Parallelization of the Multigrid Method on High Performance Computers

K. S. Kang
High Level Support Team (HLST)
Max-Planck-Institut für Plasmaphysik
Boltzmannstraße 2
D-85748 Garching bei München
Germany
kskang@ipp.mpg.de

August 16, 2010

Abstract

This article gives an introduction to the multigrid method. It discusses the issues involved and provides suggestions for parallelization of the multigrid method on HPC platforms. As an example problem, we consider cell-centered finite differences for the Poisson problem on a rectangular domain with uniform meshes. We consider two different intergrid transfer operators and investigate the convergence behavior of the multigrid method with these operators. In addition, different solvers are tested as a "lowest level" solver at the dip of the V-cycle of the multigrid algorithm. Furthermore, the scaling property of the multigrid method on massively parallel machines is investigated. We show that the multigrid algorithm has both good weak and strong scaling properties up to thousands of processors.

1 Introduction

The multigrid method is a well-known, fast and efficient algorithm to solve many classes of problems including linear elliptic, nonlinear elliptic, parabolic, and hyperbolic partial differential equations, and the set of Navier-Stokes and Magnetohydrodynamic (MHD) equations. [1, 2, 4, 5, 6, 7, 8, 11]. Although the multigrid method is complex to implement, researchers in many areas think of it as an essential algorithm. They apply it to their problems because the number of operations of the multigrid method depends on

Method	Storage	Flops
Gauss-Elimination (banded)	n^5	n^7
Gauss-Seidel Iteration	n^3	$n^5 \log n$
Optimal SOR	n^3	$n^4 \log n$
Conjugate Gradient Method	n^3	$n^{3.5} \log n$
Full Multigrid Method	n^3	n^3

Table 1: Order of required storage and flops for linear solvers.

the degree of freedom times the number of levels (log of the degree of freedom). We summarize the order of memory consumption and floating point operations per seconds (flops) of well-known linear solvers, applied to the linear Poisson problem on a uniform cubic domain, which has n^3 degrees of freedom and n vertices in each direction, in Table 1.

From many experiments it is well known that simple iterative methods effectively reduce high-frequency errors but have difficulties in handling low-frequency errors. In addition, the low-frequency structure of a function on finer grids is well approximated on coarser grids. Using these facts, we can reduce the low-frequency errors with the coarse grid error correction method which requires less work and storage. Combining simple iterative methods which reduce high-frequency errors and coarse grid error correction methods which reduce low-frequency errors is the basic idea of the two-grid method. Multigrid methods apply the two-grid method recursively from finest grids to coarsest grids as shown in Fig. 1.

1.1 Basic analysis

To implement and analyze the multigrid method, we have to consider two main parts of the multigrid algorithm, the smoothing operator and intergrid transfer operator, separately.

We can use any type of smoothing operator including, Richardson type (simplest one), Jacobi iteration, Gauss-Seidel iteration, Kancmarz iteration, and incomplete LU decomposition (probably the most complex one). Naturally, smoothing operators affect the performance of the multigrid method, but one can have a good performance with the Richardson iteration. To analyze the multigrid method, many authors use certain assumptions on smoothing operators which, in turn, allow them to be replaced by simpler smoothing operators.

The other important operators are the intergrid transfer operators, the

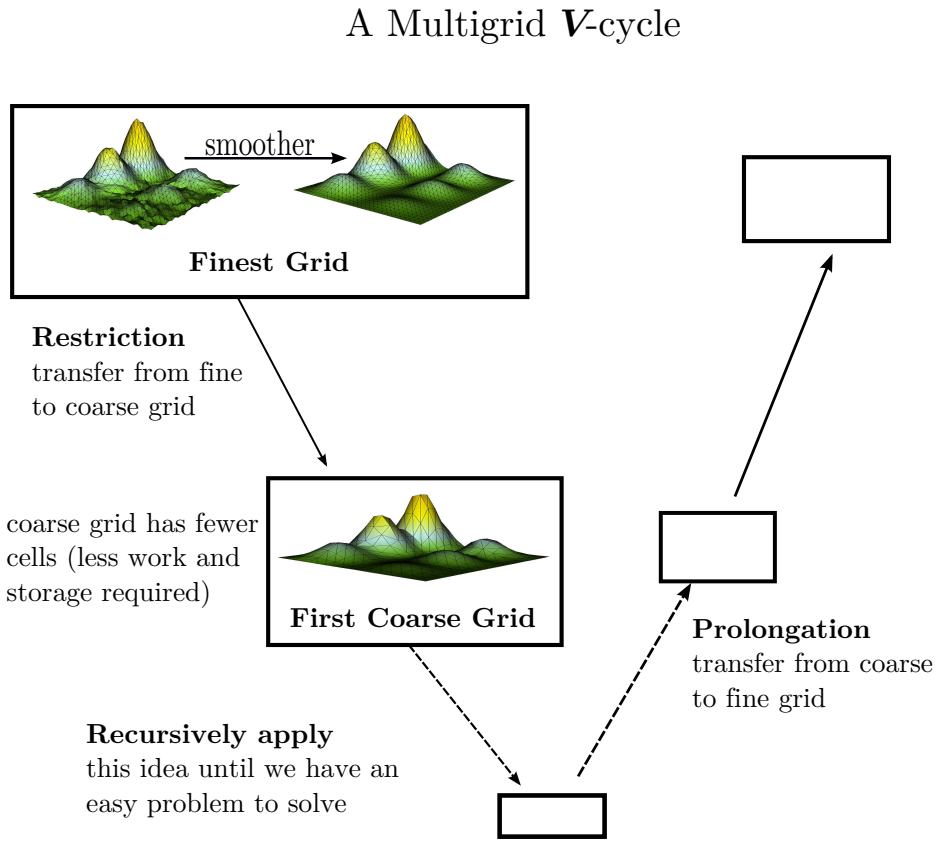


Figure 1: Basic idea of the multigrid method

prolongation and restriction operators. These operators depend on the geometry and functional spaces, i.e., conforming or nonconforming method, discretization method; e.g., finite element method, finite difference method, finite volume method, covolume method, and cell-centered scheme or vertex-centered scheme, etc. There are natural intergrid transfer operators for conforming and inherited functional spaces. For the nonconforming method or non-inherited functional spaces, we can define the intergrid transfer operators in several different ways, e.g., geometrical based for the finite element method, control volume based for the finite volume method.

Even if we do not have any geometric information, we can define the intergrid transfer operators according to the properties of a linear operator, i.e., the algebraic multigrid method.

In Section 2.2 we will discuss the properties of the chosen intergrid transfer operators in detail.

1.2 Parallelization Issues

To get a good performance when parallelizing the program, we need to have a good load balance over all cores (processors).

In general, the ratio of communication to computation on a coarse level grid is larger than on a fine level grid. Since the multigrid method works on both coarse and fine grid levels, we need to consider, in detail, the balance between computational work and communication. Usually, the multigrid method requires more work on coarse problems in comparison to other iterative or direct methods.

In general, the balance between computation and communication is highly dependent on machine architecture and problem sizes. Hence, we need to determine the level at which we need to stop coarsening according to the number of cores on each machine and problem size.

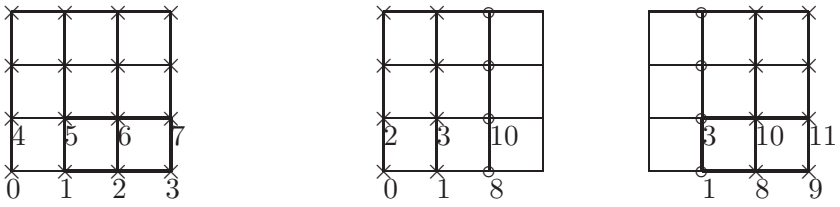
It is well known that the number of iterations required by the multigrid method to get to a well converged solution depends only on the number of levels. In addition, many iterative methods with the exception of the Richardson and the Jacobi method are hard to parallelize or have a dependence on the number of cores they are executed on. As an example, we consider the Gauss-Seidel iteration method which is a preferred smoothing operator. We consider a simple mesh and its numbering on a single core and two cores in Fig. 2. To compute the updated value at point 2 in case of a single core (point 8 on two cores), we use the updated value at point 1 and the old values at the other points (3,5,6,7). On two cores we use the old values at the points (1,3,9,10,11) on PE1 otherwise PE1 would have to wait until it receives the updated value of point 1 from PE0. However, this would be not feasible.

The issues about parallelization of the multigrid method and the choice of the optimal coarsest level will be handled in Section 2.7.

2 The cell-centered finite difference multigrid method

2.1 Discretization and boundary condition

The cell-centered finite difference (CCFD) is one of the most popular methods for solving second-order elliptic boundary value problems numerically



(a) Single core (b) Two cores

Figure 2: The numbering on a single core and two cores (∘: ghost points)

[9, 10]. CCFD is a finite volume type method and has been commonly used by engineers and scientists because of its simplicity and local conservation property.

We consider the following model problem:

$$-\nabla \cdot p\nabla u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \partial\Omega, \tag{2}$$

where Ω is the unit square. For $k = 1, 2, \ldots, J$, divide Ω uniformly into $n \times n$ axiparallel subsquares, where $n = 2^k$. Then the mesh sizes Δx and Δy are $h = h_k = 1/2^k$. Such subdivisions are denoted by $\{\mathfrak{E}_k\}$, and each subsquare in $\{\mathfrak{E}_k\}$ is called a cell and denoted by E_{ij}^k, $i, j = 1, 2, \ldots, 2^k$. Note that cell E_{ij}^k is centered at the point $(x_i, y_j) = ((i - \frac{1}{2})h, (j - \frac{1}{2})h)$ as given in Fig. 3. For $k = 1, 2, \ldots, J$, let V_k denote the space of functions that are piecewise constant on each cell. Integrating Eq. (1) by parts on each cell and replacing the normal derivative on the edges by difference quotient of a function u in V_k, we derive the finite difference equations as follows:

$$\begin{aligned} &- p_{i-\frac{1}{2},j}(u_{i-1,j} - u_{i,j}) - p_{i+\frac{1}{2},j}(u_{i+1,j} - u_{i,j}) \\ &- p_{i,j-\frac{1}{2}}(u_{i,j-1} - u_{i,j}) - p_{i,j+\frac{1}{2}}(u_{i,j+1} - u_{i,j}) = f_{i,j}h^2, \end{aligned} \tag{3}$$

where $p_{i-\frac{1}{2},j} = p(x_{i-\frac{1}{2}}, y_j)$, etc.

Because in the cell-centered scheme there are no grid points on the boundary, the treatment of boundary conditions is different from the vertex-centered case. To impose the boundary condition in y-direction, we put a ghost point b outside the domain shown in Fig. 3. Next, we set the functional value at b, so that the function u satisfies the boundary condition on

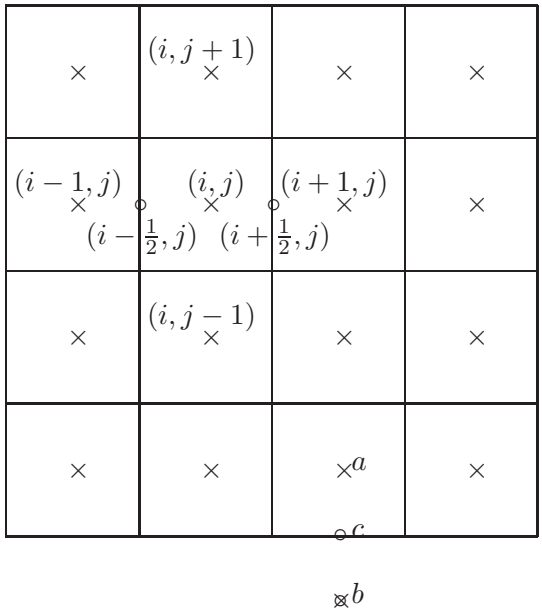


Figure 3: Cell-centered finite difference method on a quadratic domain.

the boundary point c, i.e.,

$$u(c) = \frac{u(a) + u(b)}{2} = 0, \qquad \text{with } u(b) = -u(a)$$

in case of the Dirichlet boundary condition or

$$\left.\frac{\partial u}{\partial n}\right|_c = \frac{u(a) - u(b)}{h_y} = 0, \quad \text{with } u(b) = u(a)$$

in case of the Neumann boundary condition.

As a model problem, we consider the Poisson problem ($p = 1$) with the periodic boundary condition in x-direction and the Dirichlet boundary condition in y-direction. As smoothing operator, we use the Red-Black Gauss-Seidel iteration because it shows relatively good performance and does not depend on the number of cores for CCFD of the Poisson problem. The Red-Black Gauss-Seidel iteration divides the domain in red and black points as shown in Fig. 4. In each sweep the values of one color are exclusively updated using the values of the other color. Hence, there is no dependence on the order of updating during a sweep. This can be also seen

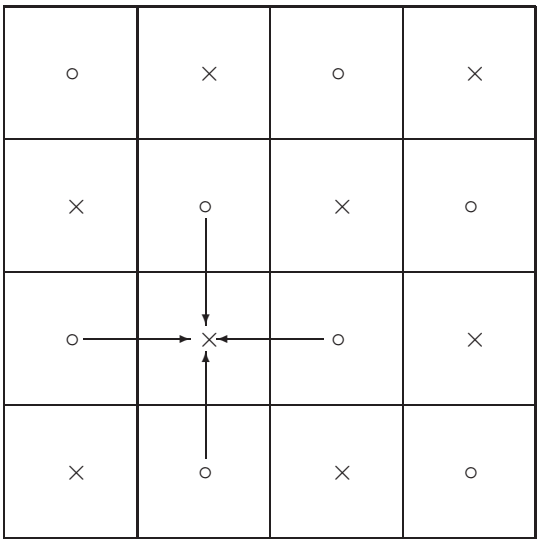


Figure 4: Red-Black Gauss-Seidel iteration for the cell-centered finite difference method on a quadratic domain. ($\times$: Black $\circ$: Red)

in the corresponding structure of the matrix which has nonzero values only on the diagonal and its neighborhood off-diagonal belonging to the other color. However, the Red-Black Gauss-Seidel iteration does not have this property for general problems (e.g. $\partial_x \partial_y p \neq 0$).

2.2 Intergrid transfer operators

We have to define two intergrid transfer operators, one is the restriction, i.e., fine-to-coarse operator and the other the prolongation, i.e., coarse-to-fine operator. In addition, these two intergrid transfer operators are the adjoint operator of each other [11].

Here, we consider two different pairs of the intergrid transfer operators, one preserves piecewise constant functions (zeroth-order) and the other preserves piecewise bilinear functions (first-order). To define the intergrid transfer operator, we use the notation given in Fig. 5. The former operator is simple and needs less work, but cannot be used for some problems that require more regularities.

The expressions to compute the restriction (R_0, fine-to-coarse) and prolongation (P_0, coarse-to-fine) operators for piecewise constant are given in

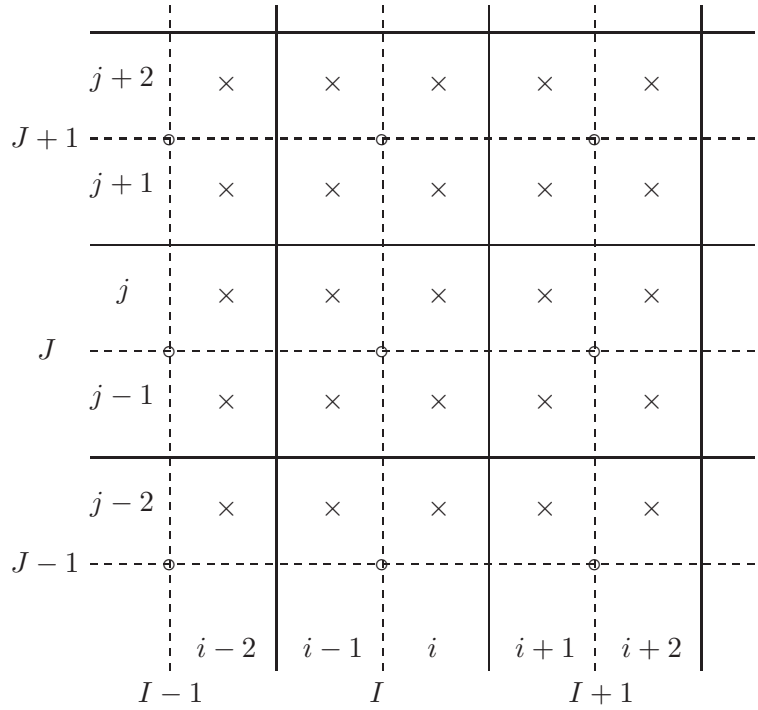


Figure 5: Intergrid transfer operators. $\times$: fine grid cells, $\circ$: coarse grid cells

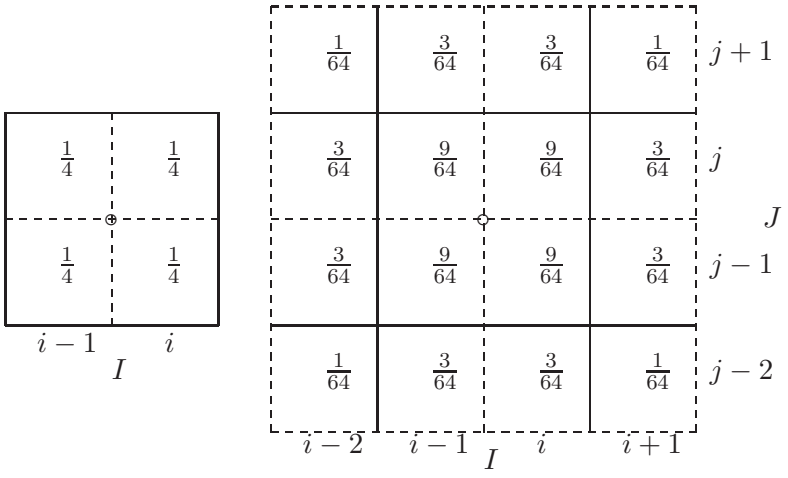


Figure 6: Contributions of fine grid cells of fine-to-coarse (restriction) transfer operators to the coarse grid cells ∘ (compare with Fig. 5).

Eqs. (4) and (5).

$$R_0[u(I,J)] = \frac{u(i-1,j-1) + u(i,j-1) + u(i-1,j) + u(i,j)}{4} \tag{4}$$

and

$$\begin{aligned} P_0[u(i,j)] &= u(I,J) \\ P_0[u(i-1,j)] &= u(I,J) \\ P_0[u(i,j-1)] &= u(I,J) \\ P_0[u(i-1,j-1)] &= u(I,J). \end{aligned} \tag{5}$$

The expressions to compute the restriction (R_1) and the prolongation (P_1) operators for piecewise bilinear function are given in Eqs. (6) and (7).

$$\begin{aligned} R_1[u(I,J)] = \frac{1}{64}\Big\{ & 9\big[u(i-1,j-1) + u(i,j-1) + u(i-1,j) + u(i,j)\big] \\ & + 3\big[u(i-2,j) + u(i-2,j-1) + u(i+1,j) + u(i+1,j-1) \\ & + u(i,j-2) + (i-1,j-2) + u(i,j+1) + u(i-1,j+1)\big] \\ & + u(i-2,j-2) + u(i-2,j+1) + u(i+1,j-2) \\ & + u(i+1,j+1)\Big\} \end{aligned} \tag{6}$$

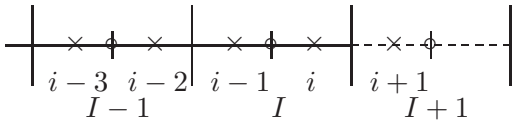


Figure 7: Boundary treatment of the first-order intergrid transfer operator, $\times$: fine grid cells, $\circ$: coarse grid cells

and

$$\begin{aligned}
P_1[u(i,j)] &= \frac{9u(I,J) + u(I+1,J+1) + 3\big[u(I,J+1) + u(I+1,J)\big]}{16} \\
P_1[u(i-1,j)] &= \frac{9u(I,J) + u(I-1,J+1) + 3\big[u(I-,J) + u(I,J+1)\big]}{16} \\
P_1[u(i,j-1)] &= \frac{9u(I,J) + u(I+1,J-1) + 3\big[u(I+1,J) + u(I,J-1)\big]}{16} \\
P_1[u(i-1,j-1)] &= \frac{9u(I,J) + u(I-1,J-1) + 3\big[u(I-1,J) + u(I,J-1)\big]}{16}.
\end{aligned} \tag{7}$$

In Fig. 6, we show the contributions of all associated fine cells to a coarse cell for R_0 and R_1.

To define the first-order intergrid transfer operators (R_1 and P_1), we need some values which are located outside the domain ($I+1$ and $i+1$ in Fig. 7). We can define the values outside the domain by using the boundary conditions, e.g., homogeneous Dirichlet or Neumann boundary conditions.

2.3 Numerical experiments with intergrid transfer operators

We have tested the multigrid method on a uniform square grid on a unit square domain $[0,1] \times [0,1]$. In each direction, we have 2^l grid points where l is a certain "level", the "highest level" is the level on which we want to achieve the solution and the "lowest level" is the level on which we solve exactly. We use a V-cycle multigrid method with one time pre-smoothing, two times post-smoothing and the Red-Black Gauss-Seidel iteration as smoothing iterations. As test problem, we choose a sine source function, i.e., $f(x,y) = \sin(20\pi x)\sin(30\pi y)$ in Eq. (1) and start the multigrid algorithm with a zero value initial solution.

We have chosen test cases with varying values of the "highest level" and the "lowest level". However, we have found that the choice of the "lowest

	R_0 P_0		R_1 P_1	
"highest level"	# of iter.	error red.	# of iter.	error red.
6	9	0.0605	9	0.0586
7	9	0.0650	11	0.0939
8	10	0.0795	12	0.1386
9	12	0.1032	13	0.1507
10	13	0.1261	13	0.1539
11	14	0.1514	13	0.1547
12	16	0.1779	13	0.1549
13	17	0.2082	13	0.1550
14	18	0.2444	13	0.1555

Table 2: The number of iterations and the average error reduction factor of the multigrid method.

level" has virtually no effect on the number of iterations. In Table 2, we report the number of iterations to get the required residual error (less than 10^{-10} times the initial residual error) and the average error reduction factor, i.e., the ratio of post-error to pre-error according to the "highest level" for both intergrid transfer operators. We used level one as the "lowest level" by default. Only in the case given in the last row where the "highest level" is 14, a "lowest level" of two has been used.

The results listed in Table 2 show that the error reduction factor and the number of iterations are clearly dependent on the "highest level" in case of the zeroth-order intergrid transfer operator. Instead, in case of the first-order intergrid transfer operator there is hardly any influence left when the "highest level" is further increased after the "highest level" of nine is reached. Thus the zeroth-order intergrid transfer operator shows a better performance for small problems, whereas the first-order intergrid transfer operator shows a better performance for larger problems.

2.4 "Lowest level" approximations

In general, the ratio of communication to computation on coarse grid levels is larger than on fine grid levels. We need to target a balance between computation and communication which is highly dependent on machine architecture and problem sizes.

There are many considerations to improve the performance of the multigrid method, for example, selection of coarsest approximation, using only a

small number of cores in parallel, stop of coarsening on a certain level. We consider four popular solvers which will be used as coarsest problem solvers on the "lowest level"; the Red-Black Gauss-Seidel method (Relaxation), a sparse direct solver (IBM WSMP) [3], a dense direct solver (LAPACK and ScaLAPACK for parallel usage), and the Conjugate Gradient Method (CGM).

First, we compute the solution time using the above mentioned solvers and the multigrid method on a single core for several problem sizes. In the multigrid method we use a quite small 2×2 grid as "lowest level" to avoid a strong influence of the "lowest level" solver which is actually the CGM. In Sec. 2.7.1 it will be shown which choice of the "lowest level" is optimal.

The direct solvers (WSMP and LAPACK) spend typically more time at the Choleski factorization which is performed only once for a given matrix, than at the back substitution which is performed at every iteration. So it is adequate to plot only the back substitution solving times of the direct solvers [denoted as WSMP(solving) and LAPACK(solving)]. As a sparse solver, WSMP performs the Choleski factorization in two consecutive steps. One is the symbolic factorization which depends only on the matrix structure and the other is the numerical factorization. This consecutive approach has its advantages when the structure of the matrix persists and only its numerical values are changing at each time step, i.e., one has to perform the numerical factorization and solving steps on each time step. Hence, we report the times for the numerical factorization and solving for WSMP as WSMP(chol).

We plot the graph of the solution time vs. the degree of freedom (DoF), i.e., the problem size in Fig. 8. We run the code on an IBM p575 Power6 system. The graph shows the solution time of each solver together with the requirements of using the multigrid method to solve different problem sizes. For the Relaxation method, the solving time of the problem which has 2^{12} DoF is faster than of 2^{10} DoF. This is because the number of iterations for 2^{12} DoF is less than 20, but more than 100 for 2^{10} DoF. This phenomenon sometimes happens for iterative methods on small problems. For the smallest problem size (2^8 DoF), CGM, Relaxation, LAPACK(solving) are faster than the other solvers. CGM is the fastest solver for relatively small problem sizes (from 2^{10} to 2^{16} DoF). For large problem sizes, multigrid and WSMP(solving) give similar good results.

This result suggests that we might get the fastest solution time when we use the multigrid method with the single-core version of the CGM, Relaxation, and LAPACK(solving) as "lowest level" solver on grids with 2^8 DoF and CGM as the "lowest level" solver on coarsest grids with 2^{10} to 2^{16} DoF.

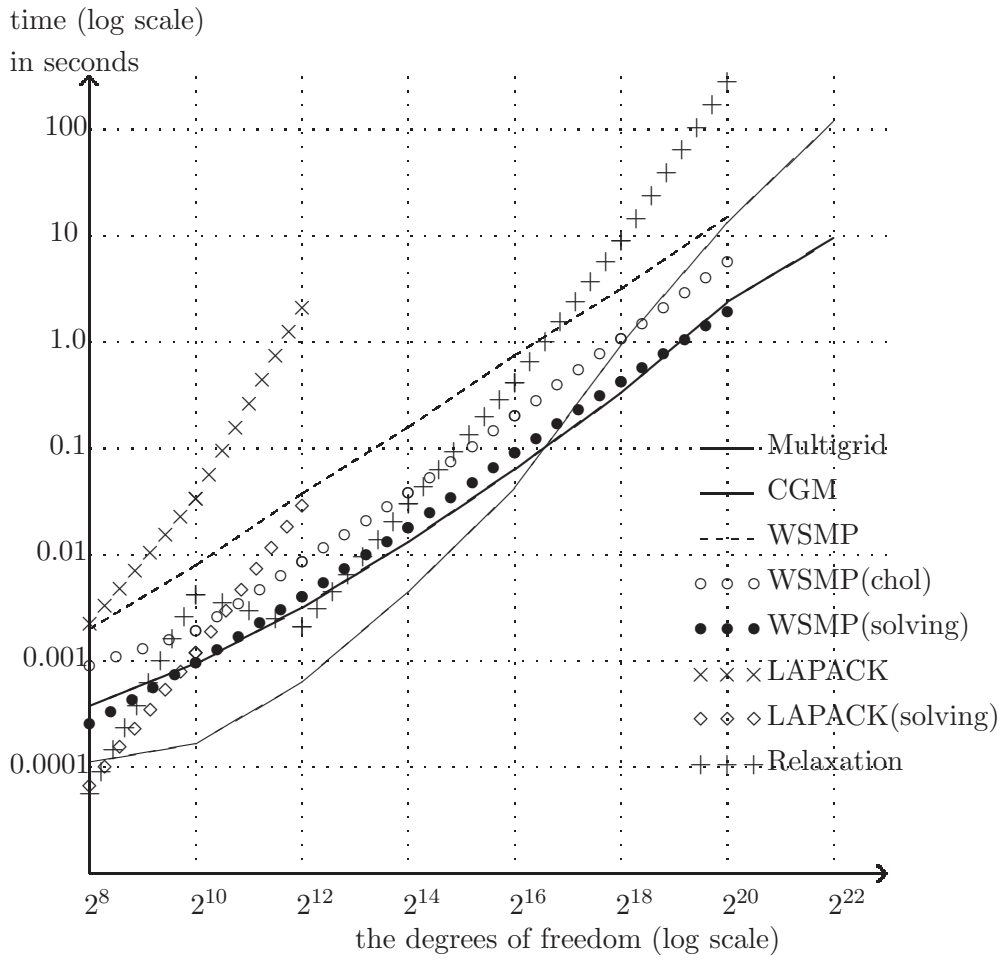


Figure 8: Solution times of different solvers on a single core of an IBM p575 Power6 system.

2.5 Parallelization of the different "lowest level" solvers

Next, we investigate the behavior of the different solvers mentioned in Sec. 2.4 when the number of cores is increased. It is well known that there exists a maximum number of cores which shows the best performance to solve a given problem with a certain size. Using larger numbers of cores than this maximum number may even increase the solution time. This investigation gives hints to determine what level is optimal as "lowest level" and what method is optimal as "lowest level" approximation for a given number of cores. These results depend strongly on the given HPC machine.

In Table 3, we report the results measured on the VIP machine at RZG. VIP is an IBM p575 Power6 system with 205 compute nodes and connected by a fast 8-link InfiniBand network. Each node of VIP has 32 processors with just a single core per processor, i.e., 32 cores. We report the solution times for Relaxation, CGM, LAPACK/ScaLAPACK (for more than one core we use the latter) as 'LAP', LAPACK(solving) as '(S)', WSMP (as 'WSMP' and 'WSMS'), WSMP(Chol) as 'C+S', and WSMP(solving) as '(S)' (compare with Fig. 8). The WSMP library uses two different approaches for parallelization, the message passing interface (MPI) and a multithreaded version. We use the multithreaded version on one node with 32 threads. We report the results of both versions, WSMP for MPI and WSMS for the multithreaded version.

The full-matrix solver (LAPACK) could not be used for the larger problems due to memory limitations. Hence, we denote "–" in Table 3 for LAPACK and the problem size of 2^{14} DoF. Due to these limitations, we did not solve the larger problems (more than 2^{16} DoF) with the full-matrix direct solver.

For each solver and problem size we emphasized the optimal solution time on a certain number of cores in boldface. Also, we emphasized superior solution times of all solvers and all number of cores for a fixed problem size by an underline.

From these results, we can derive the optimal number of cores which has the best performance for a given problem size and solver. In addition, the results show that CGM is the best parallel solver for problem sizes in the range of 2^{10} to 2^{16} DoF. The optimal number of cores increases as the problem size increases.

	level	# of cores/threads					
	(DoF)	1	2	4	8	16	32
Rel	$5(2^{10})$	0.00413	**0.00075**	0.00115	0.00364	0.00157	0.00461
	$6(2^{12})$	0.00210	0.00442	**0.00135**	0.00371	0.00404	0.00431
	$7(2^{14})$	0.02865	0.01430	0.00892	0.00953	0.00805	**0.00560**
	$8(2^{16})$	0.42355	0.16906	0.07925	0.05286	0.03567	**0.02764**
CGM	$5(2^{10})$	**0.00017**	0.00033	0.00042	0.00340	0.00114	0.00395
	$6(2^{12})$	**0.00064**	0.00079	0.00081	0.00345	0.00394	0.00214
	$7(2^{14})$	0.00441	0.00399	**0.00280**	0.00540	0.00522	0.00576
	$8(2^{16})$	0.04129	0.02151	0.01301	0.01102	**0.00761**	0.01000
LAP	$5(2^{10})$	0.03341	0.07861	0.03633	0.02172	**0.01645**	0.04826
(S)		0.00117	0.00197	0.00121	**0.00108**	0.00122	0.02774
	$6(2^{12})$	2.07746	7.88744	1.90999	0.72824	0.32742	**0.18356**
(S)		0.02875	0.04127	0.02900	0.01668	0.00877	**0.00722**
	$7(2^{14})$	–	–	148.769	50.5465	21.8205	**8.31443**
(S)		–	–	0.49466	0.29237	0.20640	**0.11590**
WSMP	$5(2^{10})$	**0.00788**	0.01092	0.01036	0.01048	0.01103	0.01195
C+S		0.00190	0.00098	**0.00082**	0.00087	0.00126	0.00204
(S)		0.00093	0.00044	**0.00041**	0.00048	0.00075	0.00120
	$6(2^{12})$	0.03645	0.02488	0.02047	**0.01912**	0.01940	0.02044
C+S		0.00840	0.00374	0.00264	**0.00213**	0.00238	0.00303
(S)		0.00386	0.00154	0.00114	**0.00106**	0.00137	0.00181
	$7(2^{14})$	0.15784	0.09120	0.06678	0.05571	0.05427	**0.05182**
C+S		0.03910	0.01689	0.01081	0.00752	**0.00728**	0.00740
(S)		0.01754	0.00653	0.00416	**0.00339**	0.00410	0.00467
	$8(2^{16})$	0.73409	0.39989	0.26952	0.21257	0.19183	**0.18255**
C+S		0.19862	0.08535	0.04909	0.03225	0.02688	**0.02420**
(S)		0.08802	0.03150	0.01842	**0.01325**	0.01411	0.01524
WSMS	$5(2^{10})$	0.00788	**0.00657**	0.00784	0.00970	0.01181	0.01234
C+S		0.00190	**0.00163**	0.00279	0.00429	0.00608	0.00636
(S)		0.00093	**0.00085**	0.00180	0.00297	0.00436	0.00447
	$6(2^{12})$	0.03645	**0.02685**	0.02769	0.03091	0.03407	0.03600
C+S		0.00840	**0.00624**	0.00710	0.00962	0.01232	0.01278
(S)		0.00386	**0.00294**	0.00357	0.00553	0.00757	0.00753
	$7(2^{14})$	0.15784	**0.11251**	0.11749	0.12076	0.13580	0.13274
C+S		0.03910	**0.02801**	0.03143	0.03652	0.05074	0.04983
(S)		0.01754	**0.01249**	0.01293	0.01440	0.01827	0.01653
	$8(2^{16})$	0.73409	**0.51627**	0.52712	0.53850	0.57903	0.69273
C+S		0.19862	**0.14491**	0.15318	0.16571	0.20477	0.31416
(S)		0.08802	0.06079	**0.06015**	0.06671	0.07602	0.11003

Table 3: Solution times, in seconds, of solvers depending on the number of cores for several problem sizes. (Optimal core number in boldface for each solver and underlined over all solvers on a fixed problem size.)

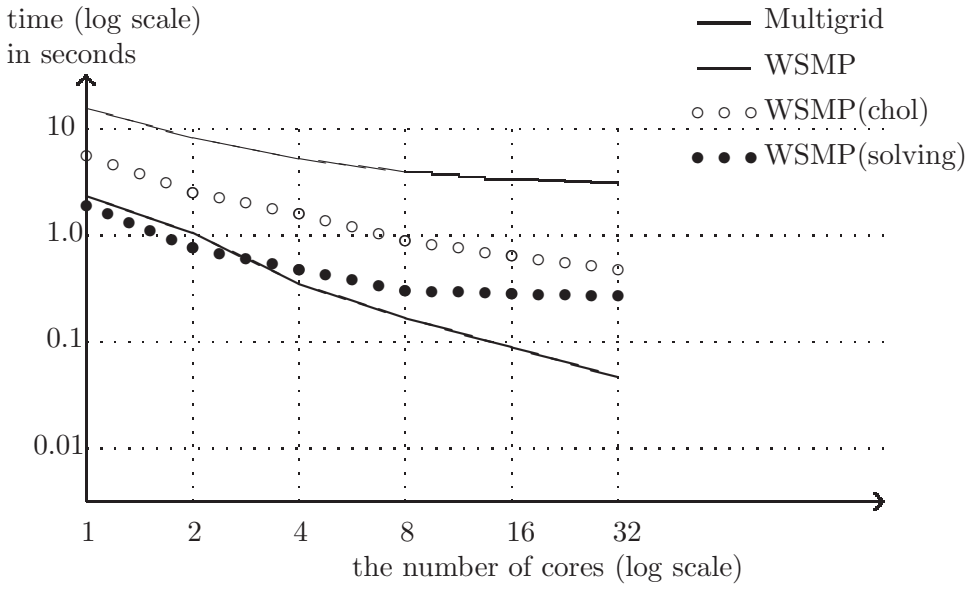


Figure 9: Comparison of solution times of WSMP and the multigrid method as a function of the number of cores on VIP to solve a problem with 2^{20} DoF within one node.

2.6 Parallelization for WSMP and the multigrid method

Next, we investigate the effect of parallelization for WSMP and the multigrid method with CGM as "lowest level" solver. We solve the problem with 2^{20} DoF on two different machines, VIP at RGZ and HPC-FF at JSC. HPC-FF is an Intel Nehalem system with 1080 compute nodes, two Intel Xeon X5570 (Nehalem-EP) quad-core processors per node and connected by an Infiniband Mellanox ConnectX QDR HCA. From now on, we use 'core' instead of 'processor' for VIP. We report the solution times in Figs. 9 and 10 and the speed up relative to the solution times on a single-core in Figs. 11 and 12. The runs on VIP had been restricted to the maximum number of 32 cores on a node to prevent internode communication. Instead, distributing our code over different nodes on HPC-FF turned out not to be critical. We ran our code on up to 64 cores, i.e. eight nodes, on HPC-FF. A vertical dashed line separates single and multiple node usage in Figs. 10 and 12.

In Figs. 9–12, we plot three different graphs for the sparse direct solver, i.e., WSMP for the total solution time, WSMP(chol) for the numerical factorization and back substitution time, and WSMP(solving) for the back

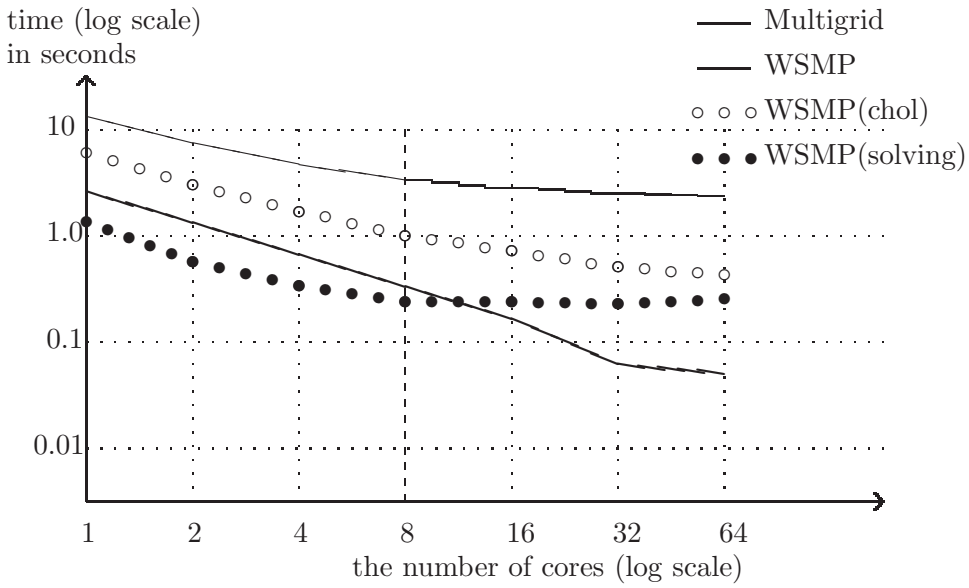


Figure 10: Comparison of solution times of WSMP and the multigrid method as a function of the number of cores on HPC-FF for a problem size of 2^{20} DoF. (Dashed vertical line separates single and multiple node usage)

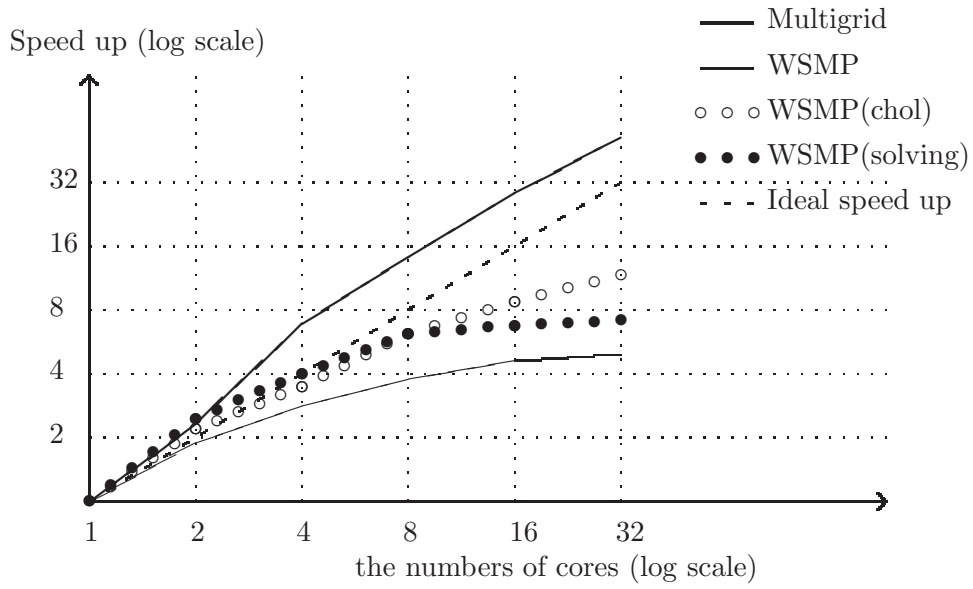


Figure 11: Speed up of solution times of WSMP and the multigrid method as a function of the number of cores on VIP for a problem size of 2^{20} DoF.

substitution time only.

The graphs show that WSMP(solving) is faster than the multigrid method on a small number of cores, less than four cores in case of VIP and less than sixteen cores in case of HPC-FF. For a larger numbers of cores, the multigrid method performs better than WSMP due to better scaling properties. The results of the multigrid method show that the solution time on VIP declines up to 32 cores and stays constant subsequently for a higher number of cores. The solution times on HPC-FF declines even up to 64 cores.

2.7 Scalability of the multigrid method

In this Subsection, we investigate the strong and weak scaling behavior of the multigrid method. Strong scaling means to run a fixed problem size on an increasing number of cores. Weak scaling means to increase the problem size simultaneously as the number of cores is increased, i.e., the number of DoF per core is fixed.

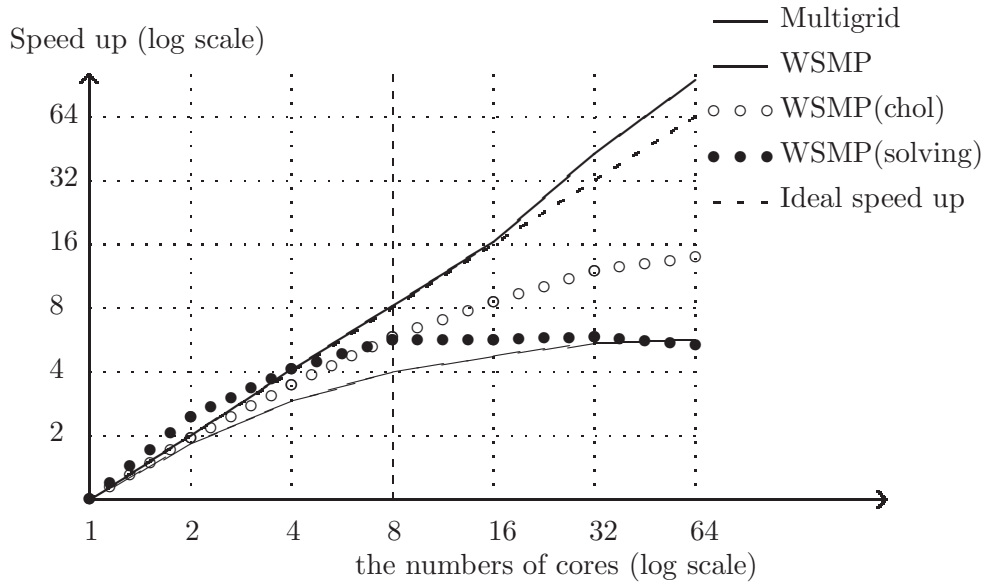


Figure 12: Speed up of solution times of WSMP and the multigrid method as a function of the number of cores on HPC-FF for a problem size of 2^{20} DoF. (Dashed vertical line distinguishes between single and multiple node usage)

2.7.1 Strong scaling of the multigrid method

We investigate the strong scaling behavior of the multigrid method on VIP and HPC-FF. We measure the solution time of the multigrid method for a larger problem size which has 2^{24} DoF ($l = 12$). We distinguish between the multi- and single-core version on the single-core levels. To construct the single-core version for calculating the single-core levels, at a certain level we gather the data from all the cores on each core (called "gathering level"). In our case either on level six or level seven. From that level on, we proceed on each core simultaneously with the serial multigrid algorithm. Finally, the algorithm terminates when it reaches the "lowest level" and solves on that level with one of the proposed methods for the "lowest level" solution. Then we collect the part of the solution when ascending the levels, starting with the serial and after we have reached the "gathering level" again with the parallel multigrid method. We illustrate the implementation of the whole scheme in Fig. 13.

We consider the serial multigrid version on the single-core levels for the following reasons. First of all, the degree of freedom at certain levels may be less than the number of cores. In those cases, one has to switch to the serial version for calculating the single-core levels. And secondly, the execution time of the serial multigrid algorithm can be smaller than in the parallel version, even if there is some additional communication overhead for gathering the data. This is e.g., the case for the CGM solver when using problem sizes between 2^{10} and 2^{12} DoF (see Table 3).

We use the first-order intergrid transfer operator and three different "lowest level" solvers: CGM, Relaxation, and WSMP. For the WSMP "lowest level" solver, we run the Cholesky factorization (the symbolic and numerical factorization) only once and run the back substitution as many times as needed for the iterations of the multigrid method.

We report the results measured on VIP in Table 4 and on HPC-FF in Table 5. We tested three "lowest levels" (4, 5, 6) and different numbers of cores ranging from 2 to 256 on VIP and from 4 to 512 on HPC-FF. Due to the low accessibility of batch queues with 512 or more cores on VIP, we restricted the maximum number of cores to 256. Instead, we used up to 512 cores on HPC-FF. Larger number of cores are not feasible as a problem size of 2^{24} DoF turns out to be too small (less than 0.1 sec) for such runs.

Depending on the problem size the maximum number of cores which can be used with the parallel WSMP version is limited. In Tables 4 and 5, we denote '$**$' in cases where the number of cores exceeds this maximum limit. If the number of cores exceeds the number of degrees of freedom, we are also

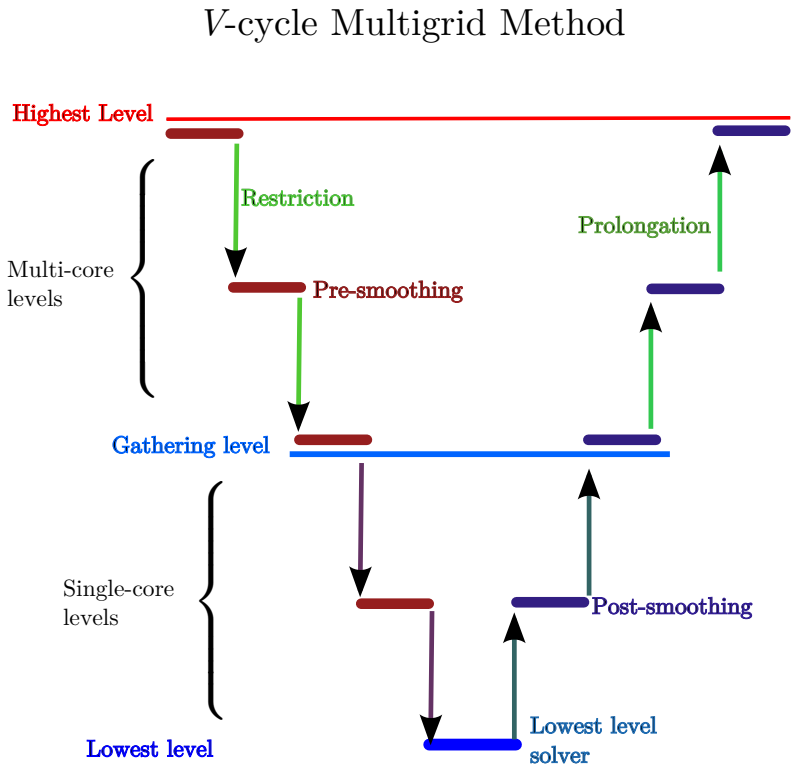


Figure 13: A schematic view of the V-cycle multigrid method which starts as a parallel multigrid implementation, then after passing the gathering level converts to a single-core multigrid version and finally ends up in one of the possible choices for the "lowest level" solver.

not able to use multiple cores. Such cases are denoted with '–' in Table 5. In Table 4 and 5, the numbers in the first column give the "lowest levels" l. Then the degree of freedom of the "lowest level" is 2^{2l}. The number in $(\cdot)$ represents the "gathering level" at which we gather the distributed data on each core and continue with the single-core multigrid version.

In Table 4 and 5, we emphasize the fastest solution time for a certain "lowest level" solver using a fixed number of cores given in boldface and the fastest solution time of all "lowest level" solvers marked by an underline. In addition, we highlight in red all solution times which match within 1% the fastest solution time over all "lowest level" solvers. As a reference, we report the optimal solution times plus 1% given in seconds in Table 6. It becomes

L\ P	2	4	8	16	32	64	128	256
4	20.339	9.974	4.729	2.633	1.104	0.397	**0.196**	0.114
5	20.048	9.955	4.814	2.631	1.120	**0.390**	0.209	0.113
6	20.215	10.121	4.766	2.616	1.124	0.396	0.203	**0.106**
4(6)	**20.058**	10.474	**4.705**	**2.615**	**1.074**	0.402	0.200	0.112
5(6)	20.247	10.988	4.710	2.622	1.116	0.405	0.205	0.123
6(6)	20.139	10.000	4.780	2.658	1.123	0.438	0.241	0.153
4(7)	20.245	**9.943**	4.721	2.700	1.110	0.435	0.228	0.136
5(7)	20.453	10.007	4.767	2.627	1.116	0.439	0.233	0.154
6(7)	20.176	10.060	4.774	2.675	1.155	0.475	0.291	0.184

(a) Conjugate Gradient Method as "lowest level" solver.

L\ P	2	4	8	16	32	64	128	256
4	20.180	9.923	4.716	2.823	1.074	0.395	0.199	0.117
5	20.124	9.931	**4.699**	2.635	**1.072**	0.399	0.199	**0.109**
6	**20.033**	9.917	4.709	2.666	1.079	**0.394**	**0.196**	0.112
4(6)	20.161	9.926	4.711	**2.633**	1.086	0.397	0.203	0.116
5(6)	20.192	**9.911**	4.706	2.634	1.082	0.399	0.205	0.112
6(6)	20.236	9.976	4.811	2.633	1.087	0.411	0.216	0.137
4(7)	20.587	10.032	4.753	2.656	1.121	0.435	0.228	0.140
5(7)	20.187	9.937	4.731	2.670	1.116	0.438	0.244	0.140
6(7)	20.132	9.991	4.744	2.666	1.119	0.446	0.252	0.147

(b) Gauss-Seidel Relaxation method as "lowest level" solver.

L\ P	2	4	8	16	32	64	128	256
4	20.009	**9.943**	**4.753**	2.652	1.132	**	**	**
5	**19.992**	10.209	4.764	2.693	1.143	0.471	**	**
6	20.170	10.218	5.050	2.989	1.333	0.491	0.279	0.292
4(6)	20.076	10.029	4.903	2.673	**1.121**	**0.406**	**0.205**	**0.119**
5(6)	20.217	10.073	4.818	**2.645**	1.135	0.422	0.217	0.139
6(6)	20.189	10.220	4.877	2.739	1.202	0.487	0.283	0.193
4(7)	20.253	10.023	4.772	2.704	1.161	0.439	0.232	0.141
5(7)	20.270	10.054	4.903	2.677	1.172	0.455	0.250	0.160
6(7)	20.161	10.125	4.907	2.762	1.237	0.522	0.312	0.218

(c) Sparse direct method (WSMP) as "lowest level" solver.

Table 4: Total solution time in seconds of the multigrid method with the first-order intertransfer operator for a problem size of 2^{24} with several different "lowest levels" on VIP.

L\ P	4	8	16	32	64	128	256	512
4	11.061	5.599	2.806	1.416	**0.705**	0.358	0.178	–
5	**11.035**	**5.581**	2.805	**1.415**	0.706	**0.353**	0.175	**0.073**
6	11.047	5.588	**2.803**	1.415	0.713	0.356	**0.174**	0.083
4(6)	11.074	5.586	2.808	1.418	0.709	0.353	0.182	0.084
5(6)	11.104	5.597	2.817	1.426	0.719	0.374	0.189	0.096
6(6)	11.099	5.651	2.875	1.483	0.773	0.425	0.247	0.155
4(7)	11.075	5.607	2.816	1.427	0.731	0.372	0.192	0.101
5(7)	11.074	5.616	2.825	1.435	0.732	0.376	0.199	0.107
6(7)	11.064	5.674	2.824	1.495	0.788	0.435	0.256	0.158

(a) Conjugate Gradient Method as "lowest level" solver.

L\ P	4	8	16	32	64	128	256	512
4	11.061	**5.586**	2.831	1.421	0.709	0.358	0.186	–
5	11.043	5.597	**2.807**	1.419	0.710	**0.356**	0.182	0.092
6	**11.038**	5.651	2.813	**1.416**	**0.708**	0.364	0.183	0.090
4(6)	11.066	5.593	2.808	1.419	0.709	0.357	0.178	**0.088**
5(6)	11.059	5.603	2.807	1.419	0.710	0.361	**0.177**	0.088
6(6)	11.045	5.599	2.811	1.421	0.715	0.364	0.182	0.094
4(7)	11.064	5.602	2.822	1.433	0.724	0.371	0.189	0.099
5(7)	11.063	5.598	2.824	1.431	0.724	0.373	0.191	0.099
6(7)	11.068	5.605	2.824	1.437	0.726	0.376	0.196	0.106

(b) Gauss-Seidel Relaxation method as "lowest level" solver.

L\ P	4	8	16	32	64	128	256	512
4	**11.044**	**5.598**	2.816	**	**	**	**	–
5	11.064	5.609	2.822	1.469	0.847	**	**	**
6	11.247	5.620	2.850	1.476	0.856	0.651	0.755	1.260
4(6)	11.139	5.605	**2.815**	**1.432**	**0.716**	**0.364**	**0.185**	**0.084**
5(6)	11.076	5.624	2.825	1.457	0.736	0.381	0.215	0.105
6(6)	11.092	5.669	2.876	1.487	0.781	0.430	0.251	0.164
4(7)	11.089	5.615	2.824	1.437	0.728	0.376	0.198	0.105
5(7)	11.086	5.633	2.837	1.447	0.740	0.392	0.209	0.109
6(7)	11.115	5.680	2.891	1.502	0.794	0.443	0.262	0.170

(c) Sparse direct method (WSMP) as "lowest level" solver.

Table 5: Total solution time in seconds of the multigrid method with the first-order intertransfer operator for a problem size of 2^{24} with several different "lowest levels" on HPC-FF.

P	2	4	8	16	32	64	128	256
1	19.992	9.911	4.699	2.615	1.072	0.390	0.196	0.106
1.01	20.192	10.010	4.746	2.641	1.083	0.394	0.198	0.107

(a) VIP.

P	4	8	16	32	64	128	256	512
1	11.035	5.581	2.803	1.415	0.705	0.353	0.174	0.073
1.01	11.145	5.637	2.831	1.429	0.712	0.356	0.176	0.074

(b) HPC-FF.

Table 6: The optimal solution times with an addition of 1% in seconds of the multigrid method with the first-order intergrid transfer operator for 2^{24} DoF on VIP and HPC-FF.

obvious that the differences measured between the different multigrid implementations are within a couple of ten percents only. However, it clearly shows that for larger number of cores the difference increases. This is a matter of how well the different multigrid implementations are parallelized (see Amdahl's law). Thus, the choice for the optimal solver among the multigrid implementations presented here is of interest when a good strong scaling property is required.

From Table 4 and 5, we can see that the data measured on HPC-FF are more consistent than the data measured on VIP. This seems due to the fact that on VIP there is a higher "noise level" due to operating system processes than on HPC-FF. Consequently, the uncertainty of the measured execution times on VIP is larger. In the remainder of this subsection we will focus on the results from HPC-FF.

The parallel multigrid version using the serial version of the single-core levels together with the "gathering level" six performs in all cases better than the serial version of the single-core levels together with the "gathering level" seven. Hence, the "gathering level" should not be larger than six, otherwise the execution is switched too early from parallel to serial which results in a performance penalty.

For WSMP it can be clearly seen why a single-core version of the "lowest level" solver has it benefits. Otherwise WSMP could not be used as "lowest level" solver on level four for more than 16 cores in parallel. In addition, Fig. 8 suggests even better results can be achieved with the LAPACK solver instead of the WSMP solver for the cases with "gathering level" six and

"lowest level" four.

Within the uncertainty of the measured execution times the parallel CGM "lowest level" solver with a "lowest level" of five gives the best results on HPC-FF up to the maximum chosen number of 512 cores. Thus, only in cases where the number of cores is larger than the number of DoF a single-core "lowest level" solver version should be taken into account.

2.7.2 Strong scaling of the optimal multigrid algorithm

From Table 5, we choose the best performing solver combination to investigate the strong scaling behavior of the multigrid method, i.e., multi-core version of multigrid with parallel CGM as the "lowest level" solver and level five as the "lowest level". We plot the solution time and speed up as a function of the number of cores in Fig. 14 for VIP and in Fig. 15 for HPC-FF. In the graphs, we plot the speed up of the solution time compared to two cores for VIP and to four cores for HPC-FF. As a reference, we plot the line of ideal speed up in both graphs. We also plot the speed up ratio of the solution time to the solution time on half the number of cores. For such a speed up ratio, the ideal is two.

In both cases, the results show a super-linear speed up due to cache effects which can happen when the memory consumption per core decreases. The super-linear speed up can be identified for more than 32 cores on VIP. We conclude that the multigrid method has linear scaling speed up to the maximum number of cores used in this study, i.e., 256 cores on VIP and to 512 cores on HPC-FF.

2.7.3 Weak scaling of the optimal multigrid algorithm

Next, we investigate the weak scaling properties of the multigrid method on HPC-FF. To investigate the weak scaling, we fix the work load, i.e., the DoF per core. Because doubling the mesh size in each direction increases the DoF by a factor of four, we increase the number of cores by a factor of four as well, to fix the DoF per core. We test four cases, i.e., 2^{18} DoF, 2^{19} DoF, 2^{20} DoF, and 2^{21} DoF per core with a weak scaling test, ranging from one core to 2048 cores on HPC-FF.

Due to the choice of our discretization of the unit square domain, (see Sec. 2.1), the number of cells (the DoF), is given by 2^{2l}. As a matter of grid refinement, we can run only cases having an even power, e.g., 2^{20} DoF and 2^{22} DoF on a single core. Cases having an odd power, e.g., 2^{19} DoF or 2^{21} DoF can only run in parallel on two cores because then the total number

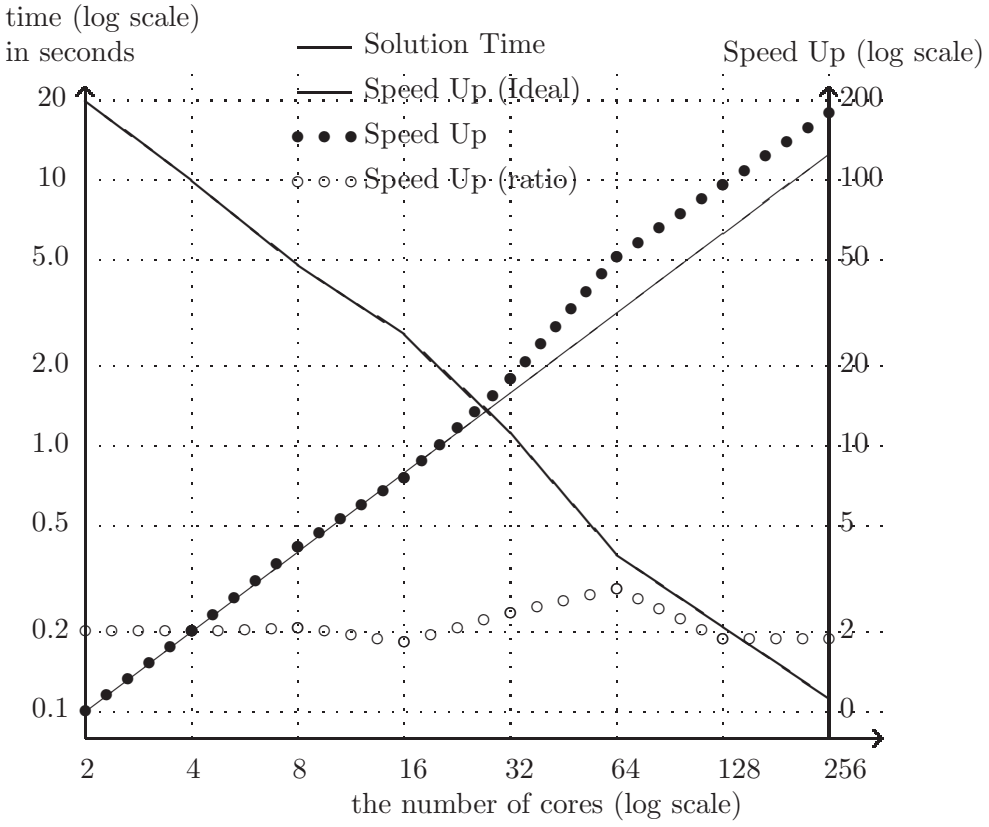


Figure 14: The solution times, speed up and speed up ratio of the multigrid method as a function of the number of cores on VIP uses a problem size of 2^{24} DoF (strong scaling).

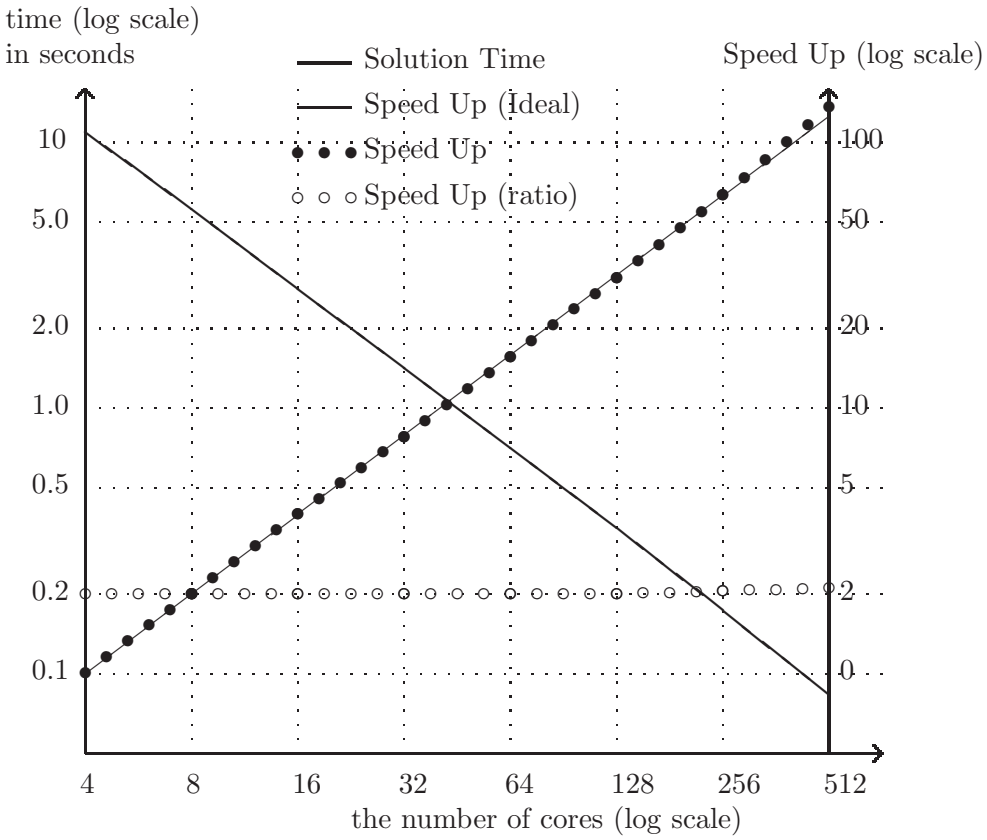


Figure 15: The solution times, speed up, and speed up ratio of the multigrid method as a function of the number of cores on HPC-FF using a problem size of 2^{24} DoF (strong scaling).

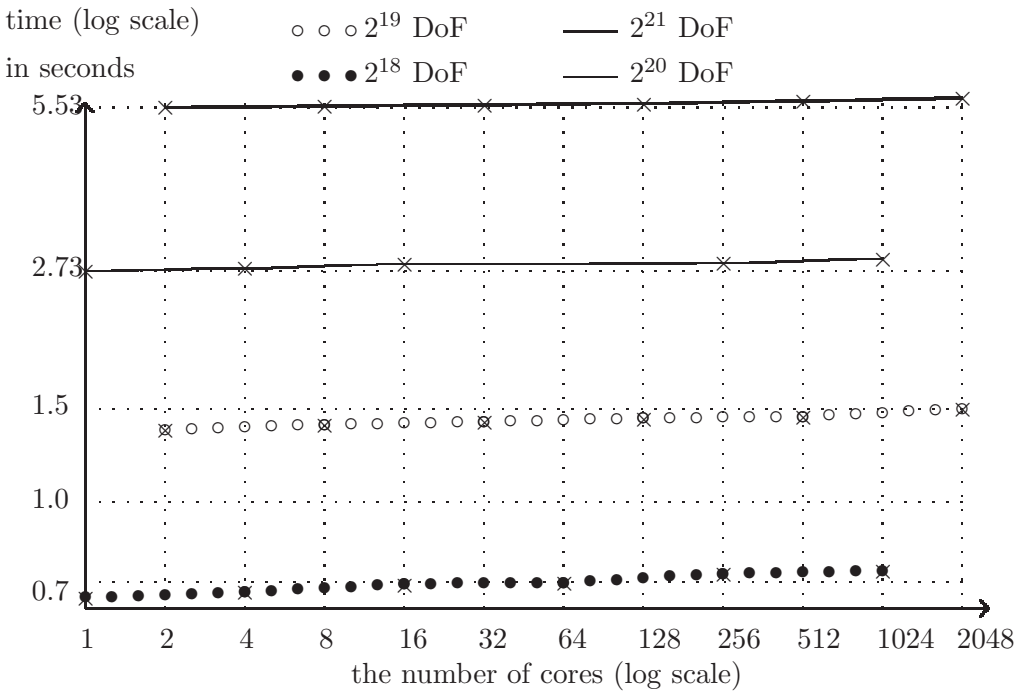


Figure 16: The solution times as a function of the number of cores on HPC-FF to solve the test problem with a fixed number of DoF per core (weak scaling).

of DoF adds up to 2^{20} and 2^{22}, respectively. We plot the solution times in Fig. 16 and the relative solution times compared to the solution time of the minimum number of cores in Fig. 17.

The graph in Fig. 16 shows that the multigrid method has very good weak scaling properties. We can investigate this further in Fig. 17. The larger the problem size per core the better is the weak scaling property. For the two largest test cases with 2^{20} DoF and 2^{21} DoF per core, the execution time increase is less than 5%. This is remarkable as the scaling of the number of cores spans a factor of 1024.

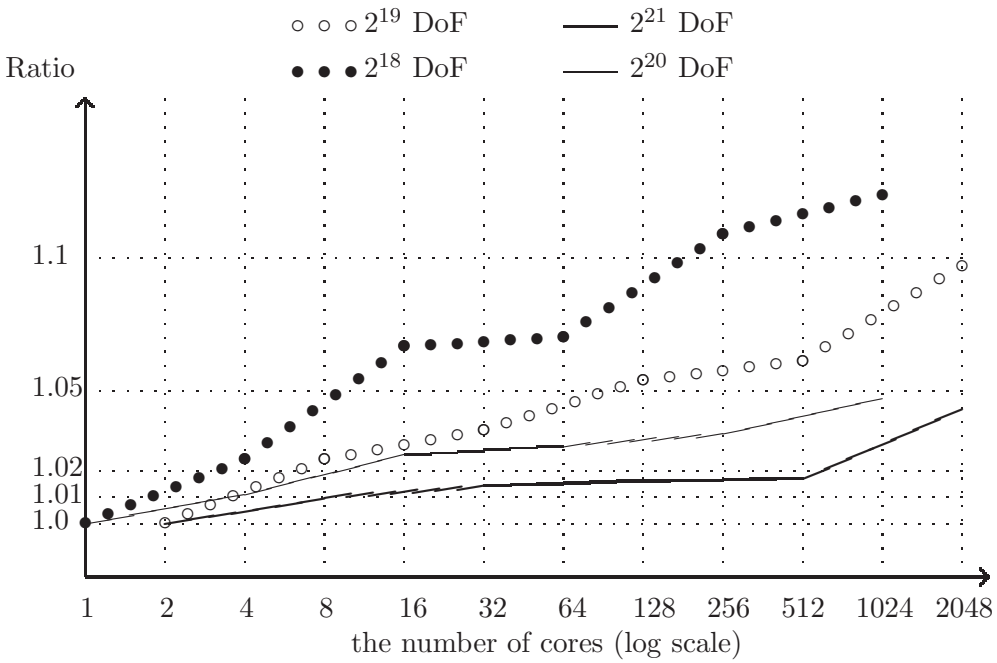


Figure 17: The relative solution times as a function of the number of cores on HPC-FF to solve the test problem with a fixed number of DoF per core (weak scaling).

3 Conclusions

The multigrid algorithm is a well-known and efficient algorithm to solve many classes of problems but it is complex to implement especially when it comes to parallel execution. It usually can not be implemented by just calling an appropriate library routine. Instead, the geometrical multigrid algorithm has to be adapted by hand to the problem of interest. Special care has to be taken about the (parallel) computer hardware which should be finally used for program execution.

In these notes we have assessed the multigrid method to solve the Poisson problem on a rectangular domain with uniform meshes using a cell-centered finite differences (CCFD) scheme. In the focus was the performance of the implemented multigrid algorithm on massively parallel machines with many thousands of processors.

Initially, it was not clear if the zeroth- or first-order intergrid transfer operators would provide better performance for the CCFD method. Corresponding tests revealed that in the case of the first-order intergrid transfer operator the error reduction factor and the number of iterations did not change any further with the total number of multigrid levels after a "highest level" of nine was reached. Instead the zeroth-order intergrid transfer operator imposes a strong correlation of these quantities to the total number of multigrid levels. Thus, we conclude that the first-order intergrid transfer operator should be preferred for larger problems.

We investigated four popular solvers as coarsest problem solvers on the "lowest level": the Red-Black Gauss-Seidel Relaxation method, a sparse direct solver (IBM WSMP), a dense direct solver (LAPACK), and the Conjugate Gradient Method (CGM). The solution time of each solver was measured for several different problem sizes from 2^8 to 2^{22} degree of freedoms (DoF). For our smallest problem size (2^8 DoF), CGM, Relaxation and LAPACK(solving) are faster than the other solvers and CGM is the fastest solver for relatively small problem sizes (from 2^{10} to 2^{16} DoF). For larger problem sizes ($> 2^{16}$ DoF), multigrid and WSMP back substitution yield similar good results.

In addition, we compared the parallel performance of the solvers. The results show that CGM is again the best parallel solver for problem sizes with 2^{10} to 2^{16} DoF. As expected, the optimal number of cores increases with the size of the problem.

Next, we focused on the scaling properties of the parallel WSMP and multigrid algorithm to solve a larger problem with 2^{20} DoF. The back substitution of WSMP is better than the multigrid method on small number

of cores, i.e., less than four cores for VIP and less than sixteen cores for HPC-FF. For larger numbers of cores, the multigrid method is faster than WSMP due to better scaling properties. Thus, it is the best scaling method for larger problems.

This result suggests that we might get the fastest solution time with one of the following strategies. Either we combine the parallel multigrid method with the parallel CGM solver as "lowest level" solver, i.e., using the CGM solver on the coarsest grid to get the exact solution. Or we gather the data from all the cores on each core at a certain level (called "gathering level") and proceed with the single-core version of the multigrid method until we reach the coarsest grid level. As "lowest level" solver we would have again the choice between the CGM, Relaxation, and LAPACK/WSMP solver. The second strategy has the benefit that it can enlarge the applicability of the parallel multigrid algorithm for a fixed problem size to very large numbers of cores (processors). In such cases, the degree of freedom at certain levels might be less than the number of cores when approaching to the "lowest level". Hence, one has the choice between switching to the serial version of the multigrid method for calculating the single-core levels or to set the "lowest level" high enough in the V-cycle to keep its grid large enough for parallelization. The first strategy has the disadvantage that the parallel execution is abandoned in favor of a single-core execution and the second strategy has the disadvantage that the multigrid algorithm which acts very efficiently on the "high levels" might be exchanged with a less efficient algorithm used as a "lowest level" solver. It is nearly impossible to tell in advance which strategy is optimal for a given hardware. Hence, practical test had to be carried out.

Detailed tests show that within the uncertainty of the measured execution times the fully parallel multigrid implementation with parallel CGM "lowest level" solver with a "lowest level" of five, gives the best results for a problem size of 2^{24} on HPC-FF up to the maximum number of 512 cores. Thus, only in cases where the number of cores is larger than the number of DoF, the single-core multigrid version should be taken into account. A strong scaling of this optimal solver shows a perfect linear speed up to 512 cores on HPC-FF.

Finally, we investigated the weak scaling properties of the optimal solver for four different test cases, i.e., 2^{18} DoF, 2^{19} DoF, 2^{20} DoF, and 2^{21} DoF per core with weak scaling from one core to 2048 cores on HPC-FF. The multigrid method has very good weak scaling properties. The larger the problem size per core the better it scales. For the two largest test cases with 2^{20} DoF and 2^{21} DoF per core, the execution time increases by less than

5%. This is remarkable as the scaling spans an increase of the core number by a factor of 1024.

Over all, we showed that our implementation of the multigrid method with the CGM as a "lowest level" solver and with first-order intergrid transfer operators has very good strong and weak scaling properties. Thus, it is suitable for usage on massively parallel machines like HPC-FF.

Acknowledgments

I would like to thank R. Hatzky and B. Scott for their helpful discussion.

References

[1] J. Bramble, "Multigrid Methods", Pitman, London, 1993.

[2] A. Brandt, "Multigrid Techniques: 1984 Guide, with Applications to Fluid Dynamics", *GMD studien 85*, GMD, Forschungszentrum Informationstechnik, St. Augustin, Germany, 1984.

[3] A. Gupta, "WSMP: Watson Sparse Matrix Package. Part I – direct solution of symmetric sparse systems", IBM Research Report RC21886(98462), 2000.

[4] W. Hackbush, "Multigrid Methods and Applications", Springer-Verlag, Berlin, 1985.

[5] K. S. Kang, "P_1 Nonconforming Finite Element Multigrid Method for Radiation Transport", *SIAM J. Sci. Comp.*, **25** 2003, pp. 369–384.

[6] D. E. Keyes, "Terascale Implicit Methods for Partial Differential Equations", The Barrett Lectures, University of Tennessee mathematics Department, 2001, Contemporary Mathematics 306: 29–84, AMS, Providence.

[7] D. A. Knoll, G. Lapenta, and J. U. Brackbill, "A multilevel iterative field solver for implicit, kinetic, plasma simulation", *Journal of Computational Physics,* **149** 1999, pp. 337.

[8] D. A. Knoll and W. J. Rider, "A multigrid preconditioned Newton-Krylov method", *SIAM J. Sci. Comput.,* **21** 1999, pp. 691.

[9] Do Y. Kwak, "V-cycle multigrid for cell-centered finite differences", *SIAM J. Sci. Comput.,* **21** 1999, pp. 552–564.

[10] Do Y. Kwak and J. S. Lee, "Multigrid Algorithm for the Cell-Centered Finite Difference Method II: Discontinuous Coefficient Case", *Numerical Methods for Partial Differential Equations*, **20** 2004, pp. 742–764.

[11] P. Wesseling, "An Introduction to Multigrid Methods", John Wiely, Chichester, UK, 1992.