Cache Issues of Algebraic Multigrid Methods for Linear Systems with Multiple Right-Hand Sides*

G. Haase and S. Reitzinger

25th March 2002

Institute of Computational Mathematics
Johannes Kepler University Linz
{ghaase,reitz}@numa.uni-linz.ac.at

Abstract

This paper concerns with the solution of a linear system with multiple right-hand sides. Such problems arise from non-linear, time-dependent, inverse or optimization problems. In order to solve this problems efficiently we use variants of the preconditioned conjugate gradient method and combine them with cache aware techniques. Prior to that we describe the acceleration of AMG itself by using sophisticated cache aware algorithms. Numerical studies including one application in life sciences are presented that show the high efficiency of the proposed methods.

Keywords : Algebraic Multigrid, Preconditioned Conjugate Gradient Methods, Cache Algorithms, Non Uniform Memory Access (NUMA).

1 Introduction

The simulation of stationary processes is described by elliptic, (selfadjoint) partial differential equations (PDEs). Solving such a direct problem, e.g., determining the state variables from given source and boundary data, via a finite element (FE) discretization is standard in numerical software nowadays. The situation changes for time-dependent, nonlinear, optimization, or inverse problems. Here, the numerical solution strategy yields a sequence of linear equations and it is often necessary to solve a huge number of linear systems with different right-hand sides in order to solve the overall problem. In this paper we discuss related solution methods for multiple right-hand sides and constant system matrix, i.e.,

$$K_h \underline{u}_h^i = \underline{f}_h^i \qquad i = 1, \ldots, N_r \tag{1}$$

where $K_h \in \mathbb{R}^{N_h \times N_h}$ denotes a symmetric positive definite (SPD) system matrix, $\underline{f}_h^i \in \mathbb{R}^{N_h}$ a given i^{th} right-hand side, $\underline{u}_h^i \in \mathbb{R}^{N_h}$ the corresponding solution vector and N_r expresses the overall number of right-hand sides. We are especially interested in the case where the system matrix stems from a second order selfadjoint elliptic (parabolic) PDEs. In addition we distinguish two different settings of the given right-hand sides.

*This research has been partially supported by the Austrian Science Fund FWF within the SFB "Numerical and Symbolic Scientific Computing" under grant SFB F013 and under project P14953 "Robust Algebraic Multigrid Methods and their Parallelization".

1. The right-hand sides can be computed in advance, e.g., inverse or optimization problems.

2. The right-hand sides are computed successively, e.g., time-dependent problems.

In many applications the right-hand sides $\{\underline{f}_h^i\}_{i=1}^{N_r}$ are not arbitrary, but the differences are small in a certain sense for $i = 1, \ldots, N_r$ or at least for a subsequence $i = s_1, \ldots, s_2$, $1 \leq s_1 < s_2 \leq N_r$. In [3, 11, 12, 17, 19] the properties of such a sequence of right-hand sides was exploited by using variants of the conjugate gradient (CG) (or equivalently the Lanczos) method. The first ideas concerning the CG method with multiple right-hand sides are due to [11, 12] and later an analysis of the method was given in [3, 17, 19]. An application for the sensitivity analysis in shape optimization can be found in [5]. These methods store the Krylov subspace vectors which requires memory. On the other hand memory was very expensive at this time and therefore the methods became not very popular. Additionally the known preconditioners for the CG method where not optimal and therefore the Krylov subspace became very large. However, variants were suggested (see [17, 19]) that need not to store the Krylov subspace if the right-hand sides could be calculated in advance.

Nowadays there are optimal preconditioners (with respect to memory and arithmetic costs) as multigrid (MG) methods [2, 18, 9] available for non-trivial practical problems so that the Krylov subspace can be kept small. For some theory on MG preconditioners see [10]. Since geometric multigrid methods have several drawbacks when they are applied to engineering applications we use algebraic multigrid (AMG) methods which essentially require the system matrix and the right-hand side of the original discretization, see [16, 14, 8, 18] for details.

The second part of the paper is concerned with cache aware techniques for MG-methods [6, 7] and reduction of arithmetic costs in an implementation. Furthermore we apply such cache aware algorithms for simultaneous right-hand sides even if they do not correlate.

The remaining paper is organized as follows: Section 2 describes the algorithmical tuning of MG codes in general which is especially applied to the AMG software package PEBBLES [13]. Section 3 is concerned with simultaneous right-hand sides and an appropriate memory management. Additionally we describe the preconditioned CG version for multiple right hand sides. Section 4 presents numerical studies with academic and real life applications followed by conclusions and further remarks in Section 5.

2 Algorithmical Improvements of Multigrid

We have to solve equation (1) N_r times (e.g., $N_r \approx 10000$) for different right-hand sides by using a MG preconditioned CG method. Therefore, even a minor saving of solution time for one solve accumulates to a substantial faster response to the overall problem.

MG-algorithms have the problem that their response time is much more determined by data access than by floating point operations. This is caused by the characteristics of recent processors where the floating point unit can operate much faster than the data can be transfered from main memory into its registers. The problem increases with systems of equations resulting from unstructured FE-meshes.

All CPUs possess up to three levels of cache and having all data available in the fastest cache would be the perfect data management for all sorts of algorithms. Unfortunately, our problems are much too big to fit in the caches and so the CPU has to load those data needed next into the faster caches (taking also into account the cache hierarchy). Each time the CPU does not find its data in the cache, i.e., a cache miss occurs, the processor has to stop until the data are transfered from main memory or lower level caches and the whole algorithm is significantly delayed. Recent processor and compiler technology tries to decrease the number of cache misses by data prefetching, i.e., more data then actually needed are load into the cache. This technique speculates that subsequent data in memory are also next needed for calculations [6]. Therefore, our data structures should support data prefetching by storing data linearly in the memory. Figure 1 presents a closer look on the very flexible pointer based storage scheme for the sparse stiffness matrix K_h resulting from an unstructured FE-mesh used in the original AMG-code PEBBLES [13] (written in C++). However the memory access

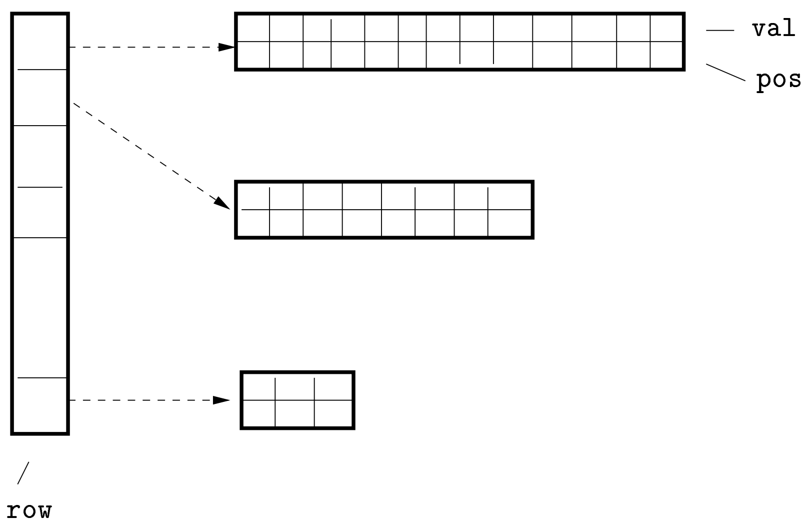

Figure 1: Principle structure of the original matrix memory management.

pattern of these data structures is not well suited for cache efficient implementations since data of adjacent matrix rows may be stored in non-adjacent memory areas. Consequently, we substituted that storage scheme by the classical CRS format for sparse matrices containing vectors `val`, `indx`, `iend` for storing matrix values, column indices and indices of last row entry, see Figure 2. We added an additional vector storing the diagonal pointers `idiag` to accelerate smoothing procedures. The interpolation weights should be stored in the original CRS format. These changes in the matrix and interpolation weight storage accelerated the code by 10-15%.

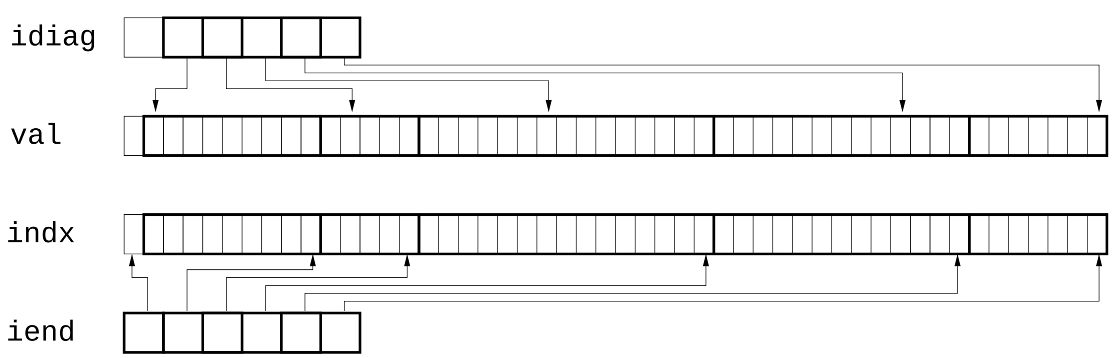

Figure 2: Matrix in CRS storage with additional diagonal pointer.

Another way to accelerate MG-codes consists in avoiding data traffic and arithmetic operations that are not necessary. Let us have a closer look at the classical MG algorithm (see, e.g. [9, 18, 2]), presented in Algorithm 1 where $q = 1$ denotes the finest level and $q = \ell$ the coarsest level. $\{K_j, P_j\}_{j=1}^{\ell}$ includes system and interpolation matrices for all levels. In the following we discuss the MG algorithm for one right-hand sides only and denote two consecutive fine and coarse levels by q and $q+1$, respectively. A modular concept of the original MG

Algorithm 1 Multigrid $\underline{u}_q \longleftarrow \mathrm{MG}(\{K_j, P_j\}_{j=1}^{\ell}, \underline{u}_q, \underline{f}_q, q)$

if $q == \ell$ **then**
  $\underline{u}_q \longleftarrow$ SOLVE $(K_q \cdot \underline{u}_q = \underline{f}_q)$
else
  $\widetilde{\underline{u}}_q \longleftarrow$ SMOOTHF$(K_q, \underline{u}_q, \underline{f}_q)$
  $\underline{d}_q \longleftarrow \underline{f}_q - K_q \cdot \widetilde{\underline{u}}_q$
  $\underline{d}_{q+1} \longleftarrow (P_q)^T \cdot \underline{d}_q$
  $\underline{w}_{q+1} \longleftarrow 0$
  $\underline{w}_{q+1} \longleftarrow \mathrm{MG}(\{K_j, P_j\}_{j=1}^{\ell}, \underline{w}_{q+1}, \underline{d}_{q+1}, q+1)$
  $\underline{w}_q \longleftarrow P_q \cdot \underline{w}_{q+1}$
  $\widehat{\underline{u}}_q \longleftarrow \widetilde{\underline{u}}_q + \underline{w}_q$
  $\underline{u}_q \longleftarrow$ SMOOTHB$(K_q, \widehat{\underline{u}}_q, \underline{f}_q)$
end if

algorithm realizes the calculation of the defect $\underline{d}_q \longleftarrow \underline{f}_q - K_q \cdot \widetilde{\underline{u}}_q$ in a way that the result of matrix-vector multiplication is stored in a temporary vector which is subtracted from $\underline{f}_q$ afterwards. Both steps can be combined in a routine DEFECT which saves two accesses to the temporary vector.

The interpolation of the correction from the coarse grid and the correction of to the existing solution vector with it, i.e.,

$$\underline{u}_q = \underline{u}_q + P_q \cdot \underline{w}_{q+1} \quad , \tag{2}$$

is usually realized in three steps ($\underline{w}_q \longleftarrow 0$, $\underline{w}_q \longleftarrow \underline{w}_q + P_q \cdot \underline{w}_{q+1}$, $\underline{u}_q \longleftarrow \underline{u}_q + \underline{w}_q$). The direct implementation of (2) in the routine INTCORR requires only minor changes in the given interpolation routine but saves initialization and access to the auxiliary vector $\underline{w}_q$. Algorithm 2 presents the modified MG algorithm.

A careful investigation of Algorithm 2 leads to the suspect that some redundant calculations will be performed in SMOOTHF and DEFECT since the smoother as well as the defect calculation require matrix-vector operations. And indeed, some results of the last Gauss-Seidel forward (analogously, backward) iteration (3) can be reused in the following defect

Algorithm 2 Modified Multigrid $\quad \underline{u}_q \longleftarrow \mathrm{MG}(\{K_j, P_j\}_{j=1}^{\ell}, \underline{u}_q, \underline{f}_q, q)$

if $q == \ell$ **then**
  $\underline{u}_q \longleftarrow$ SOLVE $(K_q \cdot \underline{u}_q \ = \ \underline{f}_q)$
else
  $\underline{\widetilde{u}}_q \longleftarrow$ SMOOTHF$(K_q, \underline{u}_q, \underline{f}_q)$
  $\underline{d}_q \longleftarrow$ DEFECT$(K_q, \underline{\widetilde{u}}_q, \underline{f}_q)$
  $\underline{d}_{q+1} \longleftarrow (P_q)^T \cdot \underline{d}_q$
  $\underline{w}_{q+1} \longleftarrow 0$
  $\underline{w}_{q+1} \longleftarrow \mathrm{MG}(\{K_j, P_j\}_{j=1}^{\ell}, \underline{w}_{q+1}, \underline{d}_{q+1}, q+1)$
  $\underline{\widehat{u}}_q \longleftarrow$ INTCORR$(P_q, \underline{\widetilde{u}}_q, \underline{w}_{q+1})$
  $\underline{u}_q \longleftarrow$ SMOOTHB$(K_q, \underline{\widehat{u}}_q, \underline{f}_q)$
end if

calculation (4), $\forall i = 1, \ldots, n := N_q$:

$$\widetilde{u}_i \;=\; K_{ii}^{-1} \underbrace{\Big(\underbrace{f_i - \sum_{j=1}^{i-1} K_{ij} \widetilde{u}_j}_{=:r_i} - \sum_{j=i+1}^{n} K_{ij} u_j \Big)}_{=:v_i} \;=\; K_{ii}^{-1} v_i \tag{3}$$

$$d_i \;=\; f_i - \sum_{j=1}^{n} K_{ij} \widetilde{u}_j \;=\; \underbrace{f_i - \sum_{j=1}^{i-1} K_{ij} \widetilde{u}_j}_{r_i} - \underbrace{K_{ii} \widetilde{u}_i}_{=v_i} - \sum_{j=i+1}^{n} K_{ij} \widetilde{u}_j$$

$$\;=\; r_i - v_i - \sum_{j=i+1}^{n} K_{ij} \widetilde{u}_j \tag{4}$$

Note, that we have access only to the non-zero elements of a matrix row although we write $j = 1, \ldots, n$ in order to simplify the notation. We combine the last smoothing sweep (3) and defect correction (4) for general matrices to Algorithm 3. The old vector $\underline{u}$ is overwritten by the new vector $\underline{\widetilde{u}}$ and thus no additional memory is required.

Algorithm 3 Merging of last smoothing sweep and defect calculation for general matrices in CRS

for all $i = 1, \ldots, n$ **do**
  $r \longleftarrow f_i - \sum_{j=1}^{i-1} K_{ij} \widetilde{u}_j$
  $inv \longleftarrow 1.0 / K_{ii}$
  $v \longleftarrow r - \sum_{j=i+1}^{n} K_{ij} u_j$
  $\widetilde{u}_i \longleftarrow v * inv$
  $d_i \longleftarrow r - v$
end for
for all $i = 1, \ldots, n$ **do**
  $d_i \longleftarrow d_i - \sum_{j=i+1}^{n} K_{ij} \widetilde{u}_j$
end for

Algorithm 3 requires $\frac{3}{2} \cdot n^2$ arithmetical operations and accesses with matrix elements instead of $2 \cdot n^2$ when smoothing and defect calculation are separated. Unfortunately, we have to start again with the first row when accessing some matrix elements in the second loop, i.e., the complete matrix has to be transfered into the cache again. The two loops in the update of the defect

$$d_i \longleftarrow d_i - \sum_{j=i+1}^{n} K_{ij}\widetilde{u}_j \qquad \forall i = 1, \dots, n$$

can be interchanged and yield together with a swapping of indices i and j to

$$\{d_j \longleftarrow d_j - K_{ji}\widetilde{u}_i\}_{j=1}^{i-1} \qquad \forall i = 2, \dots, n \ .$$

Now we could merge the two i-loops in Algorithm 3 but the matrix access pattern would be much worse than before. In case of a symmetric matrix, i.e. $K_{ij} = K_{ji}$, we can use K_{ij} instead of K_{ji} and suddenly we have to load each matrix row only once into cache. The combined routine SMOOTHFDEFECT for symmetric matrices is represented in Algorithm 4.

Algorithm 4 SMOOTHFDEFECT - Last smoothing sweep and defect calculation for symmetric matrices in CRS

for all $i = 1, \dots, n$ **do**
  $r \longleftarrow f_i - \sum_{j=1}^{i-1} K_{ij}\widetilde{u}_j$
  $inv \longleftarrow 1.0/K_{ii}$
  $v \longleftarrow r - \sum_{j=i+1}^{n} K_{ij}u_j$
  $\widetilde{u}_i \longleftarrow v * inv$
  $d_i \longleftarrow r - v$
  for all $j = 1, \dots, i-1$ **do**
    $d_j \longleftarrow d_j - K_{ij}\widetilde{u}_i$
  end for
end for

The correction on the coarse grid $\underline{w}_{q+1}$ is set to zero before the MG-procedure for the next coarser level is called in Algorithm 2. This means that $\underline{u}_q = 0$ is the initial solution for the first pre-smoothing sweep on that coarser level and iteration (3) simplifies to

$$\widehat{u}_i \;=\; K_{ii}^{-1}\Big(f_i - \sum_{j=1}^{i-1} K_{ij}\widetilde{u}_j\Big) \tag{5}$$

which saves half of the arithmetic and memory operations and can be implemented in a routine SMOOTHF0.

From our experience, the fastest MG solver is often a V-cycle using only one pre-smoothing and one post-smoothing sweep. This allows us to combine (5) with (4) on coarser levels to

$$\begin{aligned} \widehat{u}_i &= K_{ii}^{-1}\underbrace{\Big(f_i - \sum_{j=1}^{i-1} K_{ij}\widetilde{u}_j\Big)}_{=:v_i=:r_i} \\ d_i &= -\sum_{j=i+1}^{n} K_{ij}\widetilde{u}_j \end{aligned}$$

resulting in n^2 arithmetical operations and memory accesses when the only smoothing sweep is combined with the defect calculation in a routine SMOOTHF0DEFECT. The realization of SMOOTHF0 and SMOOTHF0DEFECT can be easily derived, the last one profits again from the matrix symmetry.

The general routine SMOOTHDEF($K_q, \underline{u}_q, \underline{f}_q, \underline{d}_q, \nu, q$) for merging smoothing and defect calculation on all levels has to take into account the implementation differences regarding the number ν of smoothing sweeps and the actual level q. The initial correction $\underline{u}_q$ on the coarser grids is always zero but we have to assume an arbitrary non-zero initial guess for the solution $\underline{u}_q$ on the fine grid. Replacing SMOOTHF and DEFECT in Algorithm 2 with SMOOTHDEF results in a fast MG procedure.

Another important ingredient of AMG methods is the coarse grid solver. Since the average number of non-zero entries is growing on coarser levels in general, the application of a sparse solver becomes more and more expensive and in addition the overall convergence rate of the AMG method becomes worser and worser. Thus we like to construct a minimal number of levels ℓ. On the other side, the factorization of the coarse matrix using a sparse direct solver is relatively slow compared to the setup of AMG for many unknowns N_ℓ. Consequently, we have to find a compromise when to stop the AMG coarsening process. In 2D we apply a sparse direct solver up to 50.000 unknowns and the 3D limit is 10.000 unknowns. The package SuperLU [4] is used as a sparse direct solver in our applications.

Remark 2.1. *Since the problem class considered in our applications consists of sparse symmetric matrices K_q with NZE_q non-zero entries on level q, one matrix times vector operation requires NZE_q memory accesses instead of N_q^2 for fully occupied matrices. Therefore the overall operation count for smoothing and defect calculation has been reduced from $2 \cdot \mathrm{NZE}_q$ to NZE_q in this section.*

3 Multiple right-hand sides

The second part of the paper concerns with the efficient treatment of multiple right-hand sides. We discuss two different methods. The first is related to cache aware algorithms and in principle a linear system with several right-hand sides is solved simultaneously. Therefore we need a special memory treatment of the right-hand sides. The other technique is related to the preconditioned CG method, where the previous technique is also applied.

3.1 Simultaneous right-hand sides

The idea for this method simply consists in solving some few equations at the same time independently of potential right-hand side correlations. Thus this method is well suited if the right-hand sides can be calculated in advance. The system matrix K_h remains the same and so we change the code such that BLOCKSIZE systems of equations can be solved at the same time, see Algorithm 5. The most inner loop in Algorithm 5 can be expressed as

$$\mathrm{blockdiag}\{K_h\} \left(\underline{u}_h^k\right)_{k=s}^{s+\mathrm{BLOCKSIZE}-1} = \left(\underline{f}_h^k\right)_{k=s}^{s+\mathrm{BLOCKSIZE}-1} ,$$

i.e., there is a chance to handle all BLOCKSIZE right hand sides and solution vectors at the same time accessing the (unstructured) matrix K_h only once per matrix-vector operation inside the preconditioned CG (PCG) solution method used for solving the systems of equations.

Algorithm 5 Blocked loop for BLOCKSIZE right-hand sides.

for all $s = 1, 1 + \text{BLOCKSIZE}, 1 + 2 * \text{BLOCKSIZE}, \ldots, N_r$ **do**

  $\vdots$

  for all $k = s, \ldots, s + \text{BLOCKSIZE} - 1$ **do**

    Solve $K_h \underline{u}_h^k = \underline{f}_h^k$

  end for

  $\vdots$

end for

Let us have a look at two possible implementations of the matrix-vector operation where the matrix is stored in CRS format pointing to the end of the row, see Figure 2. First, we store BLOCKSIZE vectors separately as represented in Algorithm 6. Second, we store the first

Algorithm 6 $(\underline{v}^k) \longleftarrow K \cdot (\underline{u}^k)$ with BLOCKSIZE separate vectors

```
for (i=1; i<=size; i++)                    // size rows in matrix
  {
    j1  = iend[i-1]+1; j2  = iend[i];      // begin, end of row
    for (k=1; k<=BLOCKSIZE; k++)  v[i][k] = 0.0;
    for (j=j1; j<=j2; j++)
      {
        jj  = indx[j];                     // column pointer
        aij = val[j];                      // matrix entry
        for (k=1; k<=BLOCKSIZE; k++)
             v[i][k] += aij * u[jj][k];
      }
  }
```

entries of all BLOCKSIZE vectors followed by the second entries, etc.. The implementation with block vectors is realized in Algorithm 7. Both algorithms require the same amount of memory accesses and each matrix element is used BLOCKSIZE times in the most inner loop. But the block vector version in Algorithm 7 is much better suited for state-of-the-art processors since more useful data is found in one cache line, see also Figure 3 for illustration. The indirect addressing of vectors $\underline{u}_m$ via the column pointer causes misprefetching of data

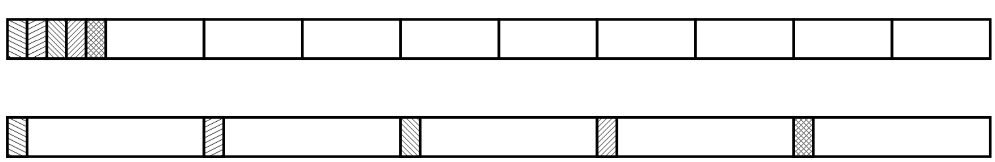

Figure 3: Storage scheme for separate vectors (bottom) and block vectors (top)

for the cache so that the cache lines have to be reloaded. In that case Algorithm 6 has reload BLOCKSIZE-1 times more cache lines than Algorithm 7. Therefore, Algorithm 7 has a higher cache hit rate than Algorithm 6 and is faster at the end.

The changes in the CG algorithm are not dramatically. One should only be aware that the scalars of the CG algorithm will be represented by small vectors containing BLOCKSIZE

Algorithm 7 $(\underline{v}^k) \longleftarrow K \cdot (\underline{u}^k)$ with block vectors

```
for (i=1; i<=size; i++)                    // size rows in matrix
  {
    j1  = iend[i-1]+1; j2  = iend[i];      // begin, end of row
    ii  = (i-1)*BLOCKSIZE;
    for (k=1; k<=BLOCKSIZE; k++)  v[ii+k] = 0.0;
    for (j=j1; j<=j2; j++)
      {
        jj  = (indx[j]-1) * BLOCKSIZE;     // column pointer
        aij = val[j];                      // matrix entry
        for (k=1; k<=BLOCKSIZE; k++)
             v[ii+k] += aij * u[jj+k];
      }
  }
```

scalars. The stopping criterion for the CG changes in that way that all relative errors have to be smaller than the tolerance. The simultaneous calculation of the inner product can be seen in Algorithm 8. Similar chances as in Algorithm 7 have to be made in the MG components

Algorithm 8 Inner products $(\alpha^k)_{k=1}^{\mathrm{BLOCKSIZE}} \longleftarrow \left\langle (\underline{u}^k), (\underline{v}^k) \right\rangle$

```
for (k=1; k<=BLOCKSIZE; k++)   alpha[k] = 0.0;

for (i=0; i<size;i++)
  {
    ii = i*BLOCKSIZE;
    for (k=1; k<=BLOCKSIZE; k++)
         alpha[k] += v[ii+k]*u[ii+k];
  }
```

smoothing, defect calculation, interpolation and prolongation.

Remark 3.1. *The handling of simultaneous vectors causes many most inner loops as*

```
for (k=1; k<=BLOCKSIZE; k++)  h[k] -= aij * u[jj+k];
```

in SmoothFDefect. *The parameter* BLOCKSIZE *is already known at compile time by a simple* `#define` *directive in a header file in order to support code optimization provided by the compiler as much as possible. We applied this simple idea to the PCG with an AMG preconditioner using the improved routines from Section 2 and accelerated the code by a factor of ∼3,5. We expected that the compiler will unroll the inner loops and that the compiler will recognize that there is no data flow between vectors belonging to different systems of equations. We checked this assumption for* BLOCKSIZE$=5$ *by unrolling manually all* BLOCKSIZE *loops in* Mult, SmoothB, SmoothF, SmoothF0, SmoothFDefect, SmoothF0Defect, Defect *and we resolved data dependencies by introducing variables* h_1, h_2, h_3, h_4, h_5 *instead of vector* $h[1, \ldots, 5]$. *These small changes resulted in an additional performance gain of 15-25% mainly achieved in the Gauss-Seidel smoothers because the code manipulations reduced significantly the compiler assumptions on data dependencies. This allows a more aggressive data and instruction prefetching at run time.*

3.2 Preconditioned Lanczos-Galerkin Projection methods

While the multiple right-hand side method does not assume any information on the right-hand side properties, i.e., they are completely independent from each other, the second strategy profits from the speculation or the potential knowledge that sequencing right-hand sides are close to each other in a certain sense. Typically such problem setting arises from time-dependent equations where the right-hand sides vary most often slowly in time. We study the method of [17, 19] where a Lanczos method was used to derive a technique for the solution of a linear system with multiple right-hand sides. The condition number $\kappa(K_h) = \frac{\lambda_{\max}(K_h)}{\lambda_{\min}(K_h)}$ of the linear systems we are interested in behaves like $\mathcal{O}(h^{-2})$ as h tends to zero, where h denotes the typical mesh length of an FE-discretization. Thus the matrices are ill conditioned and therefore a CG method would require many iterations in order to solve the linear system up to a relative accuracy ϵ.

In order to circumvent this problem we use a preconditioner C_h which is optimal with respect to memory and arithmetic costs. Conversely a constant number of iterations is required in order to solve a linear equation up to a fixed relative accuracy ϵ. Let us assume that the linear system (1) is preconditioned by C_h and we have $N_r = 2$, i.e.,

$$C_h^{-1/2} K_h C_h^{-1/2} \underline{v}_h^i = C_h^{-1/2} \underline{f}_h^i \qquad \underline{u}_h^i = C_h^{-1/2} \underline{v}_h^i \qquad i = 1, 2 \ .$$

Using the abbreviations $B_h = C_h^{-1/2} K_h C_h^{-1/2}$ and $\underline{g}_h^i = C_h^{-1/2} \underline{f}_h^i$ we can apply the original technique of [17, 19] to the equations

$$B_h \underline{v}_h^i = \underline{g}_h^i \qquad i = 1, 2\,. \tag{6}$$

The CG method applied to (6) and $i = 1$ produces after m-steps a Krylov subspace

$$\mathcal{K}_m(B_h, \underline{z}_h^{1,0}) = \operatorname{span}\{B_h^0 \underline{z}_h^{1,0}, B_h \underline{z}_h^{1,0}, \ldots, B_h^{m-1} \underline{z}_h^{1,0}\}$$

with $\underline{z}_h^{1,0} = \underline{g}_h^1 - B_h \underline{v}_h^{1,0}$, $\underline{v}_h^{1,0} \equiv 0$. Furthermore, the quantities of the Lanczos method are written down for the original linear system (1). The approximative solution $\underline{u}_h^{1,m}$ of the original system (1) can be written in the form

$$\underline{u}_h^{1,m} = \sum_{i=0}^{m-1} \beta_i^1 \cdot C_h^{-1} \underline{r}_h^{1,i}$$

with $\underline{r}_h^{1,i} = K_h^i \underline{r}_h^{1,0}$, $\underline{r}_h^{1,i} = \underline{f}_h^1$. The coefficients β_i^j are calculated via the formula

$$\beta_i^j = (T_m^{-1} \alpha^j)_i \qquad j = 1, 2\,, \tag{7}$$

where $T_m \in \mathbb{R}^{m \times m}$ is the tridiagonal matrix arising from the Lanczos process, see e.g. [17, 19]. The vector $\alpha^j \in \mathbb{R}^m$ is calculated by

$$\alpha_i^j = \frac{\langle C_h^{-1} \underline{r}_h^{1,i}, \underline{f}_h^j \rangle}{\langle C_h^{-1} \underline{r}_h^{1,i}, \underline{r}_h^{1,i} \rangle} \qquad j = 1, 2\,. \tag{8}$$

For further considerations let $\underline{f}_h^2$ be an other right-hand side for (1). Since $\underline{f}_h^1$ and $\underline{f}_h^2$ are assumed to be close to each other we perform the Galerkin projection onto $\mathcal{K}_m(B_h, \underline{z}_h^{1,0})$

of $\underline{f}_h^2$, i.e.,

$$\tilde{\underline{f}}_h^2 = \sum_{i=0}^{m-1} \alpha_i^2 \cdot \underline{r}_h^{1,i} ,$$

with α_i^2 defined in (8). Consequently we get an approximate solution $\tilde{\underline{u}}_h^2$ for the solution of (1) in the form

$$\tilde{\underline{u}}_h^2 = \sum_{i=1}^{m-1} \beta_i^2 \cdot C_h^{-1} \underline{r}_h^{1,i}$$

with right-hand side $\underline{f}_h^2$ with β_i^2 defined in (7). From an heuristical point of view $\tilde{\underline{u}}_h^2$ is a good approximation to $\underline{u}_h^2$ if the right-hand sides $\underline{f}_h^1$ and $\underline{f}_h^2$ are close. A rigorous analysis and algorithms are given in [17, 19, 3].

Without an optimal preconditioner C_h the required basis might be very large and therefore the memory requirements are unacceptable. But with an optimal preconditioner as e.g. AMG the required basis is very small. However we have to store $2 \cdot m$ vectors in $\mathbb{R}^{N_h}$, namely $\{C_h^{-1}\underline{r}_h^{1,i}\}_{i=0}^{m-1}$ and $\{\underline{r}_h^{1,i}\}_{i=0}^{m-1}$. For the efficient calculation of the Galerkin projection we use the presented memory management for multiple vectors. In this way cache can be used efficiently. In order to perform a stable Galerkin projection we use the modified Gram-Schmidt-Algorithm (see [19]).

If the right-hand sides can be calculated in advance then no storage of the Krylov basis is required [17, 19, 3].

4 Numerical Studies

The presented methods are implemented in the AMG software package PEBBLES [13]. We always apply one symmetric V-cycle with one pre- and one post-smoothing step as preconditioner C_h. The CG iteration was stopped when a relative error reduction of ϵ has been achieved in the $\|\cdot\|_{KC^{-1}K}$-norm.

We used the standard C++ MIPSpro Compilers, Version 7.30 with the compiler options
`-64 -mips4 -r10000 -Ofast=ip27 -LNO:pf2=on:prefetch=2`
`-OPT:IEEE_arithmetic=3:roundoff=3:alias=RESTRICT -IPA`.
on SGI and ORIGIN running under IRIX 6.5. Adapting the options to the processor on the ORIGIN 3800 did not influence the run time too much. The LINUX computer was running the distribution by S.u.S.e. (version 7.1) as operating system and we used the g++ (gcc) compiler, Version 2.95 with the compiler options `-O3 -m486 -ffast-math`. Again, we tried several fancy option without any further gain in CPU time.

Before we present the results for multiple right-hand sides we mention that the performance gain for one right-hand side is already 1.7 due to the improved memory management and algorithmical improvements from Section 2.

4.1 A 2D case study: simultaneous right-hand sides

The first test example is concerned with the potential equation on the unit square and zero Dirichlet boundary conditions on one side of the square. On the remaining boundary we assume homogenous Neumann boundary conditions. For an FE-discretization we use bilinear FE-functions on the equidistant grid. The right-hand sides are assumed to be of the form

$$(\underline{f}_h^i)_j = \sum_{k=0}^{n} \delta_{i,j+k} \cdot \zeta_k, \quad \zeta_k \neq 0\,, \qquad i = 1, \ldots, N_h \quad j = 1, \ldots, N_r\,,$$

which are linear dependent if $n < N_r$. δ_{ij} denotes the Kronecker symbol.

First we study the method of simultaneous right-hand sides (see Subsection 3.1) and fix the number of unknowns by $N_h = 90.000$. Since we need no assumptions on the right-hand sides we fix $n = N_r = \text{BLOCKSIZE}$ and $\zeta_k = 1,\ k = 0, \ldots, n$. The linear equation is solved up to a relative accuracy $\epsilon = 10^{-8}$. The performance gain with respect to simultaneous right-hand sides can be seen in Figure 4. The first observation is that we get a rapid speedup for

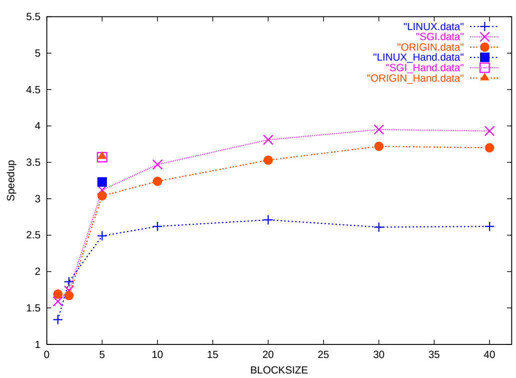

Figure 4: Performance gain with respect to simultaneous right-hand sides.

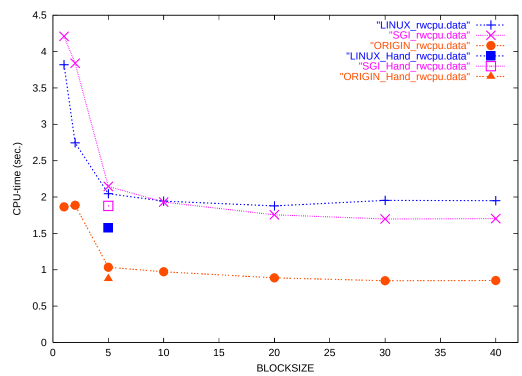

Figure 5: CPU time per system of equations with respect to simultaneous right-hand sides.

$\text{BLOCKSIZE} = 5$ and that the saturation effect starts at $\text{BLOCKSIZE} = 20$, i.e., it makes no sense to use more than 20 simultaneous right hand sides. The gap between the graphs for SGI and ORIGIN is caused by the larger secondary cache on the ORIGIN so that the cache hit rate was already better for the original code on it. The smaller caches on the LINUX computer move the saturation point there to $\text{BLOCKSIZE} = 5$.

The speedups for the manually loop unrolling are significantly better than the normal speedups. Therefore, it is worth to take some time implementing it for special applications. The large speedup difference for the LINUX computer may be caused by some weakness in the optimization strategy of the gcc compiler.

The speedup is only a measure of improving a code on a certain computer. The main interesting question is the CPU time for solving a problem. Figure 5 presents the time for solving one system of equations (1) resulting from our test example.

Second, we test the Lanczos-Galerkin projection of Subsection 3.2 for several number of unknowns, but only on a LINUX computer. Now we assume $n = 2$ and $\zeta_k = 1,\ k = 0, \ldots, n$. The results are depicted in Figure 6 and Figure 7. Due to the special structure of the right-hand sides almost a factor of 2 in the CPU-time is gained. Clearly, this can not be reached for arbitrary right-hand sides. However the optimal AMG preconditioner provides a tool to be h-independent.

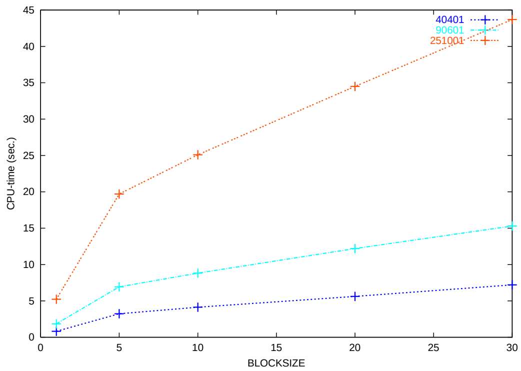

Figure 6: Performance gain with respect to Lanczos-Galerkin projection.

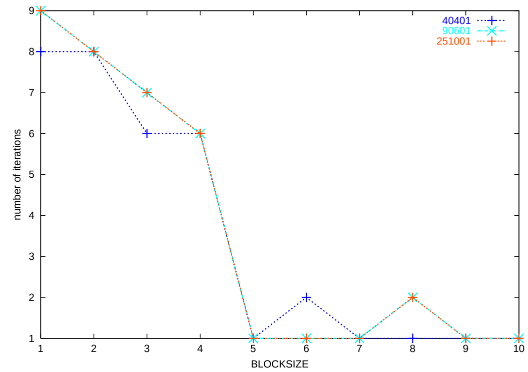

Figure 7: CPU time per system of equations with respect to Lanczos-Galerkin projection.

4.2 A 2D case study: multiple right-hand sides

In contrast to the previous section we test now a time-dependent problem but solve the sequence of arising equations with the technique of Subsection 3.2. Let us assume the discretization of a parabolic equation on the unit square with implicit time-stepping and appropriate boundary and initial conditions. Thus we get

$$(\Delta t \cdot K_h + M_h)\underline{u}_h^{i+1} = \underline{f}_h^i$$

with K_h the stiffness matrix, M_h the mass matrix and Δt denotes the time increment. The right-hand sides have the form

$$(\underline{f}_h^i)_j = \Delta t / N_h - (M_h \underline{u}_h^i)_j$$

where i denotes the time step. This problem is solved up to a relative accuracy $\epsilon = 10^{-6}$, and $\Delta t = 10^{-1}$. The required CPU-time and the number of iterations for the time-dependent

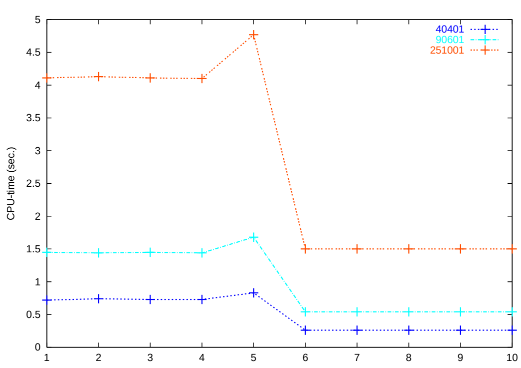

Figure 8: CPU-time for 10 time steps.

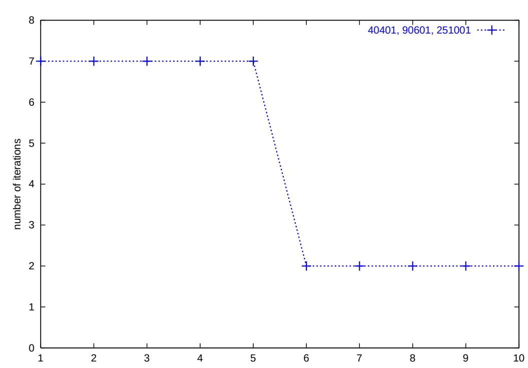

Figure 9: Number of iterations for 10 time steps.

problem using the Lanczos-Galerkin technique are depicted in Figure 8 and Figure 9. The overall CPU-time for the solution of the problem with 40401, 90601 and 251001 is 5.05 sec., 10.14 sec. and 28.73 sec., respectively. If we would solve the problem without the projection technique the required CPU-time is significantly larger, especially for growing N_h. Again the optimal AMG preconditioner is required to be h-independent.

4.3 Inverse source localization in the human brain

Finally we present a real life problem for which the technique of simultaneous right-hand sides (see Subsection 3.1) is applied. The usage of the techniques is reported on the inverse source localization in the human brain[1]. First numerical studies of the behavior of our AMG solver applied to this problem are presented in [20, 1]. The problem setting is as follows. In

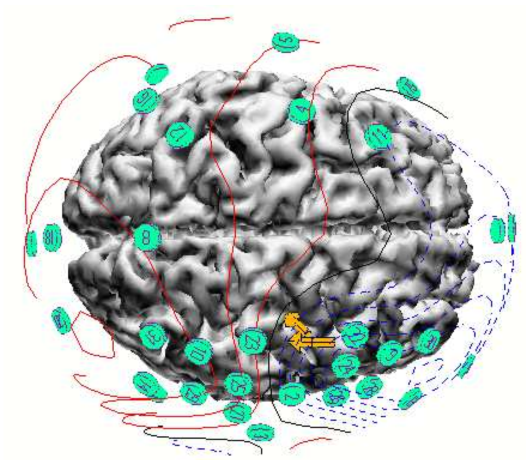

Figure 10: Dipole fit in the human brain.

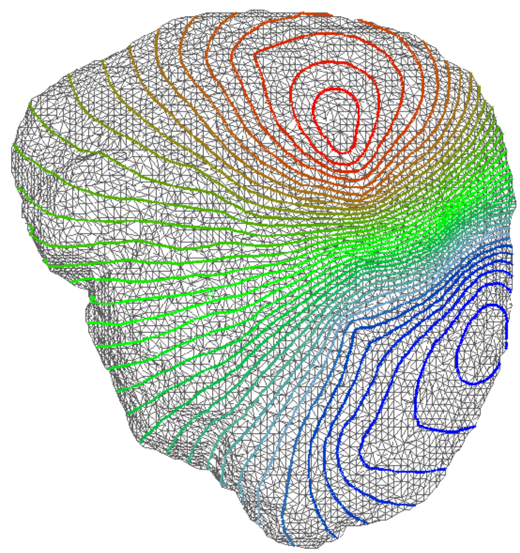

Figure 11: Solution of one dipole from a finite element discretization.

order to determine sources in the human brain from given measurements a system of linear equations have to solved for approximately 10.000 right-hand sides within a certain inverse toolbox. The underlying anisotropic potential equation is discretized by the FE-method. The human head is discretized by tetrahedral finite elements and its mesh contains 118.299 nodes. The resulting stiffness matrix has 1.815.991 non-zero elements.

In Figure 10 the principle structure of the problem and the computational domain is presented. In Figure 11 a typical solution is given for one specific right-hand side.

Let us assume we could solve one problem on a single processor computer in 50 seconds. From Figure 13 (and Figure 12) we see that we get a speedup of 2.3 for this real life problem. Equivalently one right-hand side can be solved in about 21 seconds.

Remark 4.1. *Since this special application has to be solved within less than 6 hours we require some more resources. As it was shown in [1] we can expect a speedup of 10 if we use 12 processors in parallel. Thus we can assume the solution of one linear system in 2.1 seconds if we use the simultaneous approach in parallel. Consequently the linear system with 10.000 right-hand sides can be solved within 6 hours.*

[1] Courtesy of C. Wolters, MPI Leipzig, Germany and the SimBio project.

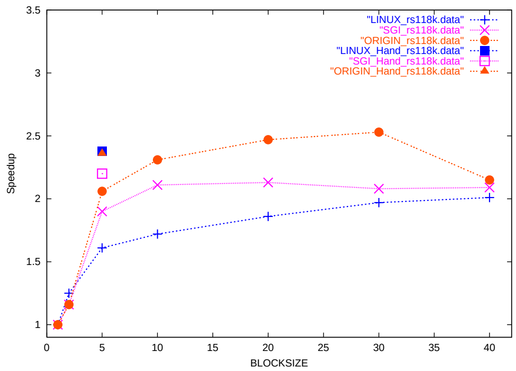

Figure 12: Performance gain with respect to simultaneous right-hand sides.

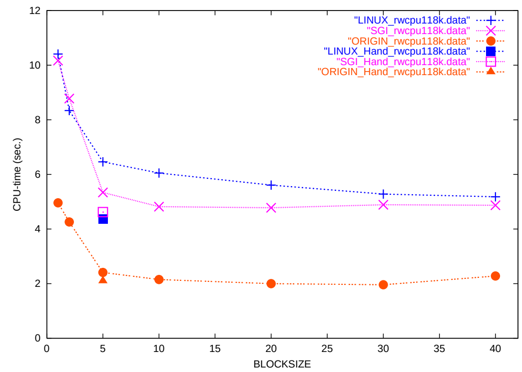

Figure 13: CPU time per system of equations with respect to simultaneous right-hand sides.

5 Conclusion and Further Remarks

In this paper we gave an overview on cache aware techniques for the solution of linear systems with multiple right-hand sides. We have shown that a good AMG preconditioner for the PCG method improves the projection techniques in a straight forward way. Moreover we can use cache effects in the case of multiple right-hand sides, if the vectors are known in advance.

A straight forward step is the parallelization of the proposed techniques in order to gain the required overall speedup for real life applications. Moreover we will concentrate on other problems which are of great importance in many practical applications [15].

References

[1] A. Anwander, M. Kuhn, S. Reitzinger, and C. Wolters, *A parallel algebraic multigrid solver for finite element method based source localization in the human brain*, Comp. Vis. Science (2002), accepted.

[2] W. L. Briggs, V. E. Henson, and S. McCormick, *A multigrid tutorial*, second ed., SIAM, 2000.

[3] T.F. Chan and W. L. Wan, *Analysis of projection methods for solving linear systems with multiple right-hand sides*, Tech. report, Department of Mathematics, UCLA, 1994, www.math.ucla.edu/~chan/papers.html.

[4] J. W. Demmel, J. R. Gilbert, and X. S. Lie, *SuperLU - User's Guide*, 1999, www.nersc.gov/~xiaoye/SuperLU/.

[5] Z. Dostal, V. Vondrak, and J. Rasmussen, *Implementation of iterative solvers in shape optimization*, Structural and Multidisciplinary Optimization (W. Gutkowski and Z. Mroz, eds.), vol. 1, 1997, pp. 443 – 448.

[6] C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, *Portable memory hierarchy techniques for PDE solvers, part I*, SIAM News **33** (2000), no. 5, 1, 8–9.

[7] ———, *Portable memory hierarchy techniques for PDE solvers, part II*, SIAM News **33** (2000), no. 6, 1, 10–11, 16.

[8] G. Haase, U. Langer, S. Reitzinger, and J. Schöberl, *A general approach to algebraic multigrid*, Tech. Report 00-33, Johannes Kepler University Linz, SFB "Numerical and Symbolic Scientific Computing", 2000.

[9] W. Hackbusch, *Multigrid methods and application*, Springer Verlag, Berlin, Heidelberg, New York, 1985.

[10] M. Jung and U. Langer, *Applications of multilevel methods to practical problems*, Surveys Math. Indust. **1** (1991), 217–257.

[11] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra and Appl. **29** (1980), 292 – 322.

[12] B. N. Parlett, *A new look at the Lanczos algorithm for solving symmetric systems of linear equations*, Linear Algebra and Appl. **29** (1980), 323 – 346.

[13] S. Reitzinger, *PEBBLES - User's Guide*, Johannes Kepler University Linz, SFB "Numerical and Symbolic Scientific Computing", 1999, www.sfb013.uni-linz.ac.at.

[14] ———, *Algebraic Multigrid Methods for Large Scale Finite Element Equations*, Schriften der Johannes-Kepler-Universität Linz, Reihe C - Technik und Naturwissenschaften, no. 36, Universitätsverlag Rudolf Trauner, 2001.

[15] S. Reitzinger, U. Schreiber, and U. van Rienen, *A general approach to algebraic multigrid*, Tech. Report 02-01, Johannes Kepler University Linz, SFB "Numerical and Symbolic Scientific Computing", 2002.

[16] J. W. Ruge and K. Stüben, *Algebraic multigrid (AMG)*, Multigrid Methods (S. McCormick, ed.), Frontiers in Applied Mathematics, vol. 5, SIAM, Philadelphia, 1986, pp. 73–130.

[17] Y. Saad, *On the Lanczos method for solving symmetric linear systems with several right-hand sides*, Math. Comp. **48** (1987), 651 – 662.

[18] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*, Academic Press, 2000.

[19] H. A. van der Vorst, *An iteration solution method for solving $f(a)x = b$, using krylow subspace information obtained for the symmetric positive definite matrix a*, Journal of Computational and Applied Mathematics **18** (1987), 249 – 263.

[20] C. Wolters, S. Reitzinger, A. Basermann, S. Burkhardt, U. Hartmann, F. Kruggel, and A. Anwander, *Improved tissue modeling and fast solver methods for high resolution FE-modeling in EEG/MEG-source localization*, Proc. of the 12th Int. Conf. of Biomagnetism (J. Nenonen, R.J. Ilmoniemi, and T. Katila, eds.), 2000, pp. 655 – 658.