Highly Parallel Multigrid Solvers for Multicore and Manycore Processors

Oleg Bessonov(✉)

Institute for Problems in Mechanics of the Russian Academy of Sciences, 101, Vernadsky Avenue, 119526 Moscow, Russia
bess@ipmnet.ru

Abstract. In this paper we present an analysis of parallelization properties and implementation details of the new Algebraic multigrid solvers. Variants of smoothers and multicolor grid partitionings are discussed. Optimizations for modern throughput-oriented processors are considered together with different storage schemes. Finally, comparative performance results for multicore and manycore processors are presented.

1 Introduction

This paper is devoted to the development of efficient parallel algebraic methods for solving large sparse linear systems arising in discretizations of partial differential equations. Historically, Conjugate Gradient methods have been widely used for solving such linear systems [1]. In order to accelerate the solution, implicit preconditioners of the Incomplete LU-decomposition type (ILU) are applied [2,3]. However, implicit preconditioners have limited parallelization potential and therefore can't be efficiently employed on massively parallel computers.

On the other hand, there exists a separate class of implicit methods, multigrid, which possess very good convergence and parallelization properties. Multigrid solves differential equations using a hierarchy of discretizations. At each level, it uses a simple smoothing procedure to reduce corresponding error components.

In a single multigrid cycle, both short-range and long-range components are smoothed. It means that information is propagated instantly throughout the domain within a such cycle. As a result, this method becomes very efficient for elliptic problems that propagate physical information infinitely fast.

At the same time, multigrid can be efficiently and massively parallelized because processing at each grid level is performed in the explicit manner, and data exchanges are needed only between adjacent subdomains at the end of a cycle.

In this paper we will consider the Algebraic multigrid (AMG) approach [4] which is based on matrix coefficients rather than on geometric parameters of a domain. Parallelization properties and implementation details of this algorithm will be analyzed, and performance results for modern throughput processors will be presented.

V. Malyshkin (Ed.): PaCT 2015, LNCS 9251, pp. 10–20, 2015.
DOI: 10.1007/978-3-319-21909-7_2

2 Iterative Methods and Their Parallelization Properties

The multigrid approach will be presented here in the context of the hierarchy of iterative methods. Iterative methods are used for solving large linear systems arising in discretizations of partial differential equations in many areas (fluid dynamics, semiconductor devices, quantum problems). They can be applied to ill-conditioned linear systems, both symmetric and non-symmetric.

The most popular class of iterative methods is the Conjugate Gradient (CG). In order to accelerate the convergence, this method requires preconditioning. There are two main classes of preconditioners: explicit, that act locally by means of a stencil of limited size and propagate information through the domain with low speed, and implicit, that operate globally and propagate information almost instantly. Due to this implicit preconditioners work much faster and have better than linear dependence of convergence on the geometric size of the problem.

The same applies to stand-alone iterative methods that possess similar properties and may be classified as being explicit or implicit (we consider here the nature of internal iterations rather than the outer properties of a method).

Parallel properties of iterative solvers strongly depend on how information is propagated in the algorithm. For this reason methods with the implicit nature of iterations can't be easily parallelized, and many efforts are needed for finding geometric or algebraic approached of parallelization [3].

However, some methods with explicit iterations are also essentially sequential, and their parallelization may become difficult. In particular, the Gauss-Seidel and Successive Over-Relaxation (SOR) methods in their original form need the sequential processing of grid points in a domain. In order to overcome this problem, multicolored approaches are used when all grid nodes in a domain are partitioned in such a way that each node has no connection with another nodes of the same color. Owing to this, computations for all grid points with the same color can be performed in any order thus allowing arbitrary splitting of the domain. For the 7-point stencil in 3D, it is enough to use 2 colors (Red-black partitioning), while the more general 27-point stencil requires 8 colors.

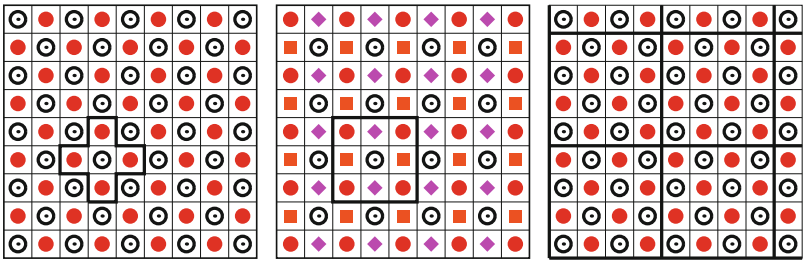

Fig. 1. Red-black (left) and multicolor (center) grid partitionings; splitting of a computational domain with the red-black partitioning for the parallelization (right) (Color figure online)

Figure 1 illustrates grid partitionings for 2 and 4 colors and splitting of a computational domain into subdomains for processing them in parallel.

Properties of different iterative methods are presented in Table 1. Solutions of the Poisson equation on two different uniform grids were used for this test, with the convergence stopping criteria 10^{-10}. The approximate cost of an iteration is presented for each method (relative to the Jacobi method). For the Gauss-Seidel and SOR, Red-black variants are given. Fortunately they have the same convergence properties as their sequential counterparts. For the Conjugate Gradient, several preconditioners were tested: plain diagonal one (CG), polynomial Jacobi explicit preconditioner (CG Jacobi) and Incomplete LU in two variants (CG ILU and MILU) [2,3]. For the multigrid, two implementations were used: a plain AMG solver and an AMG-preconditioned Conjugate Gradient.

Table 1. Convergence of different iterative algorithms for square matrices of size N

Iterative method	$N = 65$	$N = 129$	$f(N)$	Cost of iteration
Jacobi	23650	90850	$O(N^2)$	1
Gauss-Seidel	12210	46940	$O(N^2)$	1
SOR	299	600	$O(N)$	1
CG	258	514	$O(N)$	1.5
CG Jacobi	130	257	$O(N)$	2.25
CG ILU	109	212	$O(N)$	3
CG MILU	48	68	$O(N^{1/2})$	3
AMG	8	8	$O(1)$	5
CG AMG	7	7	$O(1)$	6

The above results confirm that the convergence of the multigrid doesn't depend on the problem size. The next efficient method is the CG MILU with the required number of iterations of the order of $O(N^{1/2})$. This method is still applicable and efficient for many problems that are not convenient for the multigrid (anisotropic grids, systems of equations etc.). However, the ILU methods have very limited parallelization potential [2,3] and, for this reason, simple explicit methods and preconditioners of the $O(N)$ class remain attractive in some cases.

Let's consider now the Red-black Gauss-Seidel and SOR methods. These methods become fully explicit and can be represented as two half-sweeps (Fig. 2). Here parts 1 and 2 of a matrix represent grid nodes of the first and the second color respectively. The principal operation in each half-sweep is the multiplication of a sparse matrix by a corresponding vector. Such kind of operations can be massively and efficiently parallelized. Therefore they can be used as smoothers in highly parallel implementations of the multigrid.

Efficient implementation of the matrix-vector multiplication for multicolor grid partitionings requires reordering and reorganization of the original matrix.

$$(D+L)x_{k+1} = b - L^{\mathrm{T}}x_k$$

Two half-sweeps:

$$D^{(1)}x_{k+1}^{(1)} = b^{(1)} - L^{\mathrm{T}}x_k^{(2)}$$

$$D^{(2)}x_{k+1}^{(2)} = b^{(2)} - Lx_{k+1}^{(1)}$$

Explicit representation:

$$x_{k+1} = D^{-1}(I - LD^{-1})(b - L^{\mathrm{T}}x_k)$$

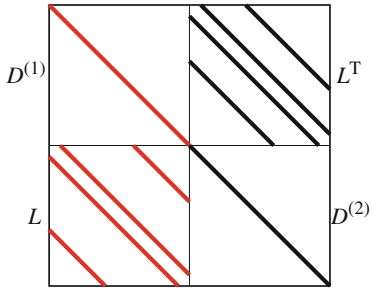

Fig. 2. Iteration of the Red-black Gauss-Seidel method for a symmetric matrix (Color figure online)

3 Throughput-Oriented Processors and Storage Schemes

All modern high-performance microprocessors belong to the class of throughput-oriented processors. This means that their performance is achieved in cooperative work of many processor cores and depends not only on the computational speed of cores but also on the throughput of the memory subsystem. The latter is determined by characteristics of the cache memory hierarchy, number and width of integrated memory controllers, memory access speed and capacity of inter-core or interprocessor communications. Additionally some memory optimization features are presented in the processor such as streamlined prefetch of regularly accessed data, fast access to unaligned data and facilities for efficient indirect accesses (gather, scatter) which are needed for processing sparse matrices.

Another main feature of throughput-oriented processors is vectorization: several elements of data can be packed in a vector and processed simultaneously by a single machine instruction. This feature is supported by the smart vectorizing compilers which can be controlled by auxiliary directives in a source code.

Finally, shared memory organization with coherent caches is needed for parallelization, together with the appropriate software support (OpenMP compilers and other parallel environments).

There are two principal classes of throughput-oriented processors – multicore and manycore. Multicore means just a presence of several traditional processor cores in a single semiconductor chip, with the typical number of cores up to 8–12, clock frequency around 3 GHz and standard integrated memory interfaces (up to 4 64-bit channels of DDR3 or DDR4). Typical vector width is 128 or 256 bits.

Manycore (MIC) is a new class of processors with large number of simple and relatively slow cores. Each core is equipped with a very powerful floating point unit (FPU) that processes 512-bit vectors organized as 8×64-bit or 16×32-bit words. The only current implementation of MIC is Intel Xeon Phi, that have the following characteristics (for the model 5110P): 60 cores, up to 4 threads per core (240 threads total), frequency 1.05 GHz, 8 channels of GDDR5 memory. Thus manycore processors are also throughput-oriented, they possess all necessary properties: parallelization, vectorization and memory optimization.

However, manycore processors differ from their multicore counterparts in the balance between components of performance: they need much more threads to saturate a processor (240 vs. 8–12), rely on slower scalar performance of a single thread (speed ratio about 1:10), benefit from regular vectorized floating point operations (512-bit vectors vs. 256-bit) and from very powerful memory subsystem (150–20 GB/s vs. 40–50 GB/s). As a results they are able to demonstrate the level of performance about 1.5 times higher than bi-processor systems built on multicore processors when running realistic memory-bound applications.

Manycore is often considered just as a superfast arithmetic engine (like GPU). However, it is conceptually a universal parallel processor and, compared to GPU, its performance potential is much wider, as well as the range of applications.

Both multicore and manycore processors need special optimizations of application programs in order to achieve high performance, such as contiguous data placement, avoiding very sparse structures etc. It is a good programming practice if the same source code is developed for both types of processors.

The most important points of these optimizations is a storage format for sparse matrices. Usually, the Compressed Row Storage (CRS) is used, when non-zero elements of a matrix are stored contiguously row-by-row, being accompanied with their column indexes. In this case, addressing elements of a vector by which this matrix is being multiplied is performed indirectly. However, a set of vector elements being processed for a given row can be located very sparsely: for example, in the discretization of a regular 3D domain of the size n^3, the distance between elements will be $O(n^2)$. Such non-regular and non-contiguous data accesses are very non-optimal for modern processors, especially if they are performed using vectorized forms of data load operations (gather).

A good alternative is the Compressed Diagonal Storage (CDS). This format is applicable for more-or-less regular discretizations with a stencil of the limited size – in particular, for Cartesian discretizations in arbitrary domains (e.g. in the Level set method). In this format, a matrix is considered as consisting of the limited number of diagonals (more exactly, quasi-diagonals), and matrix elements are stored contiguously within each diagonal, being accompanied with their column indexes. Each diagonal must contain an element for each row even if it is zero. On the other hand, corresponding vector elements being addressed by the indexes are located almost densely. As a result, vectorized forms of data load operations can be employed very efficiently.

Both storage schemes were evaluated using the Jacobi-preconditioned Conjugate Gradient solvers for two matrix sizes. For the smaller matrix (0.32 M grid points), the performance gain of the CDS scheme was about 1.25 and 1.4 on multicore processors (for the FPU vector size 128 bits and 256 bits, respectively) and about 1.8 on a manycore (vector size 512 bits). For the large matrix (4.8 M grid points), the full memory throughput rate of 80 GB/s was achieved on the tested bi-processor multicore system (Sandy Bridge). On the manycore processor (model 3120P), the achieved level was 120 GB/s that is also close to the limit. Thus, these tests demonstrated the importance of adequate data structures and access patterns for modern high-performance processors.

4 Description of the Algebraic Multigrid

Algebraic multigrid (AMG) is based on matrix coefficients rather than on the geometry of a computational domain. Owing to the strict mathematical foundation, this method works fine and demonstrates excellent convergence on solving ill-conditioned elliptic linear systems. For regular grids, however, the Algebraic multigrid has the natural geometric interpretation.

Here we consider implementations of the AMG for Cartesian discretizations in arbitrary domains. These discretizations produce unified 7-point stencils and, as a result, it becomes possible to use more simple and regular data structures.

An iteration of the multigrid is usually represented as a V-cycle (Fig. 3). Within this cycle, a smoothing procedure is performed on a hierarchy of discretizetions thus reducing high-frequency and low-frequency components of the residual vector.

1	Pre-smooth	$x_1 = S_1(x_0, b)$
2	Residual	$b_1 = b - Ax_1$
3	Restriction	$\tilde{b}_1 = R\,b_1$
4	Next level	$\tilde{A}\tilde{x}_2 \approx \tilde{b}_1$
5	Prolongation	$x_2 = P\tilde{x}_2$
6	Correction	$x_3 = x_1 + x_2$
7	Post-smooth	$x_0 = S_2(x_3, b)$

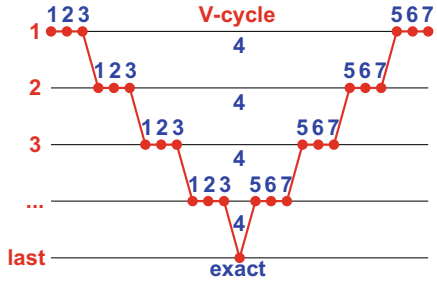

Fig. 3. Multigrid algorithm (left) and illustration of V-cycle (right)

Smoothing is the most important part of the algorithm. It is first performed in the beginning of each level reducing corresponding error components. Then the residual vector is restricted (averaged) into the more coarse grid, and the algorithm is recursively executed at the next level. After that, the new coarse residual vector is prolongated (interpolated) into the current fine grid, the result is corrected, and the smoothing procedure is performed again. At the last level, the coarsest grid equation is solved either exactly or by a simple iterative procedure (not necessarily accurate).

At the initialization phase of the algorithm, a hierarchical sequence of grids should be built, together with intergrid transfer operators R and P. Also, matrices for all grid levels should be constructed.

A procedure of building the next (coarse) grid from the current (fine) grid is called coarsening. At the particular level, a subset of variables (grid nodes) is selected for the next level. They are called Coarse nodes (C-nodes), while the remaining ones are Fine nodes (F-nodes). Different coarsening algorithms exist, more or less aggressive. Here, the natural geometric coarsening is applied, when the number of grid points is reduced by 8 at each level in 3D (Fig. 4).

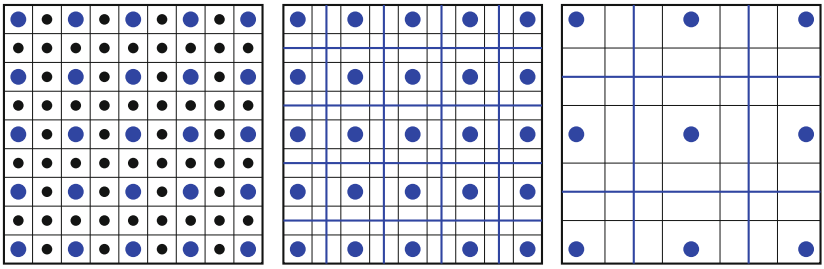

Fig. 4. Fine grid and two levels of coarsening (left to right)

F-nodes can be connected to C-nodes either directly (strong connection) or indirectly (weak connection). These connections are used for building intergrid transfer operators, firstly the prolongation (interpolation) operator P (Fig. 3).

In the AMG, operator-dependent interpolations are used, which are based on matrix coefficients. For strongly connected F-nodes, interpolations are performed in accordance with coefficient weights in the connections to adjacent C-nodes (direct interpolation). For weakly connected nodes, intermediate weights in the connections to adjacent strong-F-nodes are used (standard interpolation). For more weakly connected F-nodes, an additional interpolation step is needed [4].

The restriction (averaging) operator is constructed as a transpose of the prolongation operator: $R = P^{\mathrm{T}}$.

The next step is to build a matrix for the coarser level from the current one by means of the Galerkin operator (Fig. 5). It is related to the restriction (R) and prolongation (P) operators. The Galerkin operator retains the symmetry and some other properties of a matrix.

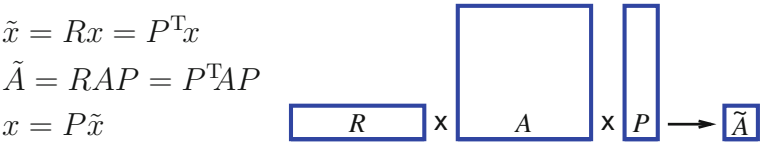

Fig. 5. Building a coarser level matrix by the Galerkin operator

For the above interpolations (in 3D), all constructed matrices have 27 quasi-diagonals, that corresponds to a 27-point discretization stencil at each level except the first one (which has a 7-point stencil for orthogonal grids). Despite this, the first level is the most computationally expensive because of the larger number of grid points. The second level with 8 times less nodes is still expensive because of 4 times larger stencil. Therefore, algorithms for the first two levels should be well optimized while the remaining levels are less important for performance.

Within an AMG iteration, pre- and post-smoothers are the most important parts of the algorithm both for efficiency and convergence. In the current implementation, Red-black Gauss-Seidel or SOR smoothers are used at the first level and 8-color Gauss-Seidel at others. Several iterations of the smoothing algorithm can be applied. In particular, the typical number of iterations is $1\frac{1}{2}$ (3 half-sweeps) or 2 for the first two levels, and up to 3 or 4 for others.

At the first level, the Compressed Diagonal storage scheme is used for the main matrix. The matrix is stored in two parts to separate red and black nodes. This is necessary because at each step of the algorithm, only elements of a particular color are processed, and storing them in the natural order (when red and black elements alternate with each other) would lead to the twofold increase of data being read from the memory.

For the next levels, matrix coefficients are stored densely with 27 elements in a row, being accompanied with their column indexes in another dense array.

For storing the restriction operator R, a similar scheme is used at all levels because this operator also corresponds to a 27-point stencil. On the other hand, stencils of the prolongation operator P are not uniform and may have different number of points, depending on the class of a grid node (C, strong-F, weak-F, weak-weak-F). For this reason, several sparse structures are organized to assist the interpolation.

Additionally, some auxiliary data array are build for storing different characteristics of grid nodes (color, interpolation class, intergrid references etc.).

The above optimizations of sparse data structures allowed to eliminate difficulties in parallelization and vectorization of the algorithm and increase its computational speed.

For the last grid level, a simple iterative solver of the Conjugate Gradient class can be applied. This solver doesn't need to be accurate, it is usually enough to reduce the residual norm by only two orders of magnitude. The solver works fine on multicore computer systems with a moderate number of threads. However, on manycore processors there is too few computational work in each thread (for the typical last-level matrix size about 500). At the same time, several barrier synchronizations for all threads are needed at each iteration of the CG algorithm that leads to large delays and increases the computational time.

In order to avoid this problem, a variant of the solver based on the matrix inversion was developed. In this approach, the original last-level sparse matrix is explicitly inverted by the Gauss-Jordan algorithm at the initialization phase. The resulting full (dense) matrix is used at the execution phase in the very simple algorithm of matrix-by-vector multiplication. This algorithm is perfectly parallelized and doesn't need synchronizations. The only requirement is that the original last-level matrix should not be large, i.e. the sufficient number of multigrid levels should be defined. This algorithm works well on both manycore and multicore processors. Another point is that it is direct and therefore more robust in comparison with iterative CG solvers.

The described multigrid solver for non-symmetric linear systems was implemented in Fortran with OpenMP parallelization. The same source code works

both on multicore and manycore processors, demonstrating good parallelization efficiency. The solver has many parameters for fine tuning the AMG algorithm.

5 Multigrid as a Preconditioner

Another approach is to use the multigrid idea for preconditioning in a solver of the Conjugate Gradient class. In this approach, the meaning of a multigrid iteration (V-cycle) is different. While in the plain multigrid V-cycles are repeated until convergence, gradually reducing the residual norm, in the multigrid-preconditioned CG a V-cycle is applied once in each CG iteration in order to reduce the condition number of a matrix.

Surprisingly, the multigrid-preconditioned Conjugate Gradient method often behaves substantially better than the plain multigrid: it is more robust and converges faster.

Additionally, it becomes possible to use the single-precision arithmetic for the multigrid part of the algorithm instead of the traditional double-precision without loosing the overall accuracy. Due to this, the computational cost of the algorithm can be decreased because of reduced sizes of arrays with floating point data and corresponding reduction of the memory traffic.

As a result, the AMG-preconditioned CG solver becomes faster than the plain AMG in a single iteration and at the same time needs less iterations to converge. Currently the AMG CG solver is implemented for symmetric matrices only. Later it will be extended for the nonsymmetric case as an AMG-preconditioned BiCGStab. Typical iteration counts are shown below for the CG AMG solver vs. the plain AMG for several test matrices:

- small spherical domain (0.32 M grid points): 7 vs. 10,
- large spherical domain (2.3 M grid points): 8 vs. 11,
- long cylindrical domain, aspect ratio 5:1 (2 M grid points): 9 vs. 14.

For the last problem, the CG AMG solver is about 1.7 times faster than the plain AMG.

In order to achieve better convergence, some tuning of the algorithm is necessary, such as defining the number of grid levels, the number of iterations (half-sweeps) for smoothing at each level and the value of the over-relaxation factor at the first level. In particular, the SOR factor around 1.15 is usually optimal for convergence.

6 Performance of the Multigrid Solvers

Performance results of the new AMG-preconditioned Conjugate Gradient solver in comparison with the Jacobi-preconditioned (explicit) Conjugate Gradient are shown in Table 2. The following computer systems were used for these tests:

- Bi-processor Xeon E5-2690v2, 3 GHz, 2×10 cores, 20 threads;
- Xeon Phi model 5110P, 1.05 GHz, 59 cores, 236 threads.

Table 2. Performance results for multicore and manycore computer systems

Computer system	CG Jacobi	CG AMG	AMG : CG
multicore bi-Xeon	4.87 s	0.172 s	28 : 1
manycore Xeon Phi	3.15 s	0.185 s	17 : 1
manycore : multicore	1.55 : 1	0.93 : 1	—

The test matrix represents a discretization in a long cylindrical domain with the aspect ratio 5:1 (2 M grid points). Iteration counts are 9 for the CG AMG and 690 for the CG Jacobi (this corresponds to 1380 iterations for the simple diagonally-preconditioned CG [3]).

The above results demonstrate the great superiority of the AMG-preconditioned method over the plain Conjugate Gradient. The important reason is that this particular matrix represents a long domain with the number of grid points about 400 in the longest direction, that determines the iteration count for explicit CG algorithms. This example also confirms that the multigrid iteration count doesn't depend on the problem size.

On shorter domains, superiority of the multigrid is a little bit less. In particular, for a large spherical domain (2.3 M grid points), iteration counts are 8 and 350 respectively, and AMG : CG speed ratio is about 16:1 for the multicore Xeon processor and about 10:1 for the manycore Xeon Phi. These proportions are expected to be typical for most matrices of the similar size.

The comparison of the manycore Xeon Phi to the multicore bi-Xeon system demonstrates its moderate superiority in the simple explicit CG which is a pure memory-bound algorithm. The speed proportion 1.55:1 in favor of Xeon Phi roughly corresponds to the ratio between achievable memory throughput rates for these systems. The multigrid algorithm, in turn, is much more complicated and less regular, especially in memory access patterns and at higher (coarser) grid levels. For this reason, the relative performance of Xeon Phi becomes less. Nevertheless this result should be considered as very reasonable taking into account the highly parallel nature of this processor.

7 Conclusion

In this work we have presented new efficient parallel solvers for Cartesian discretizations in general domains. Two variants of the solvers are built, based on the Algebraic multigrid approach (AMG) and on the Conjugate Gradient method with the AMG preconditioning (CG AMG). Both solvers use the advanced Compressed Diagonal storage format suitable for efficient processing on modern throughput-oriented computer systems. The solvers are targeted on the solution of Poisson-like and other ill-conditioned linear systems, both symmetric and non-symmetric.

The new solvers have been evaluated and tested on multicore (bi-Xeon) and manycore (Xeon Phi) processors using several matrices of different size.

In comparison to the Conjugate Gradient class solvers with explicit preconditioning, they have demonstrated the speed increase up to 10–16 times typically and up to 17–28 for elongated domains.

The obtained results have also demonstrated that manycore processors can be efficiently employed for solving algebraic problems of the general class using standard programming languages and parallel environments (Fortran, OpenMP).

Acknowledgements. This work was supported by the Russian Foundation for Basic Research (project RFBR-15-01-06363) and by the Institute of mathematics (IMATH) of the University of Toulon. Computations have been performed at the BULL's Computing Center, IMATH and Mésocentre of the University of Aix-Marseille, France.

References

1. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. Carnegie Mellon University, School of Computer Science, Pittsburgh (1994)
2. Accary, G., Bessonov, O., Fougère, D., Gavrilov, K., Meradji, S., Morvan, D.: Efficient parallelization of the preconditioned conjugate gradient method. In: Malyshkin, V. (ed.) PaCT 2009. LNCS, vol. 5698, pp. 60–72. Springer, Heidelberg (2009)
3. Bessonov, O.: Parallelization properties of preconditioners for the conjugate gradient methods. In: Malyshkin, V. (ed.) PaCT 2013. LNCS, vol. 7979, pp. 26–36. Springer, Heidelberg (2013)
4. Stüben, K.: A review of algebraic multigrid. J. Comput. Appl. Math. **128**, 281–309 (2001)