EPCC-SS99-01

# The Parallelisation of a Finite Element Code on a Shared-Memory Computer using OpenMP

Klaus Aehlig

**Abstract**

The finite element method (FEM) is a computer-aided mathematical technique for obtaining approximate numerical solutions to the abstract equations of calculus that predict the response of physical systems subject to external influences.

A typical FE code was parallelised using the `OpenMP` programming standard for shared memory machines. The original direct linear algebra solver has been replaced by an iterative method. The parallelised code scales on the 8-processor shared memory Sun 3500 at EPCC.

**Keywords:** OpenMP, Shared Memory Machines, Finite Element Method, Iterative Methods

## Contents

**1 Introduction 3**

**2 Background 3**
2.1 Shared Memory Computers and `OpenMP` . . . 3
2.2 The Finite Element Method . . . 3
2.3 Iterative Methods . . . 6
2.3.1 CG-Method . . . 7
2.3.2 CG on normal equations . . . 8
2.3.3 Initial guess and stopping criteria . . . 8
2.3.4 Preconditioning . . . 9

**3 Approach to parallelisation 10**
3.1 Overview over the `thermique` Code . . . 10
3.2 Analysis of the Code . . . 10
3.2.1 Timing by Instrumenting code . . . 10
3.2.2 Analysing the runflow using `tcov` . . . 11
3.2.3 Measuring the time spent in different functions using `prof` . . . 11
3.3 Mesh generation section . . . 11
3.4 Matrix assembly . . . 11

**4 Iterative methods and parallelisation 12**
4.1 Matrix storage . . . 12
4.1.1 Skyline-storage . . . 12
4.1.2 Line-based way of storage . . . 13
4.2 Inner products and vector sums . . . 14

**5 Results 15**
5.1 Preconditioner . . . 15
5.2 CG . . . 15
5.3 Matrix Assembly . . . 16

**6 Conclusion 16**
6.1 Using `OpenMP` . . . 18
6.2 Synchronisation in `OpenMP` . . . 18
6.2.1 Implicit barrier at the end of a `for`-loop . . . 19
6.2.2 False sharing . . . 19
6.2.3 Synchronisation between threads on the same processor . . . 19
6.2.4 `atomic` commands . . . 20

## 1 Introduction

The finite element method (FEM) is a computer-aided mathematical technique for obtaining approximate numerical solutions to the abstract equations of calculus that predict the response of physical systems subject to external influences. It is widely used for a wide range of problems and in many application areas.

`OpenMP` is a new standard allowing portable code for shared memory computers to be written. Shared memory computers are flexible and provide large computational power at relatively low costs. They are becoming more prevalent both in academia and in industry.

EPCC has recently installed a new shared memory machine. A large number of applications, including many finite element calculations, are expected to use this machine in the near future. Therefore, this project provided the opportunity to gain useful experience in parallelising a typical finite element code on this type of machine.

The linear algebra direct method of the finite element code is to be replaced with an iterative method which will give the code better performance for very large problem sizes.

## 2 Background

### 2.1 Shared Memory Computers and `OpenMP`

An increasing number of parallel machines make use of the shared memory architecture. In this type of platform each processor has access to a global memory store and processors communicate with one another by accessing the shared memory. This communication paradigm simplifies programming multiprocessor machines by removing the requirement for explicit communications.

Parallelisation of codes using shared memory is mainly carried out using compiler directives. Until recently each manufacturer provided their own set of machine specific compiler directives which, while they were similar in style and functionality, meant that codes were not trivially portable. The `OpenMP` [7, 8] standard was designed to address this issue and to provide a standard interface to shared memory achitectures. `OpenMP` is a specification for a set of compiler directives, library outines, and environment variables that can be used to specify shared memory parallelism in `Fortran` and `C/C++` programs.

EPPC has recently installed a new Sun Enterprise HPC 3500. It is an symmetric multi-processor (SMP) system with eight $400$ MHz UltraSPARC-II processors, $8$ Gbyte of shared memory and $54$ Gbyte disc space. All the work decribed subsequently was carried out using this machine. The nominal peak performance is 6.4 Gflops ($400$ MHz $\times$ $8$ processors $\times$ $2$ flops per cycle). However this is only an interim service. A new machine will be installed in the year 2000. This will consist of a cluster of R24s, each of which will be able to hold up to $24$ UltraSPARC-IIIs.

### 2.2 The Finite Element Method

The finite element method (FEM) is a flexible and powerful computational tool used widely in industry and universities to solve problems in many areas of engineering, science and applied mathematics [3, 6]. Its main advantage compared to other discretisation schemes is its ability to handle problems with very complex geometric features and the associated difficultly of encorporating their boundary conditions. In fact, the consequence of this procedure allows boundary conditions to be encorporated naturally without the need to resort to some special computational device at the boundaries.

Simliarly, as with other discretisation methods, the problem domain is partitioned into smaller regions known as elements. These touch without gaps or overlapping. Typical element shapes are triangles and squares for 2D problems while tetrahedrons and bricks are used for 3D problems. These basic elements can be distorted to fit complex geometries.

The essence of the FE method is quite simple. We wish to find the approximate solution to an equation that models the behaviour of the problem subject to certain boundary equations. The general two-dimensional boundary-value problem in residual form is

$$R_U(x,y) = f\left(\frac{\partial^2 U}{\partial x^2}(x,y)\,,\,\frac{\partial^2 U}{\partial y^2}(x,y)\,,U(x,y)\,,x\,,y\right) = 0$$

where $U$ is the exact solution.

Boundary conditions can be composed of $U$, gradients of $U$ or a mixture of both. We can now replace the exact solution $U$ with the approximate solution $\bar{U}$ with the result that the residual is no longer zero.

$$R_{\bar{U}}(x,y) = f\left(\frac{\partial^2 \bar{U}}{\partial x^2}(x,y)\,,\,\frac{\partial^2 \bar{U}}{\partial y^2}(x,y)\,,\bar{U}(x,y)\,,x\,,y\right) \overset{?}{\approx} 0$$

The approximate solution $\bar{U}$ across each element is composed of a linear combination of relatively simple known functions $\phi_j^{(e)}$

$$\bar{U}^{(e)}(x,y) = \sum_{j=1}^{n} a_j \phi_j^{(e)}(x,y)$$

where $\bar{U}^{(e)}$ is the approximate solution across the element, $\phi_j^{(e)}$ is a known function, commonly known as trial/shape function for each element node $j$ and $a_j$ is the contibution of shape function $j$ to the elemental solution. $n$ is the number element nodes. $n = 3$ for a triangular element.

The elemental solution for a triangular element using linear shape functions is illustrated in figure 1.

The FEM requires that $\bar{U}^{(e)}$ is an interpolating polynominal. The coefficents $a_j$ represent the value of the solution at each node $j$ of the element. This implies that the trial functions have the "Kronecker delta property". Thus

$$\phi_i^{(e)}(x_j) = \delta_{ij} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

The shape function associated with node 1 for the linear triangular element is shown in figure 2.

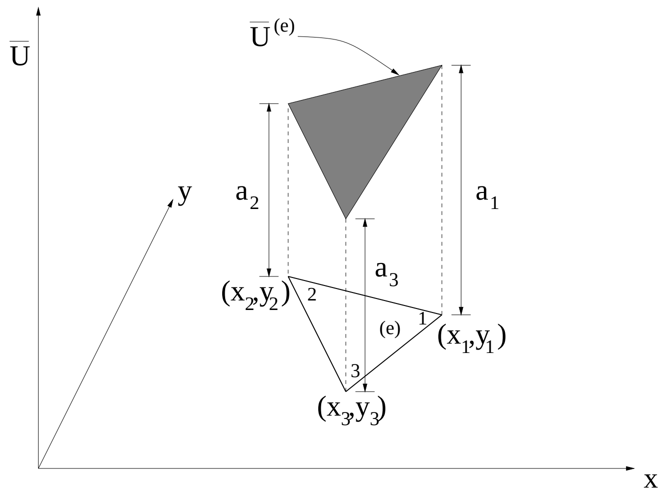


Figure 1: A linear approximation function $\bar{U}^{(e)}$ over one element

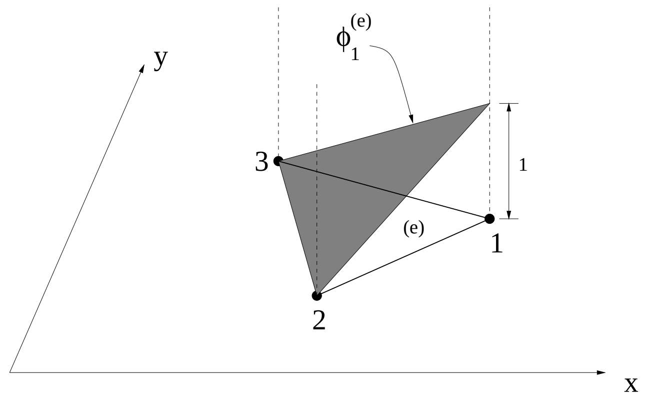


Figure 2: The linear trial function associated with node 1

So far there is an infinite set of $a_j$'s that could approximate the solution over the element. We need to find the numerical values that will make $R(x, y)$ as close to possible to zero across the element.

This can be achieved by introducing the condition that the residual $R_{\bar{U}}(x, y)$ has to be orthogonal to the trial fuctions $\phi_i^{(e)}$. This leads to the following equations for each element $^{(e)}$:

$$\iint^{(e)} R_{\bar{U}}(x, y)\ \phi_i^{(e)}(x, y)\ dx\, dy = 0 \qquad i = 1, 2, \ldots, n$$

where the double integral means integrate over the area of element $(e)$. This equation is also known as the "Galerkin criterion" and is a special case of the "Methods of Weighted Residuals" (MWR).

The definition of $\bar{U}$ is substituted into the above equation. Integration by parts leads to leads to a reduction in the order of all differential terms and allows boundary conditions to be introduced during the numerical solution.

The governing equations of a problem have now been transformed into a set of algebraic equations for each element which are then known as the elemental equations. Note, that the algebraic equations are the same for each triangular linear element for those problem. The elemental equations for each element used in the problem can be generated using the geometric and phyiscal properties prescribed.

In abbreviated matrix notation the element equations can be written as

$$[K]^{(e)} a^{(e)} = F^{(e)}$$

where $K^{(e)}$, $a^{(e)}$ and $F^{(e)}$ are the stiffness matrix, displacement vector and force vector respectively. This terminolgy derives from the FEM's origins in structural engineering. All boundary conditions are located in the force vector $F$.

It is obvious that unless one element is being used to model the whole problem domain then the elements have to be connected in some fashion to mimic the continous nature of the original problem. This is achieved by adding the elemental equations together using the fact that elements share nodes and hence the approximation is continuous as shown in figure 3. This process is known as assembly which produces a global stiffness matrix, a global displacement vector and a global force vector.

A global set of linear algebraic equations are formed. In FE problem applications this will produce a sparse positive definite stiffness matrix which can be efficiently solved using direct and iterative methods.

## 2.3 Iterative Methods

The term "iterative method" refers to a wide range of techniques that use successive approximation $x^{(0)}, x^{(1)}, \ldots$ to obtain a more accurate solution to the system of linear equations $Ax = b$ at each iteration.

As the matrix $A$ is only accessed via matrix-vector-multiplications, the sparsity of the matrix, and matrices typically resulting from in finite element analysis *are* sparse, can be exploited. Furthermore this provides a straightforward way to parallelise the calculations, as the computations for the different entries in a matrix-vector-product are independent.

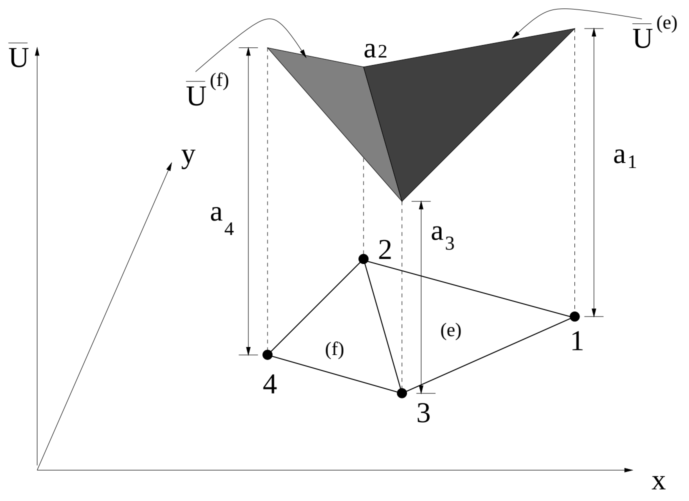


Figure 3: Two finite elements sharing the nodes $2$ and $3$

Although some iterative methods, such as the CG-Method, are guaranteed to find the exact solution after a fixed number of iterations a good aproximation is usually found after significantly less iterations. This is the case if the matrix has a certain form of regularity, e.g. several Eigenvalues being (almost) the same.

For large sparse systems of linear equations iterative methods are usually prefered to direct methods (such as Gauss-elimination),

The following iterative methods have been implemented. For an "easy to use"-overview over more iterative methods see [1].

### 2.3.1 CG-Method

The "conjugate gradient method" [10, 1] is a popular and easy to implement method. It can be used if the matrix $A$ is symmetric and positive definite. There are generalisations for indefinite matrices [9]. For a discussion of the convergence see [12]. For a history of this method with annotated bibliography see [4].

This method exploites the fact that the solution $\hat{x} = A^{-1}b$ of $Ax = b$ minimizes the function $f(x) = \frac{1}{2}x^T Ax - b^T + c$ (with $c$ an arbitrary scalar). In each step $f$ is minimised in some search direction $d^{(i)}$. The vectors $d^{(0)}, d^{(1)}, \ldots$ are $A$-orthogonal (i.e. $d^{(i)T} Ad^{(j)} = 0$ for all $i \neq j$). More precisely $d^{(i)}$ is defined to be the $A$-othogonal part of the $i$'th residu $r^{(i)} = b - Ax^{(i)} = -f'(x^{(i)})$, i.e. $d^{(i)}$ defined by being $A$-orthogonal to all previous $d^{(0)}, d^{(1)}, \ldots, d^{(i-1)}$ and by $d^{(i)} - r^{(i)}$ being a linear combination of the $d^{(0)}, d^{(1)}, \ldots, d^{(i-1)}$. It can be shown that for $i \geq 1$, $d^{(i)} - r^{(i)}$ in fact is a multiple of $d^{(i-1)}$, so that it is no necessay to store the $d^{(i)}$'s. Moreover $x^{(i+1)}$ can be shown to minimise $f$ in $x^{(0)} + span\{d^{(0)}, \ldots, d^{(i)}\} = x^{(0)} + span\{r^{(0)}, \ldots, r^{(i)}\}$. From this fact one can see that the method converges in at most $n$ steps, where $n$ is the dimension of the underlying vector space, i.e. the number of unknowns.

An algorithm in pseudo-code of the preconditioned (section 2.3.4) CG-Method can be found in figure 4. The (unpreconditioned) CG-Method is the special case of the precondition matrix $M$ being the identity-matrix. Note that the residual $r$ is updated using a recursive formula, which might lead to accumulation of floating-point inaccuracies. Therefore $r$ should be recomputed

Given the inputs $A$, $b$, an initial guess $x$, a preconditioner $M$, a maximum number of iterations $i_{max}$ and an error tolerance $\varepsilon < 1$.
$A$ is a matrix; $M^{-1}$ a (possibly implicitly defined) matrix; $b$, $d$, $r$, $q$ and $x$ are vectors; $\alpha$, $\beta$, $\delta_{old}$, $\delta_{new}$, $\delta_0$ and $\varepsilon$ are scalars; $i$, $i_{max}$ are integers.

```
$i$ :=$0$
$r$ :=$b - Ax$
$d$ :=$M^{-1}r$
$\delta_{new}$ :=$r^T d$
$\delta_0$ :=$\delta_{new}$

while $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
        $q$ :=$Ad$
        $\alpha$ :=$\frac{\delta_{new}}{d^T q}$
        $x$ :=$x + \alpha d$
        $r$ :=$r - \alpha q$
        $s$ :=$M^{-1}r$
        $\delta_{old}$ :=$\delta_{new}$
        $\delta_{new}$ :=$r^T s$
        $\beta$ :=$\frac{\delta_{new}}{\delta_{old}}$
        $d$ :=$s + \beta d$
        $i$ :=$i + 1$
end while
```

Figure 4: Pseudo-code for the CG-Method

regularly using its definition $r = b - Ax$. It should be definitely recomputed before the algorithm stops to avoid early termination with the real residual not yet being small enough.

### 2.3.2 CG on normal equations

If $A$ is not symmetric or not positive definite, but $A^T$ is available a simple method to solve $Ax = b$ is to solve the linear system $A^T Ax = A^T b$ using the CG-Method. Remember that $A^T A$ *is* symmetric and positive semidefinite. Moreover $A^T A$ is positive definite if $A$ is non-singular (i.e. if $A^{-1}$ does exist). $A^T A$ need not be calculated as multiplication of a vector with $A^T A$ can be calculated as two successive multiplications with $A$ and $A^T$. Therefore one step in this method involves *two* matrix-vector multiplications.

Because of this increased cost and the fact that $A^T A$ for *symmetric and positive definite* $A$ in general has less favourable properties than $A$, the CG-method should be prefered for symmetric *and positive definite* $A$.

### 2.3.3 Initial guess and stopping criteria

The CG-Method, as with most iterative methods, requires an initial guess to start the computations. A criterion also has to be selected to decide when there is a sufficiently accurate solution; at this point the computation is halted.

As a physical problem is being considered the values of $x$ and $b$ only have some meaning when read in certain units. Therefore, an initial guess and stopping criterion should be independent of rescaling, i.e. change of the units they are measured in.

As initial guess $x_{(0)} = 0$ was chosen and the stopping criteria is an improvement of the norm of the residual of a factor $\varepsilon$ compared with those of the initial guess. As the residual of the initial guess 0 is $-b$ this criterion has some reasonable relation to the scale of the actual problem. The value of the factor $\varepsilon$ determines the precision required and can be changed by the user. As, of course, not the norm but its square is the value used in the actual computation, $\varepsilon^2$ should be not less than the accuracy of the floating-point-arithmetic used. Values of $10^{-5} \dots 10^{-7}$ have been found appropiate.

When using a preconditioner (see section 2.3.4) the $M^{-1}$–norm is taken instead of the norm, where $M^{-1}$ is the preconditioning-factor. This assumes that $M^{-1}$ is symmetric and positive definite. If this is not the case the overhead of an additional inner product to calculate the euclidian norm should be accepted.

### 2.3.4 Preconditioning

As the convergence rate of iterative methods depends on spectral properties of the coefficient matrix $A$, one may attempt to transform the linear system into one that has the same solution, but more favourable spectral properties.

An idea would be to solve the system $M^{-1}Ax = M^{-1}b$, where $M$ is a nonsingular matrix that (in some sense) approximates $A$ but that multiplication with its inverse is easier to calculate ($M^{-1}$ need not be calculated).

For CG, however, the problem occurs that $M^{-1}A$ need not be symmetric and positive definite, even if $A$ and $M$ are. Therefore the problem $E^{-1}AE^{-T}\tilde{x} = E^{-1}b, \quad \tilde{x} = E^T x$ is solved with $E$ being some matrix with $EE^T = M$. Using careful variable substitutions [10] one can avoid calculating $E$ explicitly as the algorithm in figure 4 shows.

The preconditioners described below were implemented. A wider discussion of preconditioning and other preconditioners can be found in [2].

**Jacobi-preconditioner.** The "jacobi" or "diagnoal" preconditioner approximates the matrix $A$ by its diagonal entries, i.e. if $A = (a_{ij})_{ij}$, then $M = (a_{ij}\delta_{ij})_{ij}$ and $M^{-1} = \begin{pmatrix} a_{11}^{-1} & & \\ & \ddots & \\ & & a_{nn}^{-1} \end{pmatrix}$.
So all the information to multiply a vector with $M^{-1}$ can in principal be found looking at $A$. However, depending on the way of matrix-storage it might be easier to have the vector $(a_{ii}^{-1})_i$ stored separately. The same is true for many preconditioners that can be found in literature [1, 2]. Although, easy to implement and easy to parallelise this preconditioner only results in modest increases in convergence rate.

$ILU(0)$**-Preconditioner.** A much more effective preconditioner is the so-called $ILU(0)$–Preconditioner. If $A = L + D + U$, where $L$, $D$ and $U$ are the lower, diagonal and upper part of the matrix, then the preconditioner is defined [1] as $M = (D + L)D^{-1}(D + U)$. It is, $M^{-1} = (D+U)^{-1}D(D+L)^{-1}$. As $D+U$ and $D+L$ are both triagonal matrices, multiplying a vector $x$ with $M^{-1}$ can be done solving the *simple* linear system $My = x$ ($M^{-1}$ is never actually computed!). However this processes involes a lot of small loops that have to be executed sequentially; therefore parallelisation it difficult.

## 3 Approach to parallelisation

### 3.1 Overview over the `thermique` Code

The code, related to the "Connect Project" [5] and provided by the HPCI Rho consortium originally consisted of $14$ files of code, additional files for `common`–blocks and a `Makefile`; it was written in `Fortran 77`, only minor parts are written in `C`.

As the `guidef90` knows most extensions of `Fortran 77`, the code could relativly easy be adapted to the `E3500`. Some minor problems occured due to differences between the `Fortran 77` compiler originally used and the new `Fortran 90` standard, such as the implicit `saved` attribute supported by many `Fortran 77` compilers.

As the program is inteded to run as a batch job, the graphical output using X-Windows had to be removed. As the data necessary to generate the graphics is written into outputfiles the graphics could be generated by a separate program. The code added to implement the CG was written into separate files and the `C` programming language was choosen. When modifing or adding new Fortran code `implicit(none)` and `intent`–declarations of `Fortran 90` were used wherever possible.

The computational part of the code can be splitted into the $3$ main parts which are typical of a FE-code: the mesh-generation section, the matrix-assembly and the solver. This, however, does not precisely reflect the actual subroutine structure of the code, as preparations for the solving process were also calculated in the routine generating the matrix. During this project the mesh generation and the solver-routines were actually split into different programs to make reusing meshes more easy and to (in principle) be able to develop these parts independent of each other.

### 3.2 Analysis of the Code

As the code was provided by a third party the first step was to anlyse the code in order to know which parts of the code are the computational heavy ones and how these can be parallelised. Several methods have been used.

#### 3.2.1 Timing by Instrumenting code

As a simple device to find out which parts of the code are computationally the most important, timing commands where inserted directly at the relevant positions in the code. This allows timing with an acceptable overhead. As the real-time-clock was used, functions like `etime` and `dtime` measure the CPU-time, i.e. they time all threads together spend in a function, making these routines worthless to measure the effects of parallelisations.

To avoid changing the code permanently and potentially making it less useful for other users, timing commands are compiled conditionally depending on preprocessor[1] flags.

#### 3.2.2 Analysing the runflow using `tcov`

Parallelising loops that do not have enough iterations will result in slowdown due to the additional overhead rather than speedup due to parallelisation, it is important to get a feeling for the average number of iterations in a loop.

`tcov` provides a simple method for analysing the runflow of a program. Using special compiler flags (`-xprofile=tcov`)[2] the program outputs, after every run enough information to find out how often each line of code is executed. As `tcov` works only on serial code a compiler ignoring `!$OMP`–lines has to be used if a program is to be analysed that is already partially parallelised.

#### 3.2.3 Measuring the time spent in different functions using `prof`

`prof` is another profiling tool, instrumenting the code at compile time. It is supported by most compilers, including `guidef90`. Using the `-p` Option of the compiler, the program writes a file (usually `mon.out`) containing enough information to find out the time spent in each function call and the number of times that a function is called. Note, that the times reported do *not* include the time spent in called subroutines.

Using `prof` for parallel programs seemed to introduce a non-acceptable overhead and results difficult to interpret. Therefore using prof was only used on serial code.

### 3.3 Mesh generation section

Delaunay triangulation is used for mesh generation but due to project time constraints this was only partially parallelised. About half of the (serial) time spent in the mesh generation section is spent in other calculations, such as smoothing the mesh, testing whether two points are too close to one another and improving the numbering of the elements (to make sure that the non-zero elements of the resulting linear equation are as close to the diagonal as possible). Several of those tasks could easily be parallelised.

Although parallel smoothing of a mesh is possible (e.g. every processor smoothes a separate area of the mesh) this might result in slightly different meshes. Therefore, the smoothing has not been parallelised but could be a subject for further is investigations.

### 3.4 Matrix assembly

To generate the global stiffness matrix the elemental equations have to be calculated and then added at the correct position within the global matrix.

[1]Although the code is written in Fortran, the `cpp`, i.e. the “C Preprocessor”. This decision was taken because of the lack of a standard preprocessor for the Fortran language, whereas the `cpp` can be considered installed on every `Unix` system.

[2]According to the `man`–page there should be also an “old style” way of using `tcov` via the `-a` flag.

The calculation of the elemental equations are completely independent but the assembly is not. Therefore the program was changed so that first, all of the elemental equations are calculated and then they are added to the global matrix afterwards. The calculation of the elemental equations was parallelised by assigning each thread the same number of elements. The assembly of the global matrix was left serial.

Note, that as each entry of the global matrix is only sum of very few entries, a reduction over the whole matrix would be to expensive. If atomic had been implemented properly (see section 6.2.4) this might be an alternative. Another alternative might be using domain decomposition techniques [11] to be able to assign every thread a part of the global matrix that can be handled without looping through all the elements i.e. rearrange the order of the elements in a clever fashion to minimise dependency between threads.

## 4 Iterative methods and parallelisation

As the model is still an area of research it was not clear which numerical properties the actual system of linear equations will eventually have. Therefore the question which iterative method and which implementation might be the best used cannot yet be answered definitely. However, the way the matrix is stored might significantly affect the time needed to solve the system of linear equations and the scalability when using several processors (as only $8$ processors could be used at the moment this could not be tested completely either).

Therefore, to be as flexible as possible a header-file was used as an interface to matrix operations in order to develop matrix-storage and iterative methods independently.

### 4.1 Matrix storage

To investigate code parallelisation with `OpenMP` two ways of storing sparse matrices have been implemented. There are more ways of storing a matrix than those explained here. For an overview of different ways of storing sparse matrices see [1].

#### 4.1.1 Skyline-storage

In the code the matrix was stored in “skyline” format. Therefore, choosing this form of storage has the advantage that no transformation of the matrix has to be calculated. On the other hand parallelisation of matrix-vector-products based on this form of matrix storage is expected not to scale that good for large numbers of processors (which could not be tested, as only $8$ processors have been available).

Skyline storage exploits the fact that the matrix is symmetric and therefore only the upper triangular part has to be stored. Moreover, using a good FE numbering scheme the matrix has all its non-zero coefficients near the main diagonal. Therefore, for each column only the elements from the diagonal up to the first non-zero element in this column are stored. All these “skyline-towers” are then stored in a single array and a second array is used to store pointers to diagonal entries in the original matrix. Figure 5 illustrates the prinicipale of skyline-storage.

Calculating the rows of the stored information is not that easy. The parallelisation was achieved by assigning each thread the same number of skyline-towers. As illustrated in figure 6, this

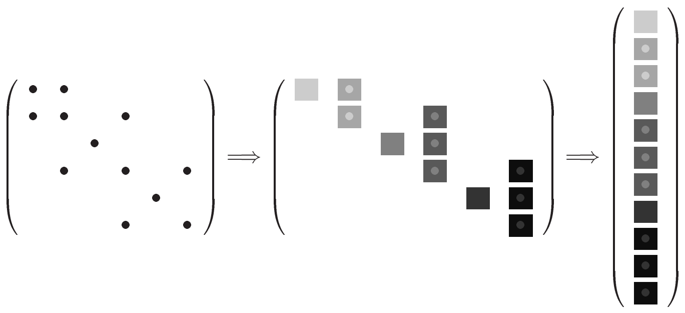


Figure 5: Skyline-Storage of a sparse symmetric matrix

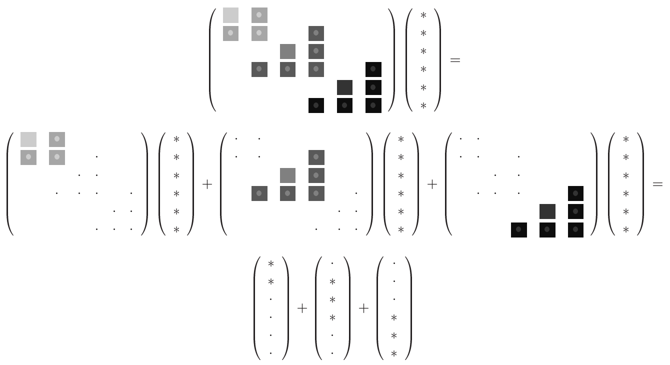


Figure 6: Parallel multiplication of a matrix in skyline storage and a vector

produces partially overlapping results that have to be added. For simplicity each thread had its own array the size of the complete vector and the results over the whole vector have been added, introducing an overhead similar to a serial part of code[3]. Further optimisation is possible by adding only the regions where non-zero entries results of each thread will fall. These regions can be determined by finding out the height of the largest skyline-tower[4]. However the time to implement this improvement was not available.

It should be noted that it is not a possible to get reasonable speedup when synchronising the access to the overlapping regions using `atomic` statements, see section 6.2.4 for more details.

### 4.1.2 Line-based way of storage

Also, another way of storing the main matrix was implemented. The overhead of this transformation is ignored in future discussions.

To make the CG code more general no use was made of the fact, that the matrix was symmetric, but each non-diagonal entry was in fact stored twice. This allowed to access all the relevant

[3]Although being executed in parallel the reduction loop is considered a sequential one, as the amount of work for this loop is proportional to the number of threads executing this function. The loop initialising all the result-vectors with 0 is really a serial loop, as *every* thread has to execute this loop on its *own* copy of data. Finally it should be noted that the overhead can be even worse if the relevant data does not fit in cache, as on most shared memory machines all main-memory accesses are serialised.

[4]As the matrix does not change during the CG computation this value has to be determined only *once* and can be used for *all* the multiplication within the CG. Moreover, as in the program the CG has to be applied to lots of different matrices all having the same shape, this value can even be reused more often.

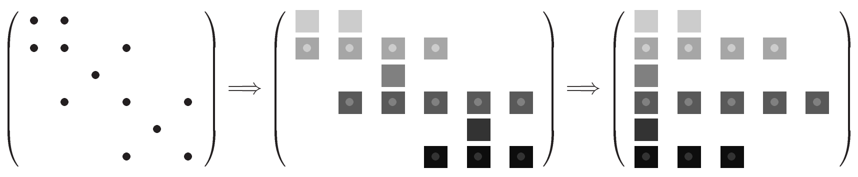


Figure 7: Line-based storage of a sparse symmetric matrix

entries in a line of the matrix in a linear way. To benefit from the fact that the matrix is sparse, only the non-zero entries where stored. The complete data structure therefore looked as follows:

- The entries of the matrix where stored in a one-dimensional array of one-dimensional arrays, not necessarly of the same length. Each entry of this array represents a line of the matrix storing the entries beginning with the first non-zero entrie ending with the last non-zero entry.
- A vector of the indices of the columns containing the first non-zero entry of each line, asuming the first column is column number $0$.
- A vector of the columns *following* the columns containing the last non-zero entry of each line.

Obviously the way of numbering the indices in the vectors discribing the shape of the matrix was choosen to best fit `C` for-loops and the way `C` numbers the entries of arrays. Figure 7 illustrates this storage principle.

Parallelising matrix-vector multiplication with a matrix stored in this format is done in the obvious way by assigning each thread a set of lines it is responsible for. Therefore, multiplication involves no communication at all. To reduce communication to a minimum the same static scheduling scheme was choosen for all vector and matrix-vector operations [5].

An analysis of the code shows that only once per iteration a vector has to be passed between the different threads. Moreover, this communication can be reduced to some form of nearest neighbours communication if using a static block schedule for distributing the lines to the different threads. Therefore this form of parallelisation can also be used on distributed memory machines and simply be rewritten using message passing[6].

## 4.2 Inner products and vector sums

Inner products $x^T y$ and vector sums $x + y$ are obvious to parallelise as all the calculations ($x_i \cdot y_i$ resp. $x_i + y_i$) are completely independent. Of course, a reduction is necessary for the inner product.

When parallelising these utility functions one should try to reduce communication by assigning each thread the same lines of the matrix every time. Therefore the way of distributing the work

[5]This assumes, that all threads keep living all the time, which is possible as `OpenMP` supports orphaned directives.

[6]This, would include an extra step to distribute the data and find out, with which neighbour communication is necessary, as this depends on the exact shape of the matrix. The last step is not necessary in `OpenMP`, as communication is asymmetric: every thread only needs to know, which data it needs; it is not necessary to know who needs the data a thread has just calculated.

| Number of elements | 159 756 | 24 247 | 10 088 | 2 406 |
|---|---|---|---|---|
| No precond. | 528 | 211 | 140 | 66 |
| Jacobi | 509 | 201 | 132 | 62 |
| $ILU(0)$ | 189 | 79 | 53 | 26 |

Table 1: The number of iterations needed for the CG-Method, depending on the mesh size and the preconditioner choosen

in this functions has to be choosen according to the way the matrix-vector multiplications are parallelised[7].

To fullfill this requirement all parallelised loops over the rows of a vector/matrix use a preprocessor macro that expands to the same schedule every time. Note, that this of course, will work with static schedules only.

## 5 Results

The serial code using skyline storage was run for different mesh sizes with the two preconditioners. Timings are for the stationary solution since this will require more CG iterations to find a solution. For all timings, timers as discribed in section 3.2.1 have been used.

The parallel version was run with different combinations of threads, mesh sizes and matrix storage methods. Timings were made for the main compuational sections of the code i.e. the matrix assembly and CG solver routines.

### 5.1 Preconditioner

Table 1 slows the the number of iterations required for convergence for the two preconditioners and with none. Skyline storage is used. It can be observed that the Jacobi preconditioning produces a very small reduction in the number of iterations while the $ILU(0)$ preconditioner reduces the number of iterations by an average of $63\%$.

However, comparison of the overall CPU time required for convergence is significantly different. The Jacobi preconditioner adds little extra computation and typically the overall time needed is reduced by $3\%$. By contrast, the $ILU(0)$ preconditioner produces significant extra computation which uses more time than that saved by the reduction in the iterations. Moreover this preconditioner has not been parallelised, so the total time on $8$ processors could be up to $6.5$ times greater than that with no preconditioner.

### 5.2 CG

Figure 8 show the speedup of CG using skyline-storage and line-based storage on a mesh with $159756$ elements. It can be observed that both show linear speedup using up four processors. For more processors line-based storage indicates better scaling potential.

[7]Of course this applies only, if the matrix-vector multiplication doesn't involve reduction over the resulting vector, as it does for example when the skyline storage is used. In this case the parallelisation for this part can be choosen more or less arbitrary.

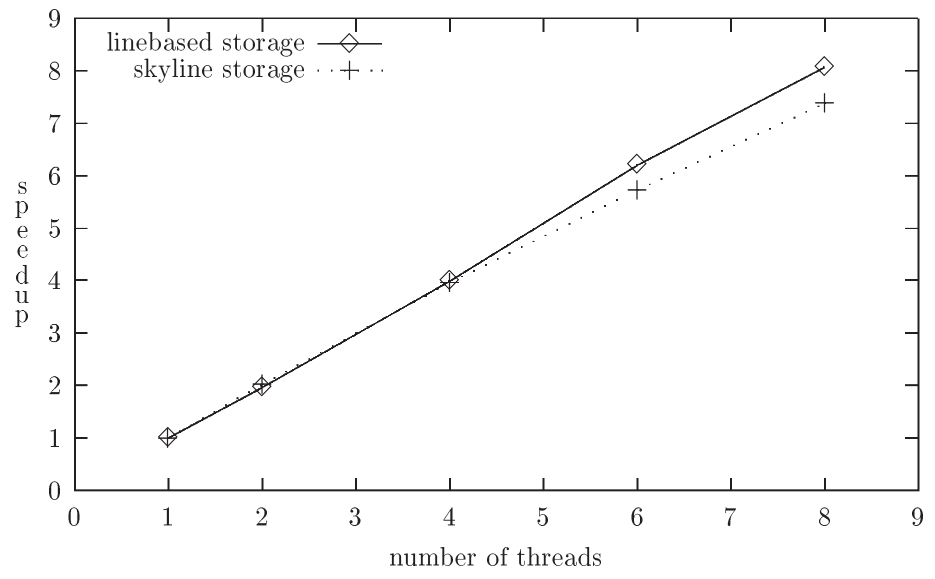


Figure 8: Speedup of CG using skyline-storage and line-based storage using Jacobi preconditioning

Figure 9 shows the time spent in the CG(skyline storage) depending on the number of threads using different mesh sizes. It can be seen that the algorithm scales well independent of the problem size.

### 5.3 Matrix Assembly

Figure 10 shows the speedup for calculating the elemental equations for 159756 elements producing 23723788 “entries”[8] in the skyline matrix. The speedup is not perfect although the calculations are completely independent. There is no load imbalance and there are no synchronisations required. There may be several reasons for this behaviour. The excessive memory accesses might have introduced system synchronisations or the system is interfering somehow and the computation was not obtaining all the processor time. However, this is an area for further investigation. Furthermore, it should be noted, that this is not the most important problem with the chosen method of parallelisation. It can be seen from figure 11 that the time for the non-parallelised part becomes more relevant as the number of processors increases. This figure shows the total time spent in this subroutine (upper line) and the time spend in the unparallelised part adding up the elemental equations (lower line).

## 6 Conclusion

As `OpenMP` is a relativly new standard (1997) experiences using this approach to parallelisation might be useful for other programmers. This section gives an overview of experiences gained during this project that could be considered interesting for others.

[8]Including the zero-entries over the diagonal with non-zero entries above it

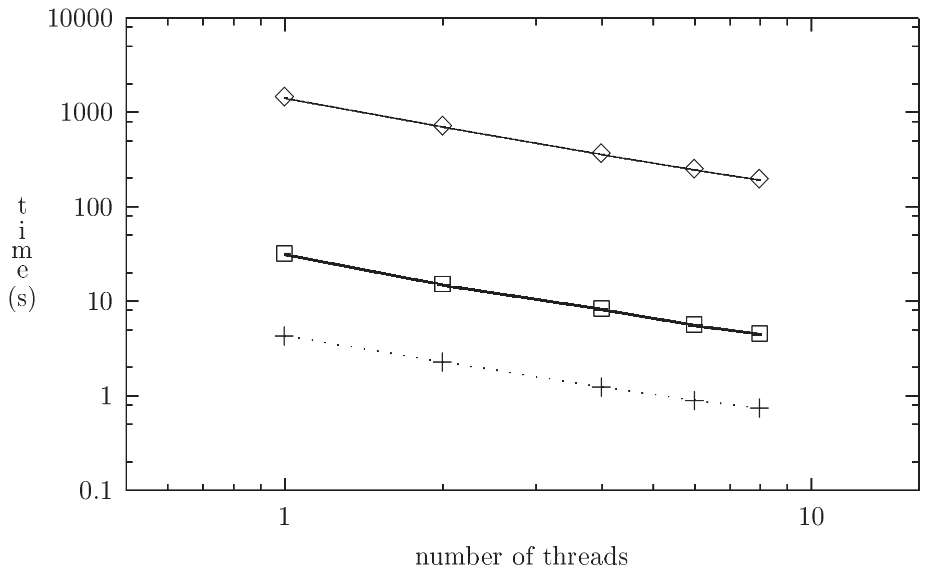


Figure 9: Time spent in CG(skyline storage) depending on the number of threads for different mesh sizes ($159756$, $24247$ and $10088$ elements)

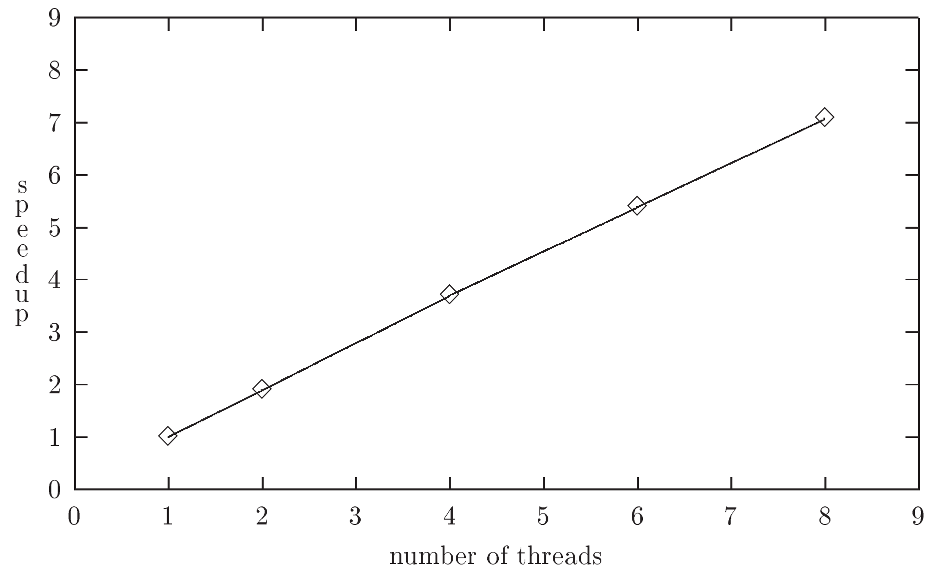


Figure 10: Speedup for calculating the elemental equations

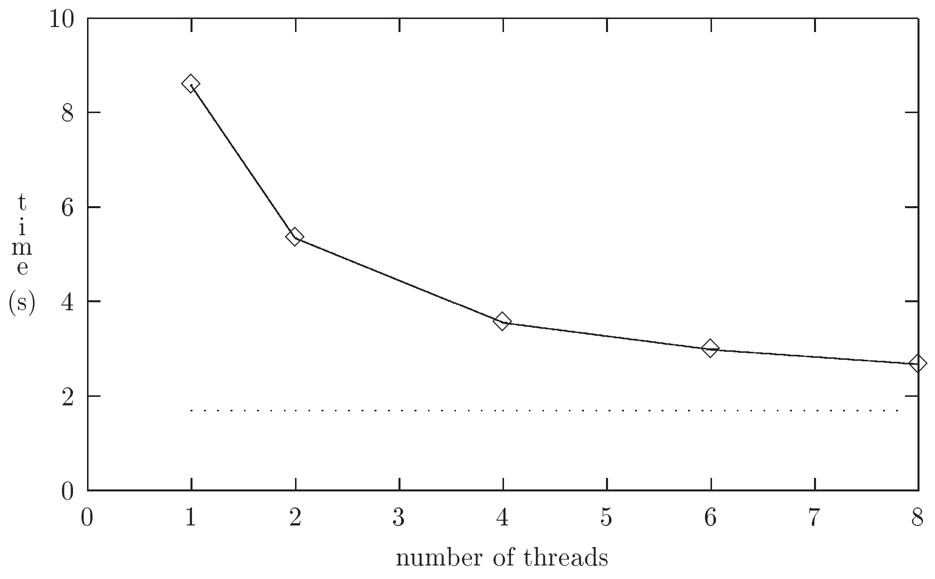


Figure 11: Time in matrix assembly as sum of time calculating the elemental equations and time spent adding coefficients

## 6.1 Using `OpenMP`

As `OpenMP` provides an easy way to parallelise the most common structures in typical numerical applications (such as `do`–loops), the main work is not to code, but to analyse the structure of the code. This might be much easier for someone familiar with his/her code, so it is suggested that while developing the serial code one should already start adding more or less formal comments about the parallel structure of certain loops. As serial compilers ignore the `OpenMP` directives these comments would not effect serial code development.

However, programs cannot always be parallelised by just adding directives. Often the structure of the program has to be changed (see sections 3.4 and 4.1.1 for examples). When writing new code for `OpenMP` this should be considered from the start.

`OpenMP` acts as a preprocessor on the code and debugging parallel code is quiet difficult as the preprocessed code looks different from the original code or is not available (as in the case of the `guidec`). When using `Fortran` it should be noticed that `private` variables are unitialised, while many `Fortran` compiler support initialising all variables with $0$. As variables that are already initialised in the program are not initialised when being declared `private` on opening a new `parallel` construct, running the code serial and parallel with $1$ thread is *not* equivialent.

## 6.2 Synchronisation in `OpenMP`

In the parallel model based on message passing (such as `pvm`, `MPI`, ...) communication is the main overhead introduced. This overhead in made quiet explicit due to the calls of the communication functions. In `OpenMP` [7, 8] communication is done implicitly.

However `OpenMP` *does* introduce an overhead. The best way to think of is thinking in synchronisation events as expensive operations. As the typical communication in `OpenMP` is of the form “one thread writes to a shared variable; barrier; the other thread reads the shared variable” the overhead is still related to the communication between processes. Therefore, the number of synchronisation events is a good measure for the overhead introduced.

### 6.2.1 Implicit barrier at the end of a `for`-loop

Most of the barriers in a typical `OpenMP`-program are the implicit barriers at the end of a `for`-loop. Therefore, before parallelising the loop one should ask oneself whether the overhead of an additional barrier is won back by the parallelisation. As a rule of thumb, it was found that the effort of an additional barrier is only worth the effort, if the loop contains the amount of work equivivalent to at least several hundred multiplications. Here one should remember that all main-memory accesses are serial on most shared memory machines. Therefore, parallelising copy/initialisation/. . . –operations is only worth it if a valid copy is already in cache or the cached value can be used immediatly afterwards.

When writing your own code one normally has an intuition of number of iterations spent in a loop and one can try to make the largest loop the outermost. Parallelising someone elses code is more difficult as one has to analyse how big the loops are and whether they can be parallelised or not. For analysing the typical size of a loop `tcov` (see section 3.2.2) was found to be a useful tool.

### 6.2.2 False sharing

As the units for cache coherency are usually larger than typical array entries like floating-point numbers, accessing different entries of the same array that are close together can force the system to keep these accesses coherent, even if in theory this is not necessary.

The best way to avoid this is using sufficently large chunck-sizes with the different schedules. However, one should be aware of the fact that there is no proper way to force allignment of arrays according to the synchronisation units of cache. Therefore false sharing effects could be observed even with chunk sizes larger than the actual synchronisation units. Block schedules have been found the best way to minimise these problems.

### 6.2.3 Synchronisation between threads on the same processor

If there is more than one thread on a processor the threads are scheduled by the operating systems. As the scheduling intervals are quite long compared with the actual cost of a barrier event and the system seems to use spin-lock mechanisms this can result in an unexpected high overhead.

Figure 12 shows the total time for solving a linear equations using the CG-Method on a simple set of data versus the number of threads used. One can clearly see the increase in the CPU time once the number of threads becomes greater than $8$, which is the number of processors on the machine. Using $16$ threads one again would expect perfect load balance.

The same effect is likely to occur when the machine has to be shared with other processes. Therefore testing the effects of parallelisation should only be done using a machine exclusively

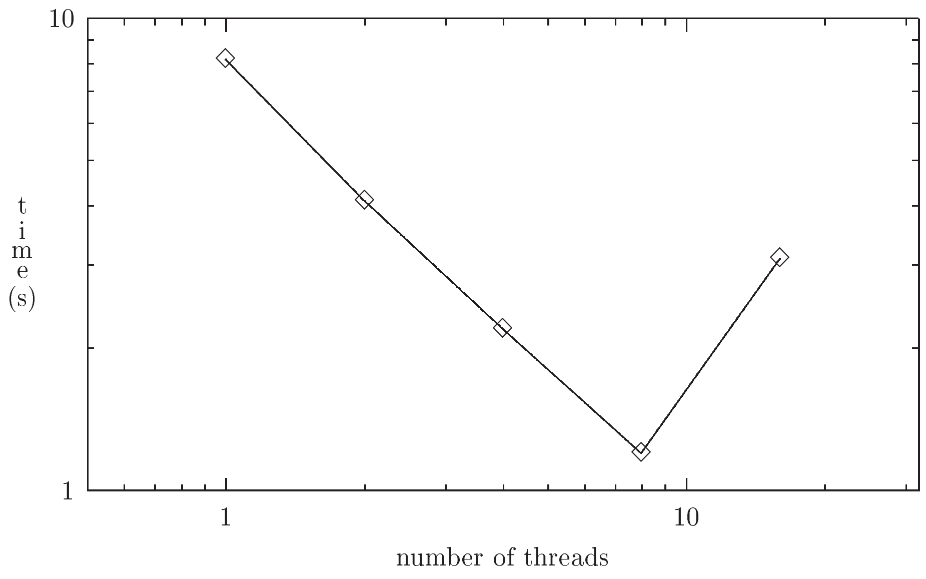


Figure 12: Time versus number of threads of a parallel implementation of the CG-Method on a machine with 8 processors

and when actually running the parallised code one should make sure that the total number of threads on the machine does not exceed the total number of processors.

### 6.2.4 `atomic` commands

When applied to a elements of an array they seem to lock the whole array and not only the relevant element — or even worse it seems to follow the suggestion of the specification and replaced it by a `critical` command. This effect could be seen when the access to different entries of a global array (see, e.g. section 4.1.1) did not scale and even became slower due to the increasing overhead. Even with only 1 thread the overhead was *not* aceptable.

It should be noticed that in this point the specification [7, 8] is inconsistent, as on the one hand it says that `atomic` guarantees the update of the variable to be atomic and on the otherhand it says that it is a correct implementation to replace it by a `critical`-section[9].

## References

[1] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roland Pozo, Charles Romine, and Henk van der Vorst. Templates for the solution of linear systems: Building blocks for iterative methods. Technical report, SIAM, 1994.

[9]Consider two threads, one doing the statement `a *= 2` atomic and the other thread doing the statement `a = 2`, which is atomic by definition (`OpenMP` guarantees that every memory access is atomic) in a situation where `a` has the value 3. If the statement `a *=2` is executed atomicly than `a` is guaranted to have either the value 2 or 4. If, however the `atomic` is replaced by a synchronised statement, the other thread would be allowed to write during the update of `a`, so in this case 6 would be a legal value as well.

[2] Michele Benzi. Preconditioning for sparse linear systems, October 1997. Available at `http://www.c3.lanl.gov/˜benzi/benzi.html`.

[3] Davis S Burnett. Finite element analysis from concepts to apllications.

[4] Gene H. Golub and Dianne P O'Leary. Some history of the conjugated gradient and lanczos algorithms. *SIAM Review*, 31(1):50–102, March 1989.

[5] Mireille Huc, Ian Main, Sergei Zatsepin, James Smith, Tom Leonard, Orestis Papasioulotis, David Henty, and Michael Bowers. Connect project. non-linear dynamics in fluid-rock systems. Technical report, Dept. of Geology and Geophysics, University of Edinburgh, 1998.

[6] Claes Johnson. Numerical solution of partial differential equations by the finite element method.

[7] OpenMP Architecture Review Board. *OpenMP Fortran Application Programm Interface*, October 1997. Available at `http://www.openmp.org`.

[8] OpenMP Architecture Review Board. *OpenMP C and C++ Application Programm Interface*, October 1998. Available at `http://www.openmp.org`.

[9] C. C. Paige and M. A. Saunders. Solution of sparse idefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12:617–629, September 1975.

[10] Jonatan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, August 1994. Available at `http://www.cs.cmu.edu/˜jrs/jrspapers.html`.

[11] Jonathan Richard Shewchuk and Omar Ghattas. A Compiler for Parallel Finite Element Methods with Domain-Decomposed Unstructured Meshes. In David E. Keyes and Jinchao Xu, editors, *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and Engineering Computing*, volume 180 of *Contemporary Mathematics*, pages 445–450. American Mathematical Society, October 1993.

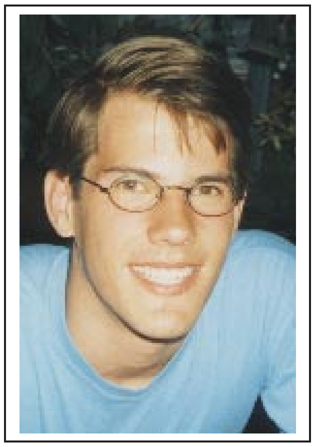
[12] A. van der Sluis and H. A. van der Vorst. The rate of convergence of conjugate gradients. *Numer. Math.*, 48:543–560, 1986.

## List of Tables

1 The number of iterations needed for the CG-Method, depending on the mesh size and the preconditioner choosen . . . . . . . . . . . . . . . . . . . . . . . . 15

## List of Figures

1 A linear approximation function $\bar{U}^{(e)}$ over one element . . . . . . . . . . . . . 5
2 The linear trial function associated with node $1$ . . . . . . . . . . . . . . . . . . 5
3 Two finite elements sharing the nodes $2$ and $3$ . . . . . . . . . . . . . . . . . . . 7
4 Pseudo-code for the CG-Method . . . . . . . . . . . . . . . . . . . . . . . . . . 8
5 Skyline-Storage of a sparse symmetric matrix . . . . . . . . . . . . . . . . . . . 13
6 Parallel multiplication of a matrix in skyline storage and a vector . . . . . . . . 13
7 Line-based storage of a sparse symmetric matrix . . . . . . . . . . . . . . . . . . 14
8 Speedup of CG using skyline-storage and line-based storage using Jacobi preconditioning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
9 Time spent in CG(skyline storage) depending on the number of threads for different mesh sizes ($159756$, $24247$ and $10088$ elements) . . . . . . . . . . . . . 17
10 Speedup for calculating the elemental equations . . . . . . . . . . . . . . . . . 17
11 Time in matrix assembly as sum of time calculating the elemental equations and time spent adding coefficients . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
12 Time versus number of threads of a parallel implementation of the CG-Method on a machine with 8 processors . . . . . . . . . . . . . . . . . . . . . . . . . . . 20



Klaus Aehlig participated on a $10$ week “Summer Scholarship Programme” at the “Edinburgh Parallel Computing Center” while writing this report. He participated in the programme after studing $3$ years of mathematics and computer science at the Ludwig Maximilians Universität in Munich.

The project was supervised by Michael W Bowers whos support is gratefully aknowledged.