Parallelization Properties of Preconditioners for the Conjugate Gradient Methods

Oleg Bessonov

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

Abstract. In this paper we present the analysis of parallelization properties of several typical preconditioners for the Conjugate Gradient methods. For implicit preconditioners, geometric and algebraic parallelization approaches are discussed. Additionally, different optimization techniques are suggested. Some implementation details are given for each method. Finally, parallel performance results are presented and discussed.

1 Introduction

Conjugate Gradient methods are widely 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 (plain CG) and non-symmetric (BiCGStab, GMRES etc.). In order to accelerate convergence, these methods require preconditioning. Now, with the proliferation of multicore and manycore processors, efficient parallelization of preconditioners becomes very important.

There are two main classes of preconditioners: explicit, that apply only a matrix-vector multiplication, and implicit, that require solution of auxiliary linear systems based on the incomplete decomposition of the original matrix. Explicit preconditioners act locally by means of a stencil of limited size and propagate information through the domain with low speed, while implicit preconditioners operate globally and propagate information instantly. Due to this implicit preconditioners work much faster and have better than linear dependence of convergence on the geometric size of the problem.

Parallel properties of preconditioners strongly depend on how information is propagated in the algorithm. For this reason implicit preconditioners can't be easily parallelized, and many efforts are needed for finding geometric and algebraic approached of parallelization. There exists a separate class of implicit methods, Multigrid, which possesses very good convergence and parallelization properties. However, Multigrid is extremely difficult for implementation, and in some cases it can't be applied at all. Due to this classical (explicit and implicit) preconditioners are still widely used in many numerical applications.

Thereby, in this paper we will analyze parallelization properties and performance of several preconditioners for different discretizations and geometries, and their implementation details on modern multicore processors.

2 Conjugate Gradient and Preconditioners

The original non-preconditioned Conjugate Gradient method [1] of the solution of a linear system $A\boldsymbol{x} = \boldsymbol{b}$ is very simple for implementation and can be easily parallelized. However, because of the explicit nature, it has low convergence rate. Because of this, the CG method is usually applied to the preconditioned linear system $(M^{-1}A)\boldsymbol{x} = M^{-1}\boldsymbol{b}$ where M is a symmetric positive-definite matrix that is "close" to the main matrix A (which is also symmetric and positive-definite).

Preconditioning works well if the condition number of the matrix $M^{-1}A$ is much less than that of the original matrix A. The simplest way to reduce this condition number and accelerate the convergence is to apply an "explicit" preconditioner ($B = M^{-1}$) than doesn't require the inversion of M.

A good example of this sort is the polynomial Jacobi preconditioner that is based on a truncated series of the approximation $1/(1-a) = 1 + a + a^2 + \ldots$

$$B = M^{-1} = \sum_{k=0}^{n}(H^k)P^{-1} \text{ where } P = \mathrm{diag}(A),\ H = P^{-1}(P-A) = I - P^{-1}A$$

For $n = 0$, this expression represents the diagonal preconditioner $B = P^{-1}$ (not considered as a true preconditioner because of its simplicity). For $n = 1$, it looks as $B = (I + (I - P^{-1}A))P^{-1}$ and improves acceleration rate by two times (with some increase of computational cost). This corresponds to the expansion of the computational stencil of one iteration of the algorithm. Therefore, it can be easily applied and parallelized. Variants for $n = 2$ or $n = 3$ are more complex and not enough efficient, for this reason they are not considered here.

Neither sort of the simple explicit preconditioner can improve the convergence radically. For this reason, it is desirable to use implicit preconditioners. The most popular implicit preconditioner is Incomplete LU (ILU) decomposition [2]. To be efficient, this decomposition must satisfy the following conditions:

- Preconditioner matrix M must be chosen "close" to the main matrix A in such a way that (for typical values of vector $\boldsymbol{x}$) the approximation error is sufficiently small: $||M\boldsymbol{x} - A\boldsymbol{x}|| = \varepsilon\,||\boldsymbol{x}|| \ll ||\boldsymbol{x}||$, or $\varepsilon \ll 1$.
- Matrix M must be suitable for decomposition into factors (e.g. $M = LU$) and these factors must be invertible with low computational cost, i.e. must allow economical solution of auxiliary linear systems $L\boldsymbol{y} = \boldsymbol{z}$ and $U\boldsymbol{x} = \boldsymbol{y}$.
- Solution of these linear systems must be subject to efficient parallelization.

Straightforward implementation of ILU preconditioner doesn't approximate the main matrix A with the required accuracy, i.e. $\varepsilon = O(1)$ [3]. As the result, its convergence properties are not good: $O(N)$ iterations are required as for explicit preconditioners (N is the dimension of a problem in one spatial direction), though the total number of iterations may become less. More accurate Modified ILU (MILU) preconditioner approximates the main matrix A with the accuracy $\varepsilon = O(h)$ (here h is the grid distance) resulting in $O(N^{\frac{1}{2}})$ iterations [4,3]. However, Modified ILU can't be accurately applied in some situations (e.g. for solving systems of equations and for the domain decomposition approach [2]).

On the other hand, both ILU and MILU can't be massively parallelized because of recursive nature of forward and backward sweeps ($\boldsymbol{L}\boldsymbol{y} = \boldsymbol{z}$, $\boldsymbol{U}\boldsymbol{x} = \boldsymbol{y}$). One idea is to use red-black grid numbering in order to parallelize sweeps. However, in this case ILU decomposition looses its implicit properties and demonstrated $O(N)$ behaviour. Other sorts of complicated explicit preconditioners (like approximate inverse) also propagate information slowly because of the limited stencil size and also belong to the $O(N)$ class.

For these reasons, simple explicit preconditioners remain attractive in some cases because of good parallelization properties. They will be considered in the next section, followed by the analysis of two sorts of implicit preconditioners: Modified ILU for Cartesian discretization in a regular domain, and plain ILU for general sparse matrices.

3 Optimization and Parallelization of Explicit Preconditioners

For the analysis of simple explicit preconditioners, solution of linear systems arising in the discretization of the Navier-Stokes equation is considered. For the components of velocity (u, v, w) these are non-symmetric linear systems to be solved by the BiCGStab, for pressure (p) the plain Conjugate Gradient method can be used. Sparse matrices of these linear systems are stored in the Compressed Row Storage (CRS) format. The test problem uses the Cartesian discretization within a spherical domain, number of grid points is about 320000. The algorithm is parallelized for shared-memory computer systems using OpenMP [5]. This method can be extended to the hybrid OpenMP/MPI environment.

The Polynomial Jacobi preconditioner of the 1-st order is used both in BiCGStab and CG solvers. If the original matrix is diagonally scaled: $\mathrm{diag}(A)=I$, the preconditioner will look as $B = 2\,I - A$. For the stable work, this preconditioner should be slightly underrelaxed by the following way: $B = I + \gamma\,(I - A)$. In the current implementation $\gamma = 0.985$.

The main computational kernel of both Conjugate gradient and preconditioning algorithms is the multiplication of a sparse matrix by a vector. Parallelization of this kernel in OpenMP is straightforward: the matrix is split into parts with equal number of rows, and each processor core independently computes the corresponding part of the resulting vector.

Parallel performance of such kind of computations is limited by the ability of the memory subsystem to read (write) data with the high speed. Therefore, data access rate requirements of the algorithm should be reduced. The Polynomial Jacobi preconditioner can be improved in this respect by using the single precision format (`real*4`) for storing a copy of the main matrix A for the preconditioning operator. Convergence properties of this preconditioner remain unchanged.

For the symmetric matrix, the CRS format is not convenient because it is not necessary to store its upper part. It is more economical and efficient to store only the strictly lower part L_A of the matrix A. If A is diagonally scaled, its representation will look as $A = L_A + I + L_A^{\mathrm{T}}$.

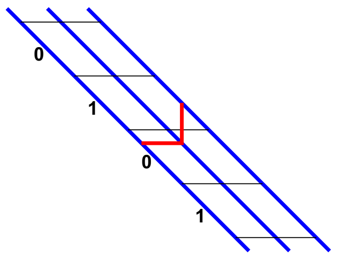
Multiplication of the symmetrically stored sparse matrix by a vector is more complex in comparison with the original storage scheme. The lower (L_A) and the upper (L_A^{T}) parts of the matrix are multiplied by a vector in different ways:

– for the lower part, scalar product of the densely packed row by the sparsely distributed elements of a vector is calculated, and the corresponding element of the resulted vector is modified;
– for the upper part, elements of the packed column are multiplied by the corresponding scalar value of a vector, and the sparsely distributed elements of the resulting vector are modified.

Parallelization of the symmetric sparse matrix multiplication algorithm is not straightforward. Unlike the original algorithm, the new one will not work correctly if we simply split all arrays by equal parts. As illustrated on Fig. 1 (left), parallel execution of the algorithm in different threads leads to the modification of the same elements of the resulting vector: thread 0 can modify elements in the data area of the immediately preceding thread 1 and corrupt its results.

In order to avoid this problem, the multicolored (or red-black) partitioning of data arrays can be applied. If we first split all arrays by equal parts in accordance with the number of threads, and then additionally split each part by two subparts of different color (marked as 0 and 1 on Fig. 1, left), we will be able to perform computations simultaneously in all subparts of the same color. After finishing processing for the particular color, we will do the same for another color. This approach is conceptually similar to the multicolored grid partitioning for some iterative methods (such as Gauss-Seidel and SOR).

The above variant of the multicolored partitioning has the obvious limitation: the maximal half-width of the matrix must be less than the size of the subpart otherwise a particular thread would modify the resulting vector of the preceding thread of the same color. In fact, this is a limitation on the number of threads for a given matrix. For the current example, this limitation is equal to 28, that is more than the number of processor cores in most available multiprocessor servers (usually 12 to 16). With increasing the problem size, this limit also will be



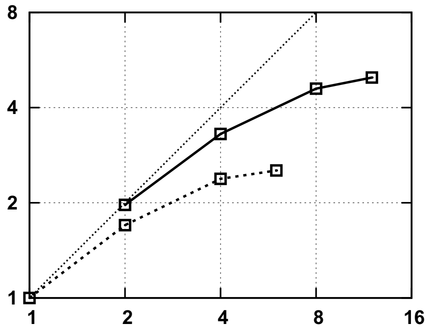


Fig. 1. Parallelization for the symmetric storage scheme (left). Parallelization results: dashed line – one processor, solid line – two processors (right)

increased. For long domains, it is recommended that grid nodes are enumerated in the direction of the short dimensions. In this case, the matrix bandwidth will be lower, and the thread limit will increase. Also, reducing the matrix width is useful from the performance point of view. However, if massive parallelization becomes necessary, it will be possible to develop more complex multicoloring approach similar to the multicolored grid partitioning.

Parallelization efficiency results for the above problem are shown on Fig. 1 (right). These results were obtained on the system with two 6-core Intel Xeon X5650 processors. Each processor has its own integrated 3-channel memory controller, therefore the peak memory access rate of the system is doubled. Parallelization tests were performed with 1 to 6 threads on one processor, and with 2 to 12 threads on two processors.

The one-processor results on Fig. 1 (right) demonstrate that the memory subsystem of a processor is almost saturated with 4 threads (additional performance increase with 6 threads is only about 6%). Similar saturation can be seen on the two-processor results. On the other hand, the two-processor results demonstrate almost linear performance increase in comparison to the single-processor runs with the same number of threads per processor (by about 1.95 times).

Thus, this test program is strongly memory-bound. The principal factor that limits parallel performance of such jobs is the memory throughput rate, while the CPU frequency is less important. The new generation Intel Xeon servers based on Sandy Bridge EP processors with 4 memory channels have much higher memory speed (51.2 GB/s vs 32 GB/s). As a result, parallel performance of similar jobs is proportionally increased. In the near future, after the switch from DDR3 to DDR4, processors with even faster memory subsystems will appear.

The above results demonstrate that explicit preconditioners have high parallelization potential and good scalability. The main factor that limits performance of explicit preconditioners is the peak memory access rate. For multiprocessor and multinode (cluster) computer systems, this limitation is scaled with the number of processors, thus increasing the total performance potential.

4 Implicit Preconditioners for Regular Domains

In this section we consider parallelization of the efficient Modified ILU (MILU) preconditioner for solving Poisson equation in rectangular parallelepipedic domains. As mentioned above, forward and backward sweeps of incomplete decomposition algorithms are recursive in their nature and can't be straightforwardly parallelized. Therefore, it is necessary to find such geometric properties of the algorithm that parallelization would become possible and efficient.

The original idea is taken from the twisted factorization of a tridiagonal linear system, when Gauss elimination is performed from both sides (for a subdiagonal and a superdiagonal, respectively). This idea can be naturally generalized to 2 and 3 dimensions. This method is called "nested twisted" [6,3]. An example of the nested twisted factorization is shown on Fig. 2 for two-dimensional case.

The nested twisted factorization method can be used for direct parallelization of the solution for up to 8 threads (in a Cartesian domain). The computational

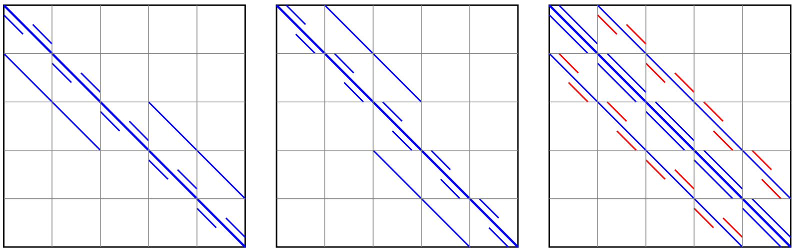


Fig. 2. Nested twisted factorization $L \cdot L^{\mathrm{T}} \to M$ suitable for parallelization

scheme of this method is as following. A rectangular parallelepipedic domain is split into 8 octants by separator planes (Fig. 3). In each octant, Gauss elimination is performed from the corner in the direction inwards (in all 3 dimensions), independently in different threads (Fig. 3, left).

After finishing eliminations in the internal points of octants, they are performed in quadrants of separator planes by the same way (Fig. 3, center). Then, intersection lines of separator planes are processed, and finally a solution at the central point is computed. The following backsubstitution is performed in the reverse order, from the central point in the direction outwards.

Parallelization for 16 threads needs another approach. For recursive algorithms, the staircase (or pipeline) method can be employed [7,3]. This method is illustrated on Fig. 3 (right). Each subdomain is split into 2 parts in the direction of `j` (see the bottom-left octant divided between threads 0 and 1). Computations in a plane `(i,j)` for a particular `k` can't be performed by thread 1 until they are finished by thread 0. However they can be fulfilled in a pipelined fashion: thread 1 computes a layer for some `k` at the same time when thread 0 computes the next layer for `k+1`. This method needs the synchronization between threads in a pair: before starting computations for some `k`, thread 1 must wait for thread 0 to finish computations in the same layer. At the backsubstitution stage of the algorithm, computations are performed in the reverse order.

Implementation of this method leads to some algorithmic overhead because at the beginning (for the first `k`) thread 1 is idle waiting for the results from thread 0, and at the end (for the last `k`) thread 0 is idle after finishing its work.

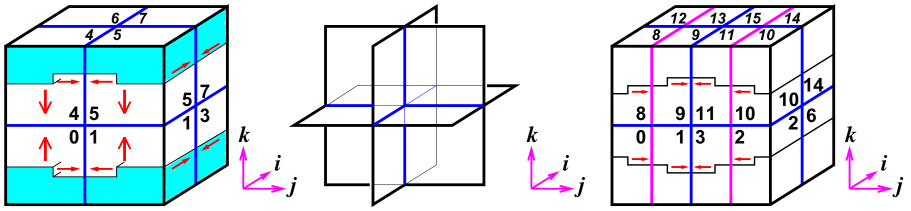


Fig. 3. Parallelization of the nested twisted factorization: illustration of the method (left); separator planes (center). Parallelization for 16 threads, staircase method (right)

The above method of parallelization can be used for more than two threads. In this case the algorithmic scheme will have more stairs and more points of synchronizations. However, because of performance overheads, the reasonable number of stairs can't be large. Therefore, parallelization potential of the above method can be estimated to be at most 32 or 64.

For computational domains of different shape, this potential depends on the number of corners. For example, a cylindrical domain has only 4 independent corners, and its parallelization potential will be at most 16 or 32. As a consequence, massive and efficient parallelization of MILU-class preconditioners for irregular domains and general sparse matrices is not possible at all.

The above parallelization method is "direct" with respect to the algebraic properties of the preconditioner matrix in such sense that the order of its approximation error remains the same as for the original (sequential) decomposition. For this reason, Modified ILU decomposition retains its convergence properties. This is not true, however, for the class of domain decomposition methods [2], when preconditioning accuracy is lost and convergence is sacrificed.

Parallelization efficiency results for the above method can be found in [3]. They are very similar to those in the previous section: the method demonstrates good efficiency if the memory subsystem is scaled with the number of threads, otherwise saturation is observed. Thus, this method is strongly memory-bound, and its performance depends firstly on the achievable memory throughput rate.

5 Implicit Preconditioners for Unstructured Grids

In this section we analyze parallelization approaches for Incomplete LU decomposition of general non-symmetric sparse matrices. These matrices are produced from the discretization of coupled systems of equations in irregular domains for the solution of stiff problems arising in different multiphysics applications (CFD, semiconductor transport, kinetic and quantum problems) [8,9].

Numerical solution of such ill-conditioned problems needs efficient CG-class solvers with ILU preconditioning. Here, parallelization of the preconditioned non-symmetric IDR solver will be considered [9]. The same ideas can be applied to any other CG-class solver for non-symmetric matrices (CGS, BiCGStab, GMRES).

Most computations in this solver are performed in two kernels – multiplication of the non-symmetric sparse matrix by a vector ($\boldsymbol{y} = A\boldsymbol{x}$), and solution of the decomposed linear system ($L\boldsymbol{y} = \boldsymbol{z}$, $U\boldsymbol{x} = \boldsymbol{y}$).

Parallelization of the first kernel is simple: if matrix A is stored in the CRS format, each thread can independently compute a part of the resulting vector. This is similar to the procedure described in the previous sections. This procedure has no strict limitations on the number of parallel threads.

On the other hand, procedure of Gauss elimination for a general sparse matrix can't be easily parallelized because such matrix does not possess any geometric parallelization properties. Thus it is necessary to look for algebraic approaches. The first idea is to use twisted factorization. An example of this factorization for a general banded sparse matrix is shown on Fig. 4.

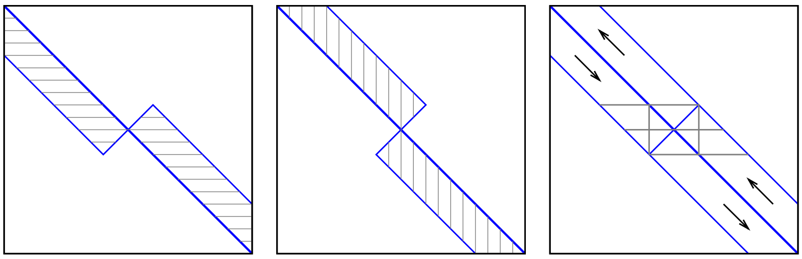


Fig. 4. Twisted factorization of the sparse matrix (left, center); illustration of the parallel Gauss elimination (right)

It can be seen that matrix factors have mutually symmetric portraits. These factors will be traditionally named as L and U. It is convenient to represent the decomposition as $M = (L + D)D^{-1}(D + U + R)$. Here R is the reverse diagonal that separates two part of each factor. The role of this reverse diagonal is seen on Fig. 4 (right) where the preconditioning matrix M is represented as the product of these factors. We can distinguish three areas on the matrix portrait: the central part (a square area with adjacent subareas on the left and on the right), the first part located above, and the last part located below.

Elimination of non-zero elements in the first and in the last parts is straightforward (as in the standard twisted-Gauss process) because these parts are formed by the simple multiplication of corresponding parts of factors L and U. Elimination of non-zero elements in the central part is much more complicated, because it is formed by the multiplication of complex central parts of L and U. Processing of the central part can't be parallelized and is performed serially. Parallel Gauss elimination is shown schematically on Fig. 4 (right).

This approach allows to parallelize the second kernel of the algorithm by 2 threads. It can be combined, for example, with 4-thread parallelization of the first kernel (multiplication of the main matrix by a vector) in order to achieve reasonable overall parallel efficiency on a multicore processor. This algorithm is also memory-bound (as those in the previous sections), and therefore its performance strongly depends on the throughput limitations of the memory subsystem.

Another parallelization approach for Gauss elimination is similar to the staircase (pipelined) method [7,3]. The idea of the approach is to split each matrix half-band (L and U) into pairs of adjacent trapezoidal blocks that have no mutual data dependences and therefore can be processed in parallel. As a result, parallelization of Gauss elimination for 4 threads will be implemented.

This new block-pipelined parallelization method is illustrated on Fig. 5. An example of the splitting of a matrix half-band is shown on Fig. 5 (center): the sub-diagonal part of L is split into adjacent pairs of blocks marked "0" and "1" (all other half-bands of matrices L and U are split into blocks similarly).

Let's consider the highlighted pair of blocks "0" (layer `k+1`) and "1" (layer `k`). It is seen that block "0" can be processed before finishing processing of block "1", provided the maximal column index of elements in "0" is less that the index

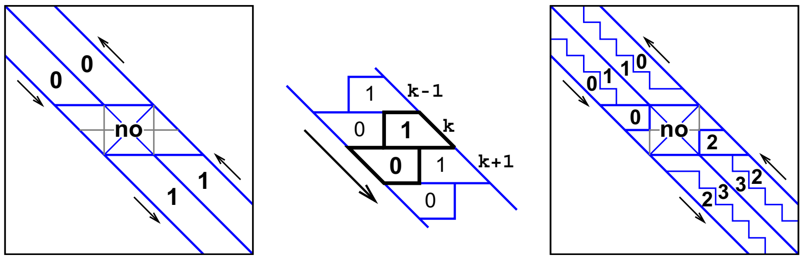


Fig. 5. Original twisted factorization method (left). Block-pipelined method: splitting of matrix L (center); combination of twisted and block-pipelined methods (right)

of the last (diagonal) element in the first row in "1". Due to this, each sweep of the Gauss procedure can be executed in two parallel threads – one for blocks "0", another for blocks "1", with the synchronization at the and of each step.

In order to implement the above scheme, it was necessary to construct the new storage scheme for all three parts of the matrix (first, last and central) and to implement proper synchronization technique for lightweight processes. Also, accurate splitting of matrices into blocks is needed in order to achieve good load balance. As a result, parallel 4-thread Gauss elimination method for a general non-symmetric sparse matrix can be developed. This combined method is illustrated on Fig. 5 (right).

This method is also "direct" with respect to the algebraic properties of the preconditioner matrix (as one described in the previous section). This is important for the solution of extremely ill-conditioned linear systems. It can be combined, for example, with 8-thread parallelization of the matrix-vector multiplication kernel in order to achieve good parallel efficiency on a two-processor system. In this case, the system's memory throughput rate will be doubled compared to a single processor, appropriately increasing performance of the solver.

Another way to increase performance is to use the single precision format (`real*4`) for the preconditioning matrix and, if possible, for the main matrix also. In this case memory access rate requirements of the algorithm will be reduced.

The considered block-pipelined method can be extended for more than two threads (for each Gauss elimination sweep). However, in this case, more accurate splitting of matrices is required that may be possible only for some sparsity patterns. When applied, this splitting will increase the parallelization potential of the kernel to 8 threads.

Parallelization efficiency results of the new method are presented in Table 1. These results were obtained for a CFD problem which matrix has 302500 unknowns and 44 non-zero elements in a half-band's row (average). Maximal half-width of the matrix is 2600. Measurements were performed on the 4-core Intel Core i7-920 processor with three DDR3-1333 memory channels.

These results demonstrate the reasonable parallelization efficiency of both methods – the twisted factorization alone, and its combination with the block-pipelined

Table 1. Parallelization results for a CFD problem

method		serial	twisted	twisted	block pipelined
threads		1	2	4	4
real*8	time	89.1 ms	55.6 ms	48.1 ms	41.0 ms
	speedup	1.00	1.60	1.85	2.17
real*4	time	89.1 ms	51.3 ms	41.6 ms	32.0 ms
	speedup	1.00	1.74	2.14	2.78

method. Another test matrix (MOSFET electronic device modeling, 77500 unknowns, 8 non-zero elements average, maximal half-width 4650) is much more sparse and less convenient for parallelization. Nevertheless, results measured for this matrix are very close (within 1-2%) to those presented in the table (with the exception of the `real*4` regime that wasn't tested).

From the table it can be seen that performance gain of using the `real*4` data format is 1.28. All these results illustrate the memory-bound property of the methods. Computational speed of the test problem on the more advanced Intel Sandy Bridge EP processor can be increased by about 1.5 times in accordance with the increase of its memory access rate. Additional speed increase can be achieved on a two-processor system with independent memory controllers.

More details on the above methods can be found in [9].

6 Conclusion

In this work we have analyzed parallelization properties and limitations of several most typical preconditioners for the Conjugate Gradient methods. This analysis confirmed that explicit preconditioners have good parallel potential and therefore remain attractive for massive parallelization. On the other hand, very efficient Modified ILU preconditioners can be moderately parallelized only for computational domains with regular geometry. In case of irregular geometry and general non-symmetric sparse matrix, only the plain ILU method can be limitedly parallelized without loosing its convergence properties. For the future development, the most promising and efficient approach is Multigrid. However, in many cases it is very difficult to implement this method. For coupled systems of equations and some other cases Multigrid can't be implemented yet due to lack of theory. For this reason, development of parallel ILU-class methods, as well as economical explicit preconditioners remains important.

Acknowledgements. This work was supported by the Russian Foundation for Basic Research (project RFBR-12-08-00034) and partly by the Institute of mathematics (IMATH) of the University of Toulon, France.

References

1. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method without the Agonizing Pain. School of Computer Science, Carnegie Mellon University, Pittsburgh (1994)
2. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing, Boston (2000)
3. 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)
4. Gustafsson, I.: A Class of First Order Factorization Methods. BIT 18, 142–156 (1978)
5. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering 5(1), 46–55 (1998)
6. van der Vorst, H.A.: Large Tridiagonal and Block Tridiagonal Linear Systems on Vector and Parallel Computers. Par. Comp. 5, 45–54 (1987)
7. Bastian, P., Horton, G.: Parallelization of Robust Multi-Grid Methods: ILU-Factorization and Frequency Decomposition Method. SIAM J. Stat. Comput. 12, 1457–1470 (1991)
8. Fedoseyev, A., Turowski, M., Alles, M., Weller, R.: Accurate Numerical Models for Simulation of Radiation Events in Nano-Scale Semiconductor Devices. Math. and Computers in Simulation 79, 1086–1095 (2008)
9. Bessonov, O., Fedoseyev, A.: Parallelization of the Preconditioned IDR Solver for Modern Multicore Computer Systems. In: Application of Mathematics in Technical and Natural Sciences: 4th International Conference. AIP Conf. Proc., vol. 1487, pp. 314–321 (2012)