data. The distribution to the various subcubes and processors may be done as indicated in section 4.1. This stage is also responsible for the distribution of the following information to the various processors.

- Element data for all internal elements. That is, information necessary to perform the numerical integration, see section 2, plus the relative node numbers to the element nodes.
- The relative node number to every node that belongs to the different subcubes ($d \geq 0$) the processor is attached to. That is, all internal nodes and every border node belonging to subcubes which the processor is a part of.

The border nodes are assumed to be distributed in the following way: The nodes that belongs to a particular subcube is distributed among all the processors within the subcube in an alternating sequence. The sequence will follow the Gray code such that border nodes next to each other will reside in processors one hop distance away. (neighbour processors)

As an example the border nodes that belongs to the two-dimensional hypercube (00xx) (figure 10) will be distributed among the processors 0-3 in the following manner:

2.1	2.2	2.3	Node number
p0	p1	p3	Processor number

Figure 11. Distribution of the border nodes belonging to the two-dimensional hypercube (00xx).

This distribution method is similar to a method used by George et. al. in [Geor89] which yields good results. The method scatter *cdiv* and *cmod* operations equally among the processors.

5.2 The numerical integration

Every processor calculates and generates the element matrices for its elements. See also section 2.

5.3 The symbolic factoriztion

Before the numerical factorization starts it is necessary for the processor to create a sparse data structure to hold the nonzero elements of the Cholesky factor **L**.

In some parallel versions of the Cholesky algorithm this stage will also compute the elimination tree corresponding to the structure of **L** for utilizing this information during the factorization phase. A good description of such an algorithm is found in the paper [Gilb88] by Gilbert and Hafsteinsson.

The proposed factorization algorithms are not dependant on computation of the elimination tree since this information is stored implicit in the way the nodes are distributed and ordered.

The symbolic factorization is supposed to consist of the following two parts:

1: Symbolic factorization of the processors internal nodes, which equals the symbolic factorization performed by sequential algorithms. One way to carry out this task is to make use of corollary 2 in Duff’s article [Duff86]:

Statement 1:

*"The structure of any column, k say, of **L** is the union of the sparsity structure of all the previous columns of **L** whose first nonzero beneath the diagonal is in row k."*

2: To determine the data structure of all border nodes the processor is responsible for. It can be verified that the structure of column *j* should be able to store an element for every column *k*, where $k > j$, which belongs to the same border or belongs to a border which surrounds the subcube containing the column *j*.

This may be verified by a corollary to theorem 5.1.2 i the book [Geor81] by George and Liu:

Statement 2:

*"If there exists a path of nodes **r** between the node **j** and **k**, where $j < k$ and $\forall r, r < j$, node **j** will have an influence on node **k**."*

Since the border nodes belonging to a subcube always will have higher ordering number than its internal nodes, the following corollary may be derived from the corollary above:

Statement 3:

*"A border node **j** belonging to a subcube of dimension **d** will have influence on <u>every</u> border node on borders surrounding the subdomain the subcube is responsible for."*

For example, the node (2.2) belonging to the subcube (00xx), see figure 10, should obtain the following data structure.

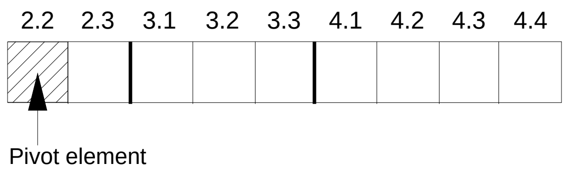


Figure 12. Data structure to node (2.2) belonging to the subcube (00xx).

5.4 A proposed data structure for the Cholesky factor L

This subsection gives a brief description of a proposed data structure for **L** that utilizes the relative numbering scheme. The examples used focuses on the data structure for border nodes. A similar data structure may be used for the internal nodes. Throughout this subsection the examples will refer to figure 10.

The data structure consists of two parts:

1: A map-table which for each border node, contains the processor number of the processor responsible for the elimination of the node and a pointer referring to the data structure of that node.

2: The data structure storing the column data.

The first will be referred to as the *map-table*, the latter one as the *column data structure* or just the *data structure*.

In the figure below a map-table for processor 6 (0110) is shown. Note that only entries for border nodes belonging to subcubes which the processor is a part of is found in the table. The border nodes are allocated to the processors using the Gray code numbering scheme as described in section 5.1.

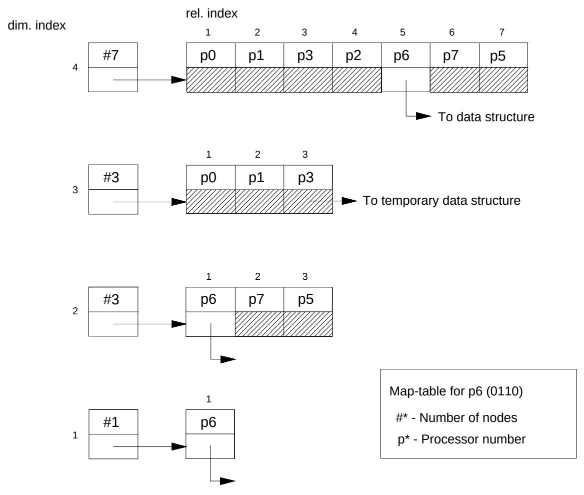


Figure 13. The map-table for processor 6 (0110), see figure 10.

The relative numbering scheme naturally leads to a two-dimensional table with the dimensional number *d* and the relative number *i* of an ordering number (*d*.*i*) as indices.

Assuming the hypercube dimension is fixed and equal to *dim*, then the dimensional index *d* may be used as an index to a fixed size one-dimensional table with *dim* entries and the relative index as an index to a dynamically created one-dimensional table.

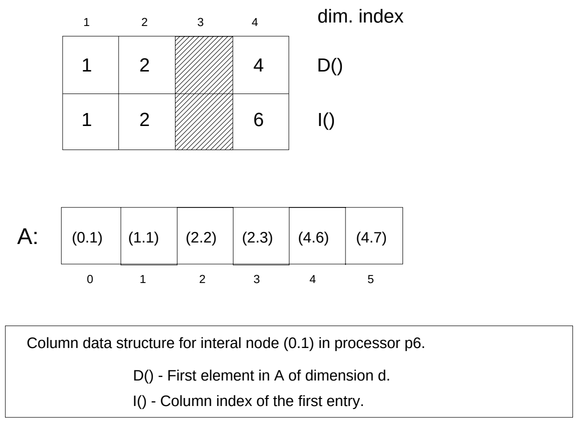
The pointers to the column data structure can be used in two ways:

1: If the processor is responsible for the node it self, the pointer refers to the physical position of the column data structure of that particular node.

2: If the node belongs to another processor, the pointer may refer to a temporary data structure for that node. This is utilized by the second of the two proposed fatorization algorithms.

For a particular border node (*a.b*) belonging to a subcube of dimension *d*, then using statement 3, section 5.3, and the way the border nodes belonging to a subcube is ordered[8)], it can be verified that border nodes with higher ordering number influenced by the border node (*a.b*) will lie in a sequence.

This observation is utilized in the design of the column data structure. For every data structure there will exist two fixed one-dimensional tables *D()* and *I()* which contains secondary data necessary to find the wanted column data. An example of the data structure is shown below.



1 2 3 4 dim. index
1 2 4 D()
1 2 6 I()
A: (0.1) (1.1) (2.2) (2.3) (4.6) (4.7)
0 1 2 3 4 5
Column data structure for interal node (0.1) in processor p6.
D() - First element in A of dimension d.
I() - Column index of the first entry.

Figure 14. The proposed data structure for node (0.1) belonging to processor no. 6.

D(i): The index of the first entry in the column data structure *A* belonging to a border dividing a subcube of dimension *i* in two subcubes of dimension *i*-1.

I(i): The column index of that first entry.

If for example, the entry for $\mathbf{L}_{(2.3),(0.1)}$ is to be modified for the given example then it may be found in the following way:

- Find the data structure corresponding to column (0.1) by using an internal map table[9)].
- Then find the correct entry by using the formula: $A[d.i] = A[D(d) + (i - I(d))]$, which gives, $A[2.3] = A[D(2) + (3 - I(2))] = A[2 + 1] = A[3]$

5.5 A proposed message protocol

The efficiency of the factorization algorithms will be dependant on effective communication. If the number of internal nodes (0.*) to a processor is very large compared to the number of border nodes, then the algorithms will be compute bound and the communication overhead will not be significant.

Nevertheless, a message protocol should be carefully designed. The protocol described here utilizes the relative numbering scheme in order to minimize the amount of secondary data.

8 In a sequence from 1.

9 The node (0.1) is one of possible several internal nodes to the processor. Internal nodes are not indicated in the map-table shown in figure 13. A similar table will exist for internal nodes.

It is only necessary to send an eliminated column $\mathbf{L}_{*,(a.b)}$ to every processor responsible for a column influenced by this column. That is to the column (x.y) corresponding to every nonzero elements in $\mathbf{L}_{*,(a.b)}$, (x.y) > (a.b). Also it is only necessary for column (c.d) to receive column entries $\mathbf{L}_{(x.y),(a.b)}$ where $(x.y) \geq (c.d)$ in order to perform the column modifications from column (a.b).

The proposed message protocol is shown in the figure below. The fields for the destination and status will equal the system message protocol and will not be further commented.

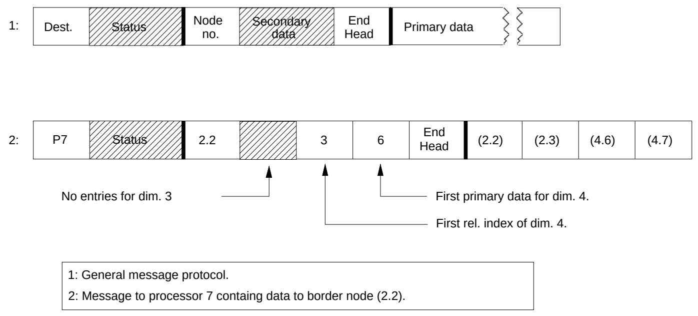

1:
Dest.
Status
Node no.
Secondary data
End Head
Primary data
2:
P7
Status
2.2
3
6
End Head
(2.2)
(2.3)
(4.6)
(4.7)
No entries for dim. 3
First primary data for dim. 4.
First rel. index of dim. 4.
1: General message protocol.
2: Message to processor 7 containg data to border node (2.2).

Figure 15. Proposed message protocol.

As indicated by the figure the message protocol consists of three parts:

1: The header which contains the destination address and some other status information. Equals the general system message protocol.

2: The ordering number of the column to be modified along with secondary data to identify the primary data which succeeds this part.

3: The primary data.

The secondary data is based on the secondary data structure used in the proposed column data structure in the previous section. The ordering number (*d.i*) will be succeeded by entries of the one-dimensional tables *D()* and *I()* described in the previous section. For example, the ordering number (*d.i*) is succeeded by $D(d+1)$, $I(d+1)$, $D(d+2)$, $I(d+2)$, ...

$D(d+1)$ will for the message protocol denote the message sequence number of the first primary data $\mathbf{L}_{(d+1.x),(d.i)}$. The secondary data sequence is ended by a marker as indicated in the figure.

If there is no primary data for a dimension, the succeeding index number may be omitted. The secondary data in the figure above is supposed to be interpreted as follows:

The primary data is going to modify node (2.2). There are no entries for modifying $\mathbf{L}_{(3.*),(2.2)}$. The third primary data is going to modify $\mathbf{L}_{(4.6),(2.2)}$. The second and fourth primary data is identified by using the sequence assumption described in the previous section.

5.6 A numerical factorization algorithm

The first proposed factorization algorithm is based on an algorithm by George et. al. in [Geor88]. This algorithm is adjusted to suit an FEM problem. The algorithm will also take advantage of how the nodes are distributed and the relative ordering scheme.

In contrast to the algorithm in [Geor88], this algorithm assumes that the three preprosessing stages are completed by the processors. That is, the factorization algorithm may start without any communication with the host.

The algorithm may be described in the following way:

```
Processor pno:
    Eliminate all internal nodes (0.*).
    Participate in a global assembly of the border nodes.
    Participate in elimination of the border nodes.
```

Figure 16. Simplified factorization algorithm.

In order to simplify the notion of the algorithm, three support variables are introduced. The first two variables are supposed to be initialized in the symbolic factorization phase, the latter one within the global assembly phase of the algorithm.

ncol: The number of border nodes processor *pno* is responsible for.

map(d.i): The processor number to the processor responsible for the elimination of node *d.i*. Every processor has a *map*-table for every border node belonging to subcubes which the processor is a part of.

nmod(d.i): The number of column modifications or *cmod* operations by columns belonging to the two subcubes of dimension *d*-1 plus the number of *cmod* operations by columns on the same border with lower ordering number. Every processor has an *nmod*-table for every border node it is responsible for.

For example, *nmod*(2.3) for the border node (2.3) belonging to subcube (00xx) in figure 10 is equal to four, because it exists two border nodes with smaller ordering number on the same border in addition to two border nodes belonging to the subcubes (000x) and (001x).

From statement 3 in section 5.1, the following important corollary may be derived:

Statement 4:

"A border node belonging to a subcube of dimension **d**, *is modified by <u>all</u> border nodes belonging to the two subcubes of dimension* **d**-*1."*

That is, if a border node *d.i* has performed *cmod* operations from every border node belonging to the two subcubes of dimension *d*-1, then it can be taken for granted that <u>every</u> border node belonging to subcubes of dimension less than *d*-1 is eliminated.

It is also assumed that the following two primitives exists for communication purposes on the hypercube:

send: Send a message to a specified processor.

wait: Wait for a message to arrive. This primitive will suspend the processor until a message is received.

The factorization algorithm is shown in figure 17. The algorithm contains two subroutines *assemble_border_nodes* and *broadcast*. These subroutines will be described in successive sub-sections. For a description of the subroutines *cdiv* and *cmod*, see section 3.1.

$^{(*)}$ refers to comments after the algorithm.

```
processor pno:
eliminate internal nodes        { using a sequential algorithm. }
assemble_border_nodes           { all-to-all communication }
{ start elimination of the border nodes. }
if map(1.1) = pno then (1)
begin
    cdiv(1.1)
    broadcast(1.1, $L_{*, 1.1}$)
    ncol := ncol - 1
end
while ncol > 0 do
begin
    Wait for a column of L; $L_{*, a.b}$
    for every nonzero element $L_{x.y, a.b}$ where map(x.y) = pno do
    begin
        cmod(x.y, a.b)
        if $a \geq x - 1$ (2) then
        begin
            nmod(x.y) := nmod(x.y) - 1
            if nmod(x.y) = 0 then
            begin
                cdiv(x.y)
                broadcast(x.y, $L_{*, x.y}$)
                ncol := ncol - 1
            end
        end
    end
end
```

Figure 17. A parallel algorithm for numerical Cholesky factorization for a stiffness matrix **K**.

Comments:

(1) The only border nodes which are independent of other border nodes, are border node 1 in the subcubes of dimension 1. These nodes may therefore be eliminated as soon all the internal nodes are eliminated and the border nodes are fully assembled.

(2) From the definition of *nmod*(x.y), only *cmod* operations on border node (a.b) from border nodes from the same border (a=x) or from the subcubes of dimension x-1 (a=x-1) will have any effect on *nmod*(x.y).

The subroutine broadcast(x.y, $L_{*, x.y}$)

```
broadcast(x.y, L*, x.y):
for every element a.b in $L_{*, x.y}$ [1] {$L_{a.b,x.y} \neq 0$} do
    send $L_{*, x.y}$ to the processor with pno = map(a.b)
{ If necessary inform the host processor. [2] }
```

Figure 18. Algorithm for the subroutine broadcast().

As shown in the above algorithm, *broadcast*() sends the computed column in the Cholesky factor **L** to every processor which is responsible for a column influenced by this column.

The main factorization algorithm immediately executes this operation when a column is eliminated in order to let the necessary *cmod* operations be executed in parallel. Hence, the elimination of the border nodes will follow the precedence graph shown in figure 5.

Comments:

(1) Only the column elements $L_{d.i, x.y}$ with ordering number (d.i) larger or equal to (a.b) will be involved in modification of column (a.b) and is thus necessary to send.

The figure below shows which elements from the data structure of node (2.2) belonging to subcube (000x) which is necessary to send to the processor responsible for the border node (3.3).

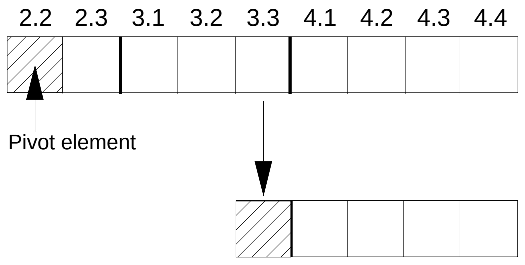


Figure 19. Sending of column entries.

(2) No algorithm for the host processor will be given in this article.

The subroutine assemble_border_nodes

This subroutine will assemble all finite element contributions for every border node and place the sum in the processor responsible for the node. As mentioned before, this subroutine will also compute the *nmod*-table for all border nodes.

In order to make the algorithm easier to understand, the following notion is introduced:

Statement 5:

*"Border number **i** is the border dividing a subcube of dimension **i** in two subcubes of dimension **i**-1. Every border nodes on the border **i** belong, according to definition 1, to the subcube of dimension **i**."*

```
assembler_grensenoder:

for $i$ := 1 to $dim$ (1) do
begin

    if $pno \wedge 2^{i-1} = 0$ then
    begin
        send column data (2) to neighbour cube $pno \otimes$ (3) $2^{i-1}$
        wait for column data (4) from neighbour cube $pno \otimes 2^{i-1}$
    end

    else
    begin
        wait for column data from neighbour cube $pno \otimes 2^{i-1}$
        send column data to neighbour cube $pno \otimes 2^{i-1}$
    end

    Maintain the data structure for nodes on the border $j$, $j \geq i$, which the processor are
    responsible for.

    Maintain data for nodes on border $j$, $j > i$, which are going to be transmitted in step
    $i$+1 (5).

    For nodes on border $i$+1, send the number of nodes on border.(6)

end
```

Figure 20. The subroutine assemble_border_nodes

.

Comments:

(1) *dim* is the hypercube dimension.

(2) Column data which is sent in step *i* are finite element contributions to border nodes on border *j*, where $j \geq i$ which are the responsibility of other processors.

(3) $\otimes$ indicates an exclusive-or operation.

(4) Column data which is received in step *i* are finite element contributions from border nodes on border *j* where $j \geq i$.

(5) Column data belonging to nodes on border *j*, $j > i$ which processor *pno* is not responsible for are added to own contributions, if any, and forwarded in step *i*+1.

(6) Study the subsection about the generation of *nmod*(*) below.

Generation of the *nmod*-tables:

From the definition of *nmod*(*d.i*), this variable should be initialized in the following manner:

$$nmod(d.i) = \text{The number of border nodes on the borders } (d\text{-}1) + (i\text{-}1) \qquad (5.1.)$$

The processor responsible for the border node (*d.i*) belongs to a subcube of dimension *d*. Initially the processor will only have information about the number of border nodes belonging to the subcube of dimension *d*-1 which the processor is a part of. For example if processor (0001) is responsible for the border node (2.2), the processor will know the amount of border nodes on border 1 in subcube (000x) but it does not know the amount of border nodes on border 1 in subcube (001x).

The example shown in figure 10, indicates an equal amount of border nodes on borders within subcubes of the same dimension. This is just accidentally, and should not be taken for granted in the general case.

This problem may be solved, as indicated in the algorithm, if processor *pno* sends information about the number of border nodes on border *i* during step *i*. This may be accomplished by using the following message protocol[10].

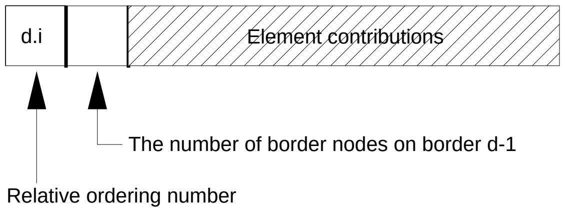


Figure 21. Message protocol for sending of finite element contributions during the global assembly phase.

The field for the number of border nodes is initially set to zero and will only be modified if the column data is forwarded in step *d*-1. The processor responsible for the node (*d.i*) adds this field to the variable *nmod*(*d.i*) after reception of such a message.

5.7 A variant factoriztion algorithm

This subsection describes a variant numerical factoriztion algorithm which is supposed to perform better than the original one described in the previous subsection.

This algorithm has two major differences compared to the other algorithm. There is a different strategy for the *cmod* operation and for the assembly of the border nodes. These changes will hopefully give a better overall performance for the elimination of the border nodes.

The *cmod*(a.b, x.y) operation for border nodes (a.b) and (x.y) residing in different processors is divided in two subtasks. As described in section 3.1, the *cmod*(a.b, x.y) operation may be expressed as follows:

$$column(a.b) = column(a.b) - column(x.y) \cdot \mathbf{L}_{(a.b),(x.y)}$$

Since the scalar $\mathbf{L}_{(a.b),(x.y)}$ is an entry of column(x.y) the multiplication part $column(x.y) \cdot \mathbf{L}_{(a.b),(x.y)}$ may be completed before the processor responsible for the column (x.y) sends it to the other processor. The result of the multiplication will be stored in a temporary data structure for the column (a.b) termed (a.b)'.

Hence the following operation may be introduced:

$$semi\_cmod(a.b,x.y) \equiv column(a.b)' = -column(x.y) \cdot \mathbf{L}_{(a.b),(x.y)}$$

Note, the algorithm does not perform this operation for nodes belonging to the same border (a=x). For these nodes the ordinary *cmod* operation is used.

The border nodes belonging to a subcube of dimension *d* will receive contributions from nodes belonging to subcubes of lower dimensions only when all border nodes belonging to the subcube of dimension *d*-1 are eliminated.

10 This message protocol should not be confused with the previous proposed protocol which is designed for the actual factorization stage of the factorization algorithm.

The variant factorization algorithm is shown in figure 22. Note that the support variable *ncol*[*d*] now is a table containing the number of border nodes belonging to the subcube of dimension *d* the processor *pno* is responsible for. Also, the subroutine *broadcast*() will be modified to only broadcast the eliminated column within the current subcube.

```
processor pno:
eliminate internal nodes    { using a sequential algorithm }
for d := 1 to dim do
begin
    exchange data for nodes in the current subcube (dim = d)
    { all-to-all communication within the current subcube }
    if map(d.1) = pno then
    begin
        cdiv(d.1)
        broadcast(d.1)            { within the current subcube }
        ncol[d] := ncol[d] - 1
        for every nonzero element $L_{e.y, d.1}$ where e > d do
            semi_cmod(e.y, d.1)
    end
    while ncol[d] > 0 do
    begin
        wait for a column of L, say $L_{*, d.a}$
        for every nonzero $L_{d.x, d.a}$ where map(d.x) = pno do
        begin
            cmod(d.x, d.a)
            nmod(d.x) := nmod(d.x) - 1
            if nmod(d.x) = 0 then
            begin
                cdiv(d.x)
                broadcast(d.x)            { within the current subcube }
                ncol[d] := ncol[d] - 1
                for every nonzero element $L_{e.y, d.x}$ where e > d do
                    semi_cmod(e.y, d.x)
            end
        end
    end
end
```

Figure 22. The variant numerical factoriztion algorithm.

5.8 A comparative analysis of the two algorithms

This subsection contains a brief comparative analysis of the two proposed algorithms. The analysis is divided into two main parts, how the natural parallelism is utilized and how communication between the processors is performed.

The utilizing of the natural parallelism

The elimination of the border nodes within a subcube have to be done in a sequentially manner utilizing only the natural parallelism. Hence, the higher dimension of the subcube the less columns may be eliminated in parallel.

By doing the most time consuming part of the *cmod* operation, the multiplication, before sending column entries to border nodes belonging to subcubes of higher dimension, the natural parallelism may be better exploited.

Of course, in the way this is done by the variant algorithm additional storage is needed since temporary data structures for border nodes belonging to other processors will be necessary. However, if the number of internal nodes are much larger compared to the number of border nodes, this extra storage overhead may be worthwhile.

The different communication strategies

By comparing the two strategies for exchanging the column data, I will predict that the communication strategy for the variant algorithm will give less communication overhead than the other.

The original algorithm communicate in a 'data-flow basis'. That is, column data is sent immediately after a column is eliminated. Especially in the earlier stages of the factorization process this may lead to several messages being sent to a particular processor simultaneously which may give a contention problem.

The variant algorithm will instead of immediately sending the column entries perform the *semi_cmod* operation and store the intermediate result in a temporary data structure. After a subcube of a given dimension *d* has eliminated all of its border nodes it will participate with the other subcube of the same dimension in order to assemble the border nodes to be eliminated in the next step.

This assembly will be an all-to-all communication process within the subcube of dimension d+1 and will be completed in d+1 hops using a well known communication algorithm.

This strategy will also make it possible to concatenate several columns before sending which leads to fewer and longer messages.

6 Summary

This paper has presented some ideas for implementation of the numerical factoriztion of an $n \times n$ matrix using a column-oriented version of the basic Cholesky algorithm.

Two algorithms were presented where the latter of the two is predicted to be the most efficient.

The presentation and explanation of the two algorithms are focused on the problem of elimination of border nodes resulting from by dividing the finite element domain into subdomains by the nested dissection method.

This is the part of the fatorization algorithms which are dependant of communication between the processors. In order to obtain a speedup factor[11)] close to the number of processors, it is necessary to minimize the communication overhead.

One might argue that if the number of internal nodes is large compared to the border nodes, most of the computation time is spent during elimination of the internal nodes. That is, a speedup factor close to the number of processors will be achieved almost independently of the efficiency of elimination of the border nodes.

11 In this context, the speedup factor is the computation time to the parallel algorithm compared to its sequential ancestor.

However, the elimination process will become more and more serial for subcubes of higher dimensions. Hence, the elimination of the border nodes may take a significant part of the overall computation time, at least for medium-sized problems. Therefore, minimizing the communication overhead will be worthwhile.

An implementation of the two proposed algorithms is planned in the near future as a pre-project for my Ph.D thesis. The implementations will be done on the HC-16, 16-node hypercube developed within our department. It is also desirable, if possible, to run the implementations on a commercial hypercube to investigate differences due to the computer architecture.

References

[Duff86] I.S. Duff, *Parallel implementation of multifrontal schemes*, Parallel Computing, 3 (1986), pp. 193-204

[Geor81] J.A. George og J.W.H. Liu, *Computer solution of large sparse positive definte systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981

[Geor88] J.A. George, M. Heath, J.W.H. Liu og E. Ng, *Sparse Cholesky factorization on a local-memory multiprosessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 327-340

[Geor89] J.A. George, J.W.H. Liu og E. Ng, *Communication results for parallel sparse Cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287-298

[Gilb88] J.R. Gilbert og H. Hafsteinsson, *Parallel solution of sparse linear systems*, Proceedings of the SWAT-88 conferance, Springer-Verlag lecture notes in computer science, vol. 318, pp. 145-153

[Liu86] J.W.H. Liu, *Computational models and task scheduling for parallel sparse Cholesky factorization*, Parallel Computing, 3 (1986), 327-342

Index