The Solution of Systems of Linear Equations using the Conjugate Gradient Method on the Parallel MUSIC-System

Jean-Guy Schneider, Edgar F.A. Lederer, Peter Schwab

Abstract

The solution of large sparse systems of linear equations is one of the most computationally intensive parts of finite element simulations. In order to solve these systems of linear equations, we have implemented a parallel conjugate gradient solver on the SPMD-programmable MUSIC-system. We outline the conjugate gradient method, give a formal specification in Maple, and describe a data-parallel program. We illustrate how the number of processors influences the speed of convergence due to different data distributions and the non-associativity of the floating point addition. We investigate the speed of convergence of the conjugate gradient method for different floating point precisions (32, 44, 64, and 128 bit) and various finite element models (linear beams, human spine segments). The results show that it is more important to concentrate on appropriate numerical methods depending on the finite element models considered than on the floating point precision used. Finally, we give the results of our speedup measurements.

Keywords: finite element simulation, systems of linear equations, sparse matrix, conjugate gradient method, MUSIC-system, Maple, floating point precision, parallel summation.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors); G.1.0 [Numerical Analysis]: General; G.1.3 [Numerical Analysis]: Numerical Linear Algebra; G.1.8 [Numerical Analysis]: Partial Differential Equations; I.1.3 [Algebraic Manipulation]: Languages and Systems.

Authors' address: Institute for Computer Science and Applied Mathematics (IAM), University of Berne, Länggassstrasse 51, CH-3012 Bern, Switzerland; e-mail: {schneidr,lederer,pschwab}@iam.unibe.ch.

1 Introduction

For the modelling and simulation of the human spine in the context of the SPINET project*, a finite element program for the SPMD-programmable MUSIC-system has been

*The SPINET research project was funded by the Swiss National Science Foundation, grant No. SPP-IF 5003-034405.

developed. As basis, a finite element program developed at the Institute of Aeronautics and Applied Mechanics at the Warsaw University of Technology has been used. Some parts of this sequential program were ported to the parallel MUSIC-system, while other parts were redesigned and newly programmed. Using this program, it was also tested whether the MUSIC-system is suitable for finite element simulations in general, and spine simulations in particular.

As the most computationally intensive parts of finite element simulations, assembly as well as solution of large sparse systems of linear equations have been identified. Hence, it is obvious to use parallel algorithms for solving these problems. Since the "Frontal Solver" integrated in the Warsaw program is difficult to parallelize, another method has been chosen. The conjugate gradient method is used, since it exploits the fact that the systems of linear equations of finite element simulations are symmetric and positive definite. In addition, it is suitable for the implementation on SPMD-programmable computer systems.

In this report we show, how a conjugate gradient solver has been specified in Maple, a tool for symbolic computation, and implemented on the MUSIC-system, a parallel computer developed at the ETH Zurich. We illustrate the influence of the number of processors on the parallel computation of sums which in terms influences the speed of convergence. We also discuss the influence of various floating point precisions in finite element simulations and give the results of our speedup measurements. We conclude with remarks about the usability of the conjugate gradient method and the MUSIC-system for spine and other finite element simulations.

2 The Conjugate Gradient Method

We consider a system $A \cdot x = b$ of linear equations. Throughout this work, we will assume that the matrix A is symmetric and positive definite. In contrast to the method of Gauss [Sch88] or the "Frontal Solver" [Iro70, BH82], which is integrated in the program of Warsaw, the conjugate gradient method is an iterative method. It is motivated by the desire to accelerate the speed of convergence of so-called stationary iterative methods [BBC+93] for the particular class of symmetric and positive definite systems of linear equations. The idea of the method is to find the minimum of a particular function, which corresponds to the solution of the system of linear equations. The method is outlined in the following section; for detailed information refer to [SRS68, Sch88, BT89].

Definitions

As a reminder, some definitions are listed, which will be used throughout this report.

- An $n \times n$ matrix A is *symmetric*, if $a_{i,j} = a_{j,i}$ $(1 \leq i, j \leq n)$.
- A real $n \times n$ matrix A is *positive definite*, if all vectors $x \in \mathbb{R}^n, x \neq 0$, satisfy $x \cdot A \cdot x > 0$.
- The diagonal matrix D of a $n \times n$ matrix A has the diagonal elements of A on its diagonal; all other elements are equal to zero: $d_{i,i} = a_{i,i}$, $d_{i,j} = 0 \; \forall \; i \neq j$ $(1 \leq i, j \leq$

n).

- A real $n \times n$ matrix A is *regular*, if the determinant $\det(A)$ is different from zero.
- Two vectors $x, y \in \mathbb{R}^n$ are *A-conjugate*, if $x \cdot A \cdot y = 0$.
- For every point of the n-dimensional space, the gradient ∇F of a function $F : \mathbb{R}^n \to \mathbb{R}$ defines the direction of steepest descent:

$$\nabla F := \left(\frac{\partial F}{\partial x_1}, \frac{\partial F}{\partial x_2}, \ldots, \frac{\partial F}{\partial x_n} \right) \tag{1}$$

Problem

We consider $n \in \mathbb{N}$, a real $n \times n$ symmetric and positive definite matrix A, a vector $b \in \mathbb{R}^n$, and are interested in the solution of the system $A \cdot x = b$ of linear equations.

Cost function

A cost function $F : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$F(x) = \frac{1}{2}\, x^T \cdot A \cdot x - b^T \cdot x. \tag{2}$$

In [BT89] the following two propositions are proven:

1. There is exactly one minimum of F.
2. x_m is the minimum of F, if and only if

$$\nabla F(x_m) = 0. \tag{3}$$

Since

$$\nabla F(x) = \nabla \left(\frac{1}{2}\, x^T \cdot A \cdot x - b^T \cdot x \right) = A \cdot x - b, \tag{4}$$

the minimum x_m of F is equal to $A^{-1} \cdot b$, which corresponds to the solution of the system of linear equations.

General iteration form

Given any vector $x_0 \in \mathbb{R}^n$, an iteration of the form

$$\begin{aligned} x(0) &= x_0 \\ x(t+1) &= x(t) + \gamma(t) s(t) \qquad (t = 0, 1, 2, \ldots) \end{aligned} \tag{5}$$

is used to find the minimum of the function F, where $t \in \mathbb{N}$ is an index of the iteration, $s(t) \in \mathbb{R}^n$ a direction of update, and $\gamma(t) \in \mathbb{R}$ a scalar stepsize. $\gamma(t)$ is chosen in a way that $F(x(t+1))$ is minimized on $x(t) + \Gamma s(t)$ ($\Gamma \in \mathbb{R}$).

The distinguishing feature ot this method is the choice of the directions of update $s(t)$; they are chosen so that they are mutually A-conjugate:

$$s(t)^T \cdot A \cdot s(r) = 0 \qquad (t, r \in \mathbb{N},\ t \neq r). \tag{6}$$

In order to simplify the notation, $g(t)$ is defined as

$$g(t) := \nabla F(x(t)) = A \cdot x(t) - b \tag{7}$$

Some important consequences of conjugacy are the following [SRS68, BT89]:

1. The directions of update $s(0), s(1), \ldots, s(t)$ are linearly independent.
2. The gradient vectors $g(0), g(1), \ldots, g(t)$ are mutually orthogonal.
3. a) The vectors $x(0), x(1), \ldots, x(t)$ satisfy $F(x(k+1)) \leq F(x(k))$ for $k \leq t$, and
   b) there exists an $m \in \{0, 1, \ldots, n\}$ with $F(x(m)) = 0$.

Another feature of this method is the fact that after each iteration step, the dimension of the space for the solution is reduced by one. Because of 3.b), the method terminates after at most n iteration steps with $g(m) = 0$. Therefore, the conjugate gradient method is called a deterministic method, in contrast to most other stationary iterative methods.

Recursive equations

It is possible to describe the conjugate gradient method by five recursive equations, which are given without derivation.

The directions of update $s(t)$ are generate by the following formula:

$$s(t) = -g(t) + \sum_{i=0}^{t-1} c_i s(i) \qquad (c_i \in \mathbb{R},\ t = 0, 1, 2, \ldots). \tag{8}$$

Therefore, the first direction of update results in $s(0) = -g(0)$.

Since all directions of update are mutually A-conjugate, the equation (8) is simplified to

$$\begin{aligned} s(0) &= -g(0) \\ s(t+1) &= -g(t+1) + \beta(t+1)s(t) \qquad (t = 0, 1, 2, \ldots) \end{aligned} \tag{9}$$

with

$$\beta(t+1) = \frac{g(t+1)^T \cdot g(t+1)}{g(t)^T \cdot g(t)} \qquad (t = 0, 1, 2, \ldots). \tag{10}$$

Finally, $\gamma(t)$ has to be specified in order to minimize $F(x(t+1))$ on $x(t) + \Gamma s(t)$ ($\Gamma \in \mathbb{R}$). This leads to

$$\gamma(t) = -\frac{s(t)^T \cdot g(t)}{s(t)^T \cdot A \cdot s(t)} \qquad (t = 0, 1, 2, \ldots). \tag{11}$$

In order to calculate the minimum of F, the equations (5), (7), (9), (10), and (11) are used. It is noteworthy that it is not necessary to explicitly calculate the function F.

Numerical behaviour

Due to numerical reasons, the gradient vectors $g(t)$ in general are not exactly mutually orthogonal. Basically, this discrepancy from the theory is not that important, since the iterative process can continue after n iteration steps. In general, the iterative process is stopped after l iteration steps, if the Euklidian norm of $g(l)$ is smaller than a predefined threshold. Particularly for the solution of large sparse systems of linear equations of finite element simulations, usually considerably less than n iterations steps are needed [Sch91]. Due to our results listed in Section 6, the last statement strongly depends on the numerical conditioning of the given problem, and less on the floating point precision of the architecture used.

Preconditioning

In order to improve the speed of convergence, it is possible to precondition iterative methods. To do so, the linear equation system $A \cdot x = b$ is multiplied by a regular matrix P (preconditioning matrix), in order to get a new system $P \cdot A \cdot x = P \cdot b$. With an appropriate choice of P, it is possible to solve the system of linear equations faster and avoid numerical instabilities [BBC+93].

In order to precondition the conjugate gradient method, a regular and symmetric matrix $P = H^2$ is used. This has the following influences on the five recursive equations [BT89]:

$$s(0) = -H \cdot g(0) \tag{12}$$

$$s(t+1) = -H \cdot g(t+1) + \beta(t+1) \cdot s(t) \qquad (t = 0, 1, 2, \ldots) \tag{13}$$

$$\beta(t+1) = \frac{g(t+1)^T \cdot H \cdot g(t+1)}{g(t)^T \cdot H \cdot g(t)} \qquad (t = 0, 1, 2, \ldots). \tag{14}$$

The other equations remain unchanged. We see that it is not necessary to calculate the matrix product $P \cdot A$. For examples of preconditioning matrices, refer to [BBC+93, Bar89]

3 Implementation of the Algorithm

In order to implement a conjugate gradient solver, the following three step method has been applied:

1. Specification of the recursive equations with Maple, a tool for symbolic computation [CGG+91].
2. Sequential implementation of the specification of step (1) in the programming language C [Amm91].

3. Porting of the sequential C program of step (2) to the MUSIC-system [Bäu92] (refer to Section 5).

After each step, the programs were checked with selected examples. In order to get further references, all the example equation systems were also solved with Mathematica, another tool for symbolic computation [Wol88].

For the Maple specification of the equations listed in Section 2, predefined data structures for matrices and vectors have been used (refer to Figure 1). This procedure has the advantage that it is possible to precisely state the given problem, without taking care of implementation details. Since the specification in Maple is also executable, it can be used as a reference for the testing of further implementations.

4 Data Structures

According to many authors [Zie84, Bar89, Sch91], the matrices of the systems of linear equations of finite element simulations are not only symmetrical and positive definite, but in general also sparse. Hence, it is obvious exploit this property, since it is possible to (1) avoid unnecessary calculations, and (2) save space in the main memory. Otherwise it would be impossible to solve systems of linear equations of a significant size. In the following, we call an element of a matrix or a vector an *entry*, if its value is not equal zero. We investigated the sparseness for concrete finite element models introduced in Section 6.1 and found out that the percentage of entries ranged from 25% for the small models to 4% for the bigger models. Therefore, we implement matrices with a sparse data structure.

In contrary to the matrices, the vectors are usually not sparse, and therefore, we do not use a sparse data structure.

In order to explain the data structures, we use the following 5×5 matrix A and a 5-dimensional vector b as examples.

$$A = \begin{pmatrix} 10.0 & 4.0 & 3.0 & 0.0 & 0.0 \\ 4.0 & 5.0 & 0.0 & -1.0 & 0.0 \\ 3.0 & 0.0 & 7.0 & 4.0 & 2.0 \\ 0.0 & -1.0 & 4.0 & 6.0 & 0.0 \\ 0.0 & 0.0 & 2.0 & 0.0 & 8.0 \end{pmatrix} \qquad b = \begin{pmatrix} 1.0 \\ 2.0 \\ 4.0 \\ 0.0 \\ -7.0 \end{pmatrix}$$

The matrix A has 15 and the vector b 4 entries. Hence, the percentage of entries of A equals 60%.

4.1 Matrices

The matrices of finite element simulations are build up by so-called *element stiffness matrices*. Their structure depends on the finite element model considered and therefore is only known at run-time. Hence, it is necessary that the matrix can be build up dynamically. In addition, the matrix data structure should support an easy distribution of

```
# x(t) = x(t-1) + gamma(t-1) * s(t-1)
x := proc (t)
  if (t = 0) then
      eval (x0)
  else
      evalf (add (x(t-1), s(t-1), 1.0, Gamma (t-1)))
  fi
end;

# gamma (t) = (s(t) * g(t)) / (s(t) * A * s(t))
Gamma := proc (t)
      evalf (- (dotprod (s(t), g(t))) /
             dotprod (s(t), multiply (A, s(t))))
end;

# s(t) = -g(t) + beta (t) * s(t-1)
s := proc (t)
  if (t = 0) then
      evalf (scalarmul (multiply (H, g(t)), -1.0))
  else
      evalf (add (multiply (H, g(t)), s(t-1), -1.0, Beta(t)))
  fi
end;

# g(t) = A * x(t) - b
g := proc (t)
  evalf (add (multiply (A, x(t)), b, 1.0, -1.0))
end;

# beta(t) = (g(t) * H * g(t)) / (g(t-1) * H * g(t-1))
Beta := proc (t)
     if (t = 0) then
         0
     else
         evalf (dotprod (g(t), multiply (H, g(t))) /
                dotprod (g(t-1), multiply (H, g(t-1))))
     fi
end;

# Recursive Call of the Solver
PCGSolv := proc (t)
        if (norm (g(t), 2) < Prec) then
            eval (x(t))
        else
            PCGSolv (t+1)
        fi
end;

# Preconditioned Conjugate Gradient Solver
PCG := proc (A, b, Precision)
    Prec := Precision;
    H := PreConMatrix (A, b);
    x0 := StartVector (H, b);

    PCGSolv (0)
end;
```

Figure 1: Specification in Maple.

the matrices to several processors and allow an efficient implementation of matrix-vector multiplications.

To fulfill these various requirements, we use a different data structure on the host and on the parallel processors. Both data structures can be easily transformed into each other.

Matrix data structure on the host

On the host, all entries of a matrix are stored in rows, where each row consists of a linked linear list. For every entry, the value and the corresponding column index is stored. Additional information about the number of rows and the total number of entries is also stored in the data structure.

The structure has the advantage, that no zeros are stored, and that it is easily possible to insert or delete entries.

Matrix data structure on the parallel processors

The matrices are distributed onto the parallel processors in a way that each processor a number of successive rows. Therefore, each processor has the information about the number of rows and entries of the whole matrix, the number of locally stored rows, and the row index of the first local row (Figure 2).

As on the host, for each entry the value and the corresponding column index is stored. In the current implementation on the MUSIC-system, the column index uses 4 byte and the value 8 byte of main memory. Due to a different internal representation of a structure consisting of a `long` and a `double`, the host computer allocates 16 byte, the parallel processors only 12 byte. In order to communicate the entries of a matrix from the host to the parallel processors in one continuous memory block, it is necessary that both processor types use the same internal representation. Therefore, two successive entries of a row are stored as a *double entry*, which has the same representation on both processor types, and uses 24 byte.

Each row of the matrix consists of a continuous memory block of double entries. For a distribution of a matrix in rows with predefined communication functions in one communication cycle [Bäu92], all rows have to be stored in one continuous memory block, and the number of double entries of each row has to be the same. Therefore, it is not possible to completely use the memory space, since dummy entries (column index $= -1$, value $= -1$) have to be added. Hence, the information about the maximal number of double entries has to be stored on the host and on the parallel processors as well.

4.2 Vectors

The data structures for vectors do not differ much on the host and on the parallel processors (Figure 3). Both structures store the dimension of the vector and the total number of entries. All elements, even the ones with a zero value, are stored in one continuous memory block.

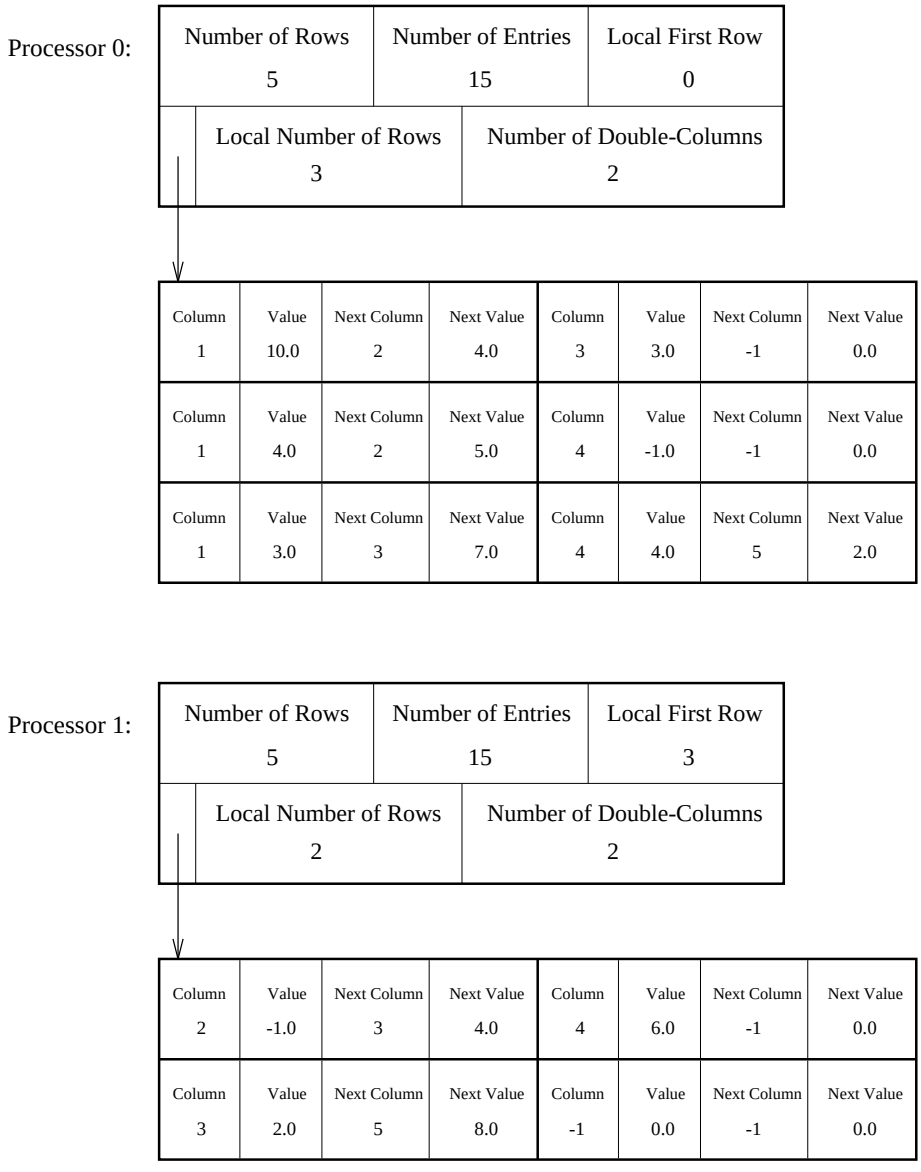

Figure 2: Matrix data structure for two parallel processors.

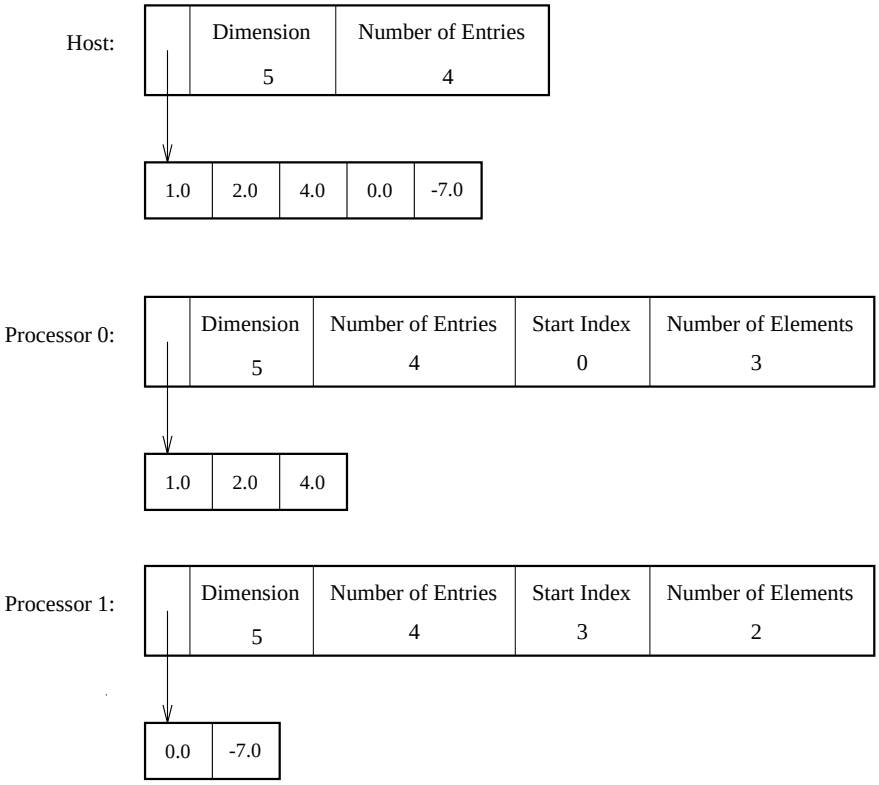

Figure 3: Vector data structure for the host and two parallel processors.

Vectors are distributed onto the parallel processors in a way that each processor has a number of successive elements. Depending on their use, vectors can be distributed quite differently, and it is not necessary to distribute them in the same way as the matrices. For a matrix-vector multiplication, the vector has to be stored completely on every processor (refer to Section 5.2). In order to distribute a vector, the index of the first local element and the total number of local elements is stored.

5 Parallel Algorithm

As we can see from the Maple specification (Figure 1), the algorithm uses the following four operations of linear algebra:

1. addition of two vectors,
2. multiplication of a vector with a scalar,
3. dotproduct of two vectors,
4. matrix-vector multiplication.

The results of profilings of the sequential implementation showed that these four operations are the most time consuming parts of the algorithm. Hence, it is obvious to parallelize these operations in an efficient way. The rest of the sequential algorithm can be used more or less unchanged. As the speedup measurements show (results in Section 7), this parallelization is quite efficient.

The data parallel algorithm we obtained was formulated as hardware independently as possible, in order to guarantee an easy porting onto other SPMD-programmable architectures.

Details of the parallelization of the dotproduct of two vectors and the matrix-vector multiplication are described in the next two sections, respectivly. The other two operations were parallelized analogously; therefore we will not discuss further details.

The program running on the host processor initializes the parallel processors, assembles the matrix A and the vector b of the system of linear equations, distributes them onto the processors, and reads the parallelly computed solution.

5.1 Dotproduct of vectors

The dotproduct of two n-dimensional vectors x and y is defined as

$$x \cdot y := \sum_{i=1}^{n} x_i y_i. \tag{15}$$

In order to calculate this sum on m processors P_k $(1 \leq k \leq m)$ in parallel, we use the following transformation:

$$s_k = \sum_{i=f_k}^{l_k} x_i y_i \tag{16}$$

$$x \cdot y = \sum_{k=1}^{m} s_k. \tag{17}$$

f_k is the index of the first and l_k the index of the last local element of both vectors stored on processor P_k. Each processor P_k calculates the partial sum s_k of the locally stored elements of the vectors, and broadcasts it to all other processors. Finally, on every processor, the m partial sums are added, which leads to the result of the dotproduct on every processor.

If the vectors are distributed evenly onto the processors, we can achieve a good parallel efficiency with this algorithm. Unfortunately, numerical problems can occur, which are described in Section 6.2.

5.2 Matrix-vector multiplication

We consider the matrix-vector multiplication $y = A \cdot x$. The ith element y_i of y is defined as

$$y_i = A_i \cdot x = \sum_{k=1}^{n} a_{i,k} x_k. \tag{18}$$

n is the dimension and A_i is the ith row vector of the matrix A. In order to exploit the sparseness of A, the equation (18) can be modified to

$$y_i = \sum_{k=1}^{n_i} a_{i,c_i(k)} x_{c_i(k)}, \tag{19}$$

where n_i is the number of entries and $c_i(k)$ the column index of the kth entry of the ith row.

For every locally stored row A_i of the matrix A, each processor P_k computes the corresponding element y_i of the matrix-vector product. In order to do so, the matrix A is distributed in rows onto the processors, and the vector x is stored completely on all processors. After the computation, all locally computed elements are sent to all other processors.

If the rows of the matrix are distributed evenly, this parallel algorithm has a good parallel efficiency. In contrast to the dotproduct of two vectors, no further numerical problems occur.

5.3 Data distribution

As mentioned in the last two sections, the rows of the matrices and the elements of the vectors should be distributed evenly onto the parallel processors, in order to achieve a good parallel efficiency. On the MUSIC-system, there exist predefined communication functions for the distribution of three-dimensional data blocks onto the parallel processors [Bäu92]. In the current implementation, these functions are used to distribute the matrices in a way that each processor gets a nearly equal number of successive rows. If there are significant differences in the number of entries per row, such a distribution is not optimal. Therefore, more sophisticated distribution functions have to be used for further implementations.

6 Numerical Behaviour

In order to investigate the numerical behaviour of the implementation, calculations have been made on the MUSIC system itself, on the MUSIC simulator *musim*, and on a DEC Alpha station. The MUSIC simulator *musim* is a tool for developing and testing data parallel MUSIC programs in a UNIX environment [Sco93]. Therefore, it was possible to run the same program with a floating point precision of 44 bit (32 bit mantissa, 12 bit exponent, 9 decimal digits) on the MUSIC hardware and with 64 bit (53 bit mantissa, 11 bit exponent, 15 decimal digits) on the simulator. For a DEC Alpha station, the sequential program was slightly modified to simulate the parallel computation of dotproducts according to Section 5.1, in order to get the same numerical behaviour as the parallel program. Therefore, it was also possible to run the calculations with a floating point precision of 128 bit (113 bit mantissa, 15 bit exponent, 33 decimal digits).

As test-examples, systems of linear equations of finite element simulations, and as preconditioning matrices H, the square of the inverse of the diagonal matrix D^{-1} of A have been used.

Number of iterations 44 bit floating point	Number of elements of a linear beam					
	10	20	40	60	80	100
Number of processors	Dimension of the system of linear equations					
	324	609	1059	1554	2049	2484
3	122	166	428	453	–	–
5	122	166	429	453	381	461
10	122	166	429	454	381	461
15	122	166	428	452	381	461
20	123	166	429	452	381	461
25	122	166	429	453	382	462
30	123	166	428	453	381	461

Table 1: Number of iterations of a finite element simulation of a linear beam with a floating point precision of 44 bit.

Number of iterations 64 bit floating point	Number of elements of a linear beam					
	10	20	40	60	80	100
Number of processors	Dimension of the system of linear equations					
	324	609	1059	1554	2049	2484
3	113	159	408	431	–	–
5	111	159	408	431	365	443
10	113	159	408	431	365	443
15	112	159	408	431	365	443
20	112	159	408	431	364	443
25	112	159	407	430	365	443
30	112	159	408	431	365	443

Table 2: Number of iterations of a finite element simulation of a linear beam with a floating point precision of 64 bit.

6.1 Speed of convergence

We investigated whether the number of iterations is indeed smaller than the dimension of the system of linear equations as mentioned in [Sch91]. For that purpose, systems of linear equations of finite element simulations of a linear beam with 10, 20, 40, 60, 80, and 100 finite elements and a motion segment model of the human spine with 52 finite elements were used. The motion segment model consists of two vertebral bodies and an intervertebral disc [MKD94]. The results of the corresponding simulations are shown in Tables 1 to 3. Due to the lack of memory on the MUSIC system, it was not possible to run the simulation for a beam with 80 and 100 finite elements and the motion segment model on less than 5 processors.

The number of iterations for the linear beams is substantially smaller than the dimension of the corresponding system of linear equations for floating point precisions of 44 and 64

Number of processors	motion segment model dimension $n = 1020$ 44 bit	64 bit	128 bit
5	28,467	18,421	8,733
10	28,668	18,465	8,778
15	28,490	18,439	8,754
20	28,730	18,537	8,736
25	28,494	18,521	8,743
30	28,701	18,547	8,738

Table 3: Number of iterations of a finite element simulation of a motion segment of the human spine.

bit. Moreover, a floating point precision of 44 bit required less than 10 per cent more iterations than 64 bit.

The dimension n of the system of linear equations of the motion segment model is equal to 1020. Therefore, it is astonishing, that more than $28,000$ iterations are needed with a floating point precision of 44 bit, more than $18,000$ iterations with 64 bit, and even more than $8,000$ iterations with 128 bit (Table 3). These high numbers of iterations can be explained by investigating the material properties of the corresponding finite element model. The Young Modulus E [Zie84, Sch91] of the vertebral bodies, nucleus and annulus differ in a factor of five orders of magnitude. Therefore, the eigenvalues of the global stiffness matrix A are spread over a large interval, which leads to a badly conditioned matrix [BT89]. In order to investigate the influence of the Young Modulus on the speed of convergence, the values of E for the nucleus and annulus were changed systematically, while the value for the vertebral bodies was left unchanged at $0.12 \cdot 10^5$ (Table 4). As we can see, significantly less iterations are needed if the values of E do not differ much. The same result was obtained when the Young Modulus of nucleus and annulus was left unchanged, but the Young Modulus of the vertebral bodies was modified.

The results of Table 3 also show that an increase of the floating point precision reduces the number of iterations much more in badly conditioned than in well conditioned equation systems, but does not necessarily lead to a better numerical behaviour.

6.2 Summation effect

From the results of the last section it is seen that the number of iterations do not only depend on the condition of the equation system, but also on the number of processors used.

As described in Section 5.1, the dotproduct of two vectors is calculated in parallel. Depending on the number of processors, the corresponding sum is calculated in a different way, since the parentheses for the summation are set differently. Due to the fact that numerical real arithmetic is commutative, but not associative, different settings of parentheses can lead to different results. The following example shows this effect. The symbol $\oplus$ stands

Young Modulus E of annulus	Young Modulus E of nucleus					
	$0.1 \cdot 10^0$	$0.1 \cdot 10^1$	$0.1 \cdot 10^2$	$0.1 \cdot 10^3$	$0.1 \cdot 10^4$	$0.1 \cdot 10^5$
$0.8 \cdot 10^1$	28,701	9,891	3,658	6,876	4,429	2,043
$0.8 \cdot 10^2$	19,172	8,176	2,903	1,092	1,231	810
$0.8 \cdot 10^3$	5,135	3,238	1,434	436	173	178
$0.8 \cdot 10^4$	1,423	1,048	664	291	109	79
$0.8 \cdot 10^5$	7,268	5,461	3,447	1,513	533	328

Table 4: Number of iterations of a finite element simulation of a motion segment with change of the Young Modulus E of nucleus and annulus on 30 processors and with 44 bit floating point precision.

for the summation of two numerical real numbers. We calculate with a floating point precision of 3 decimal digits.

A sum is calculated in parallel on two processors:

$$(2.01 \oplus 1.06 \oplus 4.04) \oplus (6.16 \oplus 2.05 \oplus 1.38) = 7.11 \oplus 9.59 = 16.7 \tag{20}$$

The summation on three processors leads to:

$$(2.01 \oplus 1.06) \oplus (4.04 \oplus 6.16) \oplus (2.05 \oplus 1.38) = 3.07 \oplus 10.2 \oplus 3.43 = 16.6 \tag{21}$$

For the summation of two real numbers with a different exponent, the mantissa of the real number with the smaller exponent has to be adopted, and its least significant digits disappear. Therefore, $3.07 \oplus 10.2$ equals 13.2, and not 13.27.

In order to avoid the effects of different number of iterations for a different number of processors, the parallel summation was replaced by a sequential one on one processor. By this means, the number of iterations becomes independent of the number of processors used, but the efficiency of the program is reduced considerably. Therefore, we are currently developing an algorithm for parallel summation where the result does not not depend on the number of processors.

Table 5 contains information about the influence of this "summation effect" on the number of iterations. The effect is much bigger for a floating point precision of 32 bit than for 44, 64, and 128 bit.

7 Results

7.1 Speedup measurements

The systems of linear equations of the finite element simulations of the linear beams (Section 6.1) were used to investigate the speedup and the parallel efficiency of the current implementation on the MUSIC-system. Since the number of iterations depends on the number of processors (refer to Section 6.2), the time for one iteration and not the time for

Number of processors	Linear beam with 10 elements dimension $n = 324$			
	32 bit	44 bit	64 bit	128 bit
1	194	123	112	102
3	381	123	113	102
5	167	123	111	102
10	163	124	113	102
15	298	123	112	102
20	171	123	112	102
25	183	123	112	103
30	237	124	112	102

Table 5: Number of iterations of a linear beam with 10 elements.

the whole calculation was used as a measure for speedup and parallel efficiency. Due to memory restrictions on the parallel processors, it was not possible to run all calculations on one processor only. Therefore, the speedup and parallel efficiency were normalized to 1.00 for the smallest number of processors with enough memory. In order to be able to compare speedup values for systems of linear equations of different size, we also normalized the speedup with respect to 5 processors. A summary of the results can be found in Figure 4 and Table 6.

7.2 Floating point precision

Using a parallel program for finite element simulations, we tested whether the MUSIC-system is suitable for finite element simulations in general, and spine simulations in particular. Since the solution of thereby occurring large sparse systems of linear equations is one of the most computationally intensive parts, we concentrated on the parallel solution of the systems of linear equations and not on the other aspects of parallel finite element simulations.

As can be seen from the results in Section 6, the numerical behaviour depends much more on the conditioning of the given problem and the numerical method chosen, and less on the floating point precision used. Therefore, it is more important to concentrate on appropriate numerical methods and not on the floating point precision. Hence, we are currently developing a parallel frontal solver.

8 Conclusions

For the solution of large sparse systems of linear equations of finite element simulations, a conjugate gradient solver has been implemented on the SPMD-programmable MUSIC-system. As a first step, the conjugate gradient method was specified in Maple. Since the Maple specification is also executable, it has been used as a reference for all further implementations. With a profiling of a sequential implementation, the most time consuming

n	P_{min}	P	$\Pi(P)$	$I(P)$	$T(P)$	$T_I(P)$	$s(P)$	$s_5(P)$	$\varepsilon(P)$
		1	1	123	32.92	0.268	1.00	0.27	1.00
		5	5	122	8.80	0.072	3.72	1.00	0.74
		10	10	122	5.30	0.043	6.23	1.68	0.62
324	1	15	15	122	4.15	0.034	7.88	2.12	0.53
		20	20	123	3.63	0.030	8.93	2.40	0.45
		25	25	122	3.27	0.027	9.93	2.67	0.40
		30	30	123	3.11	0.025	10.72	2.88	0.36
		2	1	166	46.21	0.278	1.00	0.47	1.00
		5	2.5	166	21.56	0.130	2.14	1.00	0.86
		10	5	166	12.36	0.075	3.70	1.73	0.74
609	2	15	7.5	166	9.37	0.056	4.96	2.32	0.66
		20	10	166	7.90	0.048	5.79	2.71	0.58
		25	12.5	166	7.07	0.043	6.47	3.05	0.52
		30	15	166	6.44	0.038	7.32	3.42	0.49
		2	1	428	221.77	0.518	1.00	0.48	1.00
		5	2.5	429	101.30	0.236	2.19	1.00	0.88
		10	5	429	56.65	0.132	3.92	1.79	0.78
1059	2	15	7.5	428	41.77	0.098	5.29	2.42	0.70
		20	10	429	34.97	0.082	6.32	2.89	0.63
		25	12.5	429	30.25	0.071	7.30	3.33	0.58
		30	15	428	27.21	0.063	8.22	3.74	0.55
		3	1	453	238.56	0.527	1.00	0.65	1.00
		5	1.66	453	155.68	0.344	1.53	1.00	0.92
		10	3.33	454	86.06	0.190	2.77	1.81	0.83
1554	3	15	5	452	62.86	0.139	3.79	2.48	0.76
		20	6.66	452	51.71	0.114	4.63	3.03	0.69
		25	8.33	453	44.95	0.099	5.32	3.48	0.64
		30	10	453	40.59	0.089	5.92	3.87	0.59
		5	1	381	171.29	0.450	1.00	1.00	1.00
		10	2	381	94.14	0.247	1.82	1.82	0.91
2049	5	15	3	381	68.76	0.180	2.50	2.50	0.83
		20	4	381	56.22	0.148	3.04	3.04	0.76
		25	5	382	48.81	0.127	3.54	3.54	0.71
		30	6	381	43.48	0.114	3.95	3.95	0.66
		5	1	461	254.03	0.551	1.00	1.00	1.00
		10	2	461	139.50	0.303	1.82	1.82	0.91
2484	5	15	3	461	101.51	0.220	2.50	2.50	0.83
		20	4	461	82.48	0.179	3.08	3.08	0.77
		25	5	462	71.76	0.155	3.55	3.55	0.71
		30	6	461	63.42	0.137	4.02	4.02	0.67

Table 6: Summary of speedup measurements: n: dimension of system of linear equations; P_{min}: minimum number of processors used; P: number of processors; $\Pi(P) = \frac{P}{P_{min}}$: normalized number of processors; $I(P)$: number of iterations; $T(P)$: calculation time (in seconds); $T_I(P) = \frac{T(P)}{I(P)}$: time per iteration (in seconds); $s(P) = \frac{T_I(P_{min})}{T_I(P)}$: normalized speedup; $s_5(P) = \frac{s(P)}{s(5)} = \frac{T_I(5)}{T_I(P)}$: speedup normalized to 5 processors; $\varepsilon(P) = \frac{s(P)}{P}$: parallel efficiency.

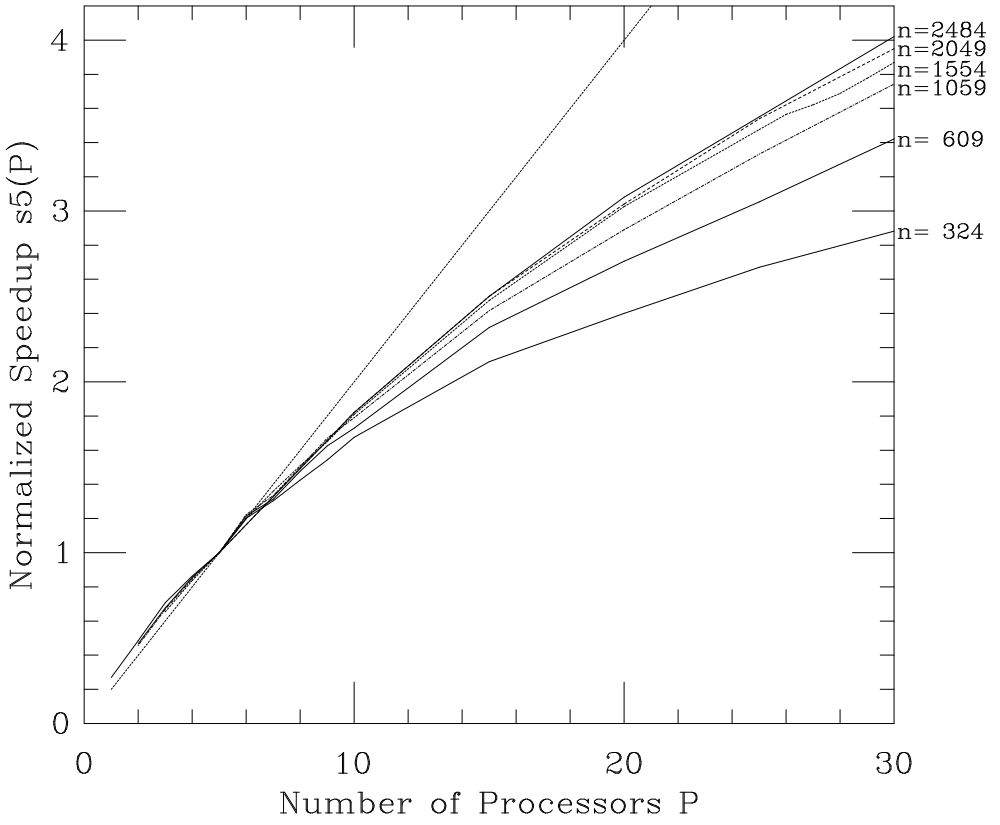

Figure 4: Speedup normalized to 5 processors of the conjugate gradient solver on the MUSIC-system.

parts of a conjugate gradient solver were determined. These parts were parallelized for the implementation on the MUSIC-system.

According to theory, the conjugate gradient method should find the solution of an n-dimensional system of linear equations after at most n iterations. In practice, the number of iterations is supposed to be even much smaller. Therefore, we investigated whether the number of iterations of our conjugate gradient solver is indeed smaller than the dimension of the system of linear equations for floating point precisions of 32, 44, 64, and 128 bit. For that purpose, systems of linear equations of finite element simulations of linear beams and the human spine were used. We found that for the linear beams, the number of iterations in fact is much smaller than the dimension of the corresponding system of equations. The systems of equations for finite element models with strongly unequal materials (differences of 3 orders of magnitude or more), as for example spine models, result in much *more* iteration steps than the corresponding dimension. The results also show that the number of iterations depends more on the numerical conditioning of the given problem and less on the floating point precision of the architecture used.

Using parallel algorithms, it is possible that numerical effects occur which normally do not occur in sequential algorithms. Due to the fact that numerical real arithmetic is not

associative, we found that the result of a parallel computation of a sum may strongly depend on the number of processors used. This "summation effect" influences the numerical behaviour of our conjugate gradient solver and can result in significant differences in the number of iterations of the same finite element simulation, especially if smaller floating point precisions are used.

The human spine mainly consists of vertebral bodies and intervertebral discs with strongly unequal material properties. This results in numerically not well conditioned systems of linear equations. Although we have obtained a good speedup of our implementation, the number of iterations is too large. Therefore, the conjugate gradient method is not suitable for this kind of finite element simulations. Hence, we are currently working on the more difficult parallel frontal solver.

Acknowledgments

We thank Peter Kropf for having organized the SPINET project, Marek Matyjewski for helping us with the Warsaw program, and Karl Guggisberg for reviewing.

References

[Amm91] L. Ammeraal. *C for Programmers: A Complete Tutorial based on the ANSI Standard.* Wiley, second edition, 1991.

[Bäu92] B. Bäumle. The MUSIC-Tutorial. Technical report, ETH Zürich, Institut für Elektronik, Gloriastrasse 35, CH-8092 Zürich, 1992.

[Bar89] P. Bartelt. *Finite Element Procedures on Vector/tightly coupled Parallel Computers.* PhD thesis, ETH Zürich, 1989.

[BBC+93] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhoui, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[BH82] G. Beer and W. Haas. A Partitioned Frontal Solver for Finite Element Analysis. *Journal of Numerical Methods in Engineering*, 18:1623–1654, 1982.

[BT89] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice Hall, 1989.

[CGG+91] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V Language Reference Manual.* Springer, 1991.

[Iro70] B.M. Irons. A Frontal Solution Program for Finite Element Analysis. *Journal for Numerical Methods in Engineering*, 2:5–32, 1970.

[MKD94] M. Matyjewski, P.G. Kropf, and M. Dietrich. Finite element method simulation of flow induced deformations in the intervertebral disc. In *Proceedings of the 1st ISSSL conference*, Bruxelles, 1994.

[Sch88] H.R. Schwarz. *Numerische Mathematik*. B.G. Teubner, zweite Auflage, 1988.

[Sch91] H.R. Schwarz. *Methode der Finiten Elemente*. Teubner, 1991.

[Sco93] W. Scott. MUSIM: The MUSIC-Simulator. Technical report, ETH Zürich, Institut für Elektronik, Gloriastrasse 35, CH-8092 Zürich, 1993.

[SRS68] H.R. Schwarz, H. Rutishauser, and E. Stiefel. *Numerik symmetrischer Matrizen*. B.G. Teubner, 1968.

[Wol88] S. Wolfram. *Mathematica*. Addison-Wesley, 1988.

[Zie84] O.C. Zienkiewicz. *Methode der Finiten Elemente*. Hanser, dritte Auflage, 1984.