World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:16, No:10, 2022

Parallel Pipelined Conjugate Gradient Algorithm on Heterogeneous Platforms

Sergey Kopysov, Nikita Nedozhogin, Leonid Tonkov

Abstract—The article presents a parallel iterative solver for large sparse linear systems which can be used on a heterogeneous platform. Traditionally, the problem of solving linear systems do not scale well on cluster containing multiple Central Processing Units (multi-CPUs cluster) or cluster containing multiple Graphics Processing Units (multi-GPUs cluster). For example, most of the attempts to implement the classical conjugate gradient method were at best counted in the same amount of time as the problem was enlarged. The paper proposes the pipelined variant of the conjugate gradient method (PCG), a formulation that is potentially better suited for hybrid CPU/GPU computing since it requires only one synchronization point per one iteration, instead of two for standard CG (Conjugate Gradient). The standard and pipelined CG methods need the vector entries generated by current GPU and other GPUs for matrix-vector product. So the communication between GPUs becomes a major performance bottleneck on miltiGPU cluster. The article presents an approach to minimize the communications between parallel parts of algorithms. Additionally, computation and communication can be overlapped to reduce the impact of data exchange. Using pipelined version of the CG method with one synchronization point, the possibility of asynchronous calculations and communications, load balancing between the CPU and GPU for solving the large linear systems allows for scalability. The algorithm is implemented with the combined use of technologies: MPI, OpenMP and CUDA. We show that almost optimum speed up on 8-CPU/2GPU may be reached (relatively to a one GPU execution). The parallelized solver achieves a speedup of up to 5.49 times on 16 NVIDIA Tesla GPUs, as compared to one GPU.

Keywords—Conjugate Gradient, GPU, parallel programming, pipelined algorithm.

I. Introduction

HIGHLY heterogeneous high performance computing (HPC) platforms, where multicore processors are coupled with GPUs, have been widely used in high performance computing as one approach to continuing performance improvement while managing the new challenge of energy efficiency [1]. Although some software packages and programming languages could be used directly, the introduction of multicore processors in HPC resulted in redesign of some critical software packages and significant refactoring of some existing parallel applications. Computing accelerators are contained in the computing nodes of supercomputers and are used quite successfully in solving many computing problems despite the fact that the central processor (CPU) is idle after running the core functions on the accelerator.

S. Kopysov, N. Nedozhogin and L. Tonkov are with the Institute of Mathematics, Information Technologies and Physics, Udmurt State University, Izhevsk, Russia (e-mail: s.kopysov@gmail.com, Nedozhogin07@gmail.com, letonkov@mail.ru).

Each architecture of the CPU and GPU has unique features and, accordingly, is focused on solving certain tasks for which typical, for example, high performance or low latency. Hybrid CPU/GPU computing is one method of realizing performance gains independent of the iterative method used. With hybrid CPU/GPU computing, we focus on separating the computationally intensive portions of the program among several workers, taking into account their different productivity.

Currently, many parallel algorithms have been proposed that provide high performance and scalability when solving large sparse systems of equations on modern multiprocessor with a hierarchical architecture.

Reference [2] proposes a combination of a hybrid CPU-GPU and a pure GPU software implementation of a direct algorithm for solving shifted linear systems with a large number of complex shifts and multiple right-hand sides.

Reference [2] proposes one of the implementations of the software of a direct algorithm for solving shifting linear systems with a large number of complex shifts and multiple right-hand sides. This implementation combines hybrid CPU and GPU and a pure GPU software implementation. The construction of hybrid solvers with a combination of direct and iterative methods for solving SLAE allows the use of several levels of parallelism [3]–[7]. The separate steps of the CG algorithm are executed on a CPU or GPU, such as a preconditioner [8].

In [9] task scheduling on Multi-CPUs/Multi-GPUs platforms for the classical CG from the `PARALUTION` library is executed in the `StarPU`. Its unified runtime system for heterogeneous multicore architectures is developed in the INRIA laboratory (France).

Our team considered one of the approaches, a hybrid method, for solving systems of equations of Schur complement, by preconditioning iterative methods from Krylov subspaces which were built and implemented when used together the cores of CPUs and GPUs [10]. The classical preconditioned conjugate gradient method (PCG) [11] was applied for the block ordered matrix and the separation of calculations in matrix operations between the CPU and one or more GPUs, when the system of equations in Schur complement was solved in parallel.

The Krylov subspace methods are some of the most effective options for solving large-scale linear algebra problems. However, the classical Krylov subspace algorithms do not scale well on modern architectures due to the bottleneck related to synchronization of computations. Pipeline methods of the Krylov subspace [12] with hidden communications provide high parallel scalability due to the global communications overlapping with computing, performing matrix-vector and dot products. The first work on reducing communications was related to a variant of the conjugate gradient method, having one communication at each iteration [13], using the three-term recurrence relations CG [14].

The next stage of development was the emergence of s-step methods from Krylov subspaces [15], in which the iterative process in the s-block uses various bases of Krylov subspaces. As a result, it was possible to reduce the number of synchronization points to one per s iterations. However, for a large number of processors (cores), communications can still take significantly longer than computing a single matrix-vector product. In [16] a CG algorithm was proposed using auxiliary vectors and transferring a sequential dependence between the computing of matrix-vector product and scalar products of vectors. In this approach, the latency of communications is replaced by additional calculations.

In this paper, we consider the pipelined variant, which is potentially better for heterogeneous multi-CPUs/multi-GPUs computing, since it requires only one synchronization point per iteration, instead of two for standard CG.

This paper presents a pipeline technique for conjugate gradient method and discusses its parallel implementation on multi-CPU/multi-GPU platform for solving large sparse linear systems. Hybrid parallel computing approaches are adopted to significantly improve performance of the solver. Specifically, we introduce a hybrid solution by fully utilizing multi-core nodes available through multi-threading techniques by means of OpenMP, and exploit an access to massively parallel hardware through GPU-offloading with CUDA, in which data are transferred to the GPU for processing. The combination of GPU-offloading and CPU-threading is explored through a hybrid CPU/GPU compute implementation.

In our work, we modify the basic CG algorithm to minimize the cost of collective communication. A modified but mathematically equal variant of the conjugate gradient algorithm is employed to reduce the cost of global communication. By using the modified algorithm, the three vector dot products in each iteration can be done simultaneously with only one nonblocking collective communication that can be further overlapped with other operations.

II. Pipelined Algorithm of the Conjugate Gradient Method

We consider now the pipelined version of the conjugate gradient method, which is mathematically equivalent to the classical form of the preconditioned CG method and has the same convergence rate.

In Algorithm 1, the modification of the vectors r_{j+1}, x_{j+1}, s_{j+1}, p_{j+1} and matrix-vector products provides pipeline computations. The computation of dot products (line 4) can be overlapped with the computation of the product by the preconditioner (line 2) and the matrix-vector product (line 3). However, the number of triads in the algorithm increases to eight, in contrast to three for the classic version and four in [15]. In this case, a parallel computation of triads and two dot products at the beginning of the iterative process and one synchronization point is possible.

Algorithm 1 Pipelined Algorithm CGwO.

1 $r = b - Ax$
2 $u = M^{-1}r$
3 $w = Au$
4 $\gamma_1 = (r, u)$
5 $\delta = (w, u)$
6 **while** $||r||_2/||b||_2 > \varepsilon$ **do**
7     $m = M^{-1}w$;
8     $n = Am$
9   **if** $(j = 0)$ **then**
10         $\beta = 0$
11   **else**
12         $\beta = \gamma_1/\gamma_0$
13   **end if**
14     $\alpha = \gamma_1/(\delta - \beta\gamma_1/\alpha)$
15     $z = n + \beta z$; $w = w - \alpha z$; $s = w + \beta s$; $r = r - \alpha s$
16     $p = u + \beta p$; $x = x + \alpha p$; $q = m + \beta q$; $u = u + \alpha q$
17     $\gamma_0 = \gamma_1$
18     $\gamma_1 = (r, u)$; $\delta = (w, u)$
19 **end while**

The pipelined version CG presented in this work can be used with any preconditioner. There are two ways to organize computations in the preconditioned pipelined CG, which provide a compromise between scalability and the total number of operations. Thus, the CG pipeline scheme is characterized by a different order of computations, the presence of global communication, which can overlap with local computations, such as matrix-vector product and operations with a preconditioner, and the possibility of organizing asynchronous communications.

The two variants of the conjugate gradient method were compared: the classical scheme and the pipelined one. Table I presents the results of numerical experiments where the execution time of a sequential version of the classical CG and the CGwO pipelined scheme (Algorithm 1) executed on the CPU and GPU are shown. Note that in the variants for the GPU, joint computation of all dot products of vectors in one kernel function was implemented, independently of each other. For this, when starting the CUDA kernel, the dimension of the `Grid` hierarchy of CUDA threads was set in two-dimensional form: 3 sets of blocks, each for performing computations on its own pair of vectors. This allowed us to reduce the number of exchanges between the CPU and GPU memory, combining all the resulting scalars in one communication.

Matrices from the SuiteSparse Matrix Collection [17] were used in the test computations. The right hand side vector was formed as a row-wise sum of matrix elements. Thus, the solution of the system $Ax = b$, dimension $N \times N$ (with the number of nonzero elements nnz) is a vector $x = (1, 1, \dots, 1)^T$.

For systems of equations of small dimension, the solution time on the CPU according to the classical CG scheme is significantly less than the GPU execution time for the same number of iterations (see Table I). For large systems, the costs of synchronization and forwarding between the CPU and GPU overlap with the speed of the GPU. In the pipelined version of CGwO, the computational execution costs on the GPU are reduced almost threefold for all the considered systems of equations only due to the reduction of exchanges between the GPU and the CPU in the computation of dot products.

III. CG with the Combined Use of CPU and GPU

Let us consider the application of the Algorithm 1 for the parallel solution of super-large systems of equations on computing nodes, each of which contains several CPUs and GPUs. To solve SLAEs on several GPUs, we construct a block pipelined algorithm for the conjugate gradient method. On heterogeneous platform, data exchange between different GPUs within the same computing node is carried out with OpenMP technology, and the exchange between different computing nodes is carried out by MPI technology. For example, we consider a node containing a central eight-core processor and two graphics accelerators. The number of OpenMP threads is selected by the number of available CPU cores. The first two OpenMP threads are responsible for exchanging data and running on two GPUs. Threads 2–6 provide computations on the CPU and can perform computations on a block of the SLAE matrix. The last thread provides data exchange with other computing nodes by MPI.

A. Matrix Partitioning

To divide the matrix A into blocks, we construct the graph $G_A(V, E)$, where $V = \{i\}$ is the set of vertices associated with the row index of the matrix (the number of vertices is equal to the number of rows of the matrix A); $E = \{(i, j)\}$ is the set of edges. Two vertices i and j are considered to be connected if the matrix A has a nonzero element with indices i and j. The resulting graph is divided into subgraphs whose number is d. For example, to split a graph, we can use the [18] layer-by-layer partitioning algorithm, which reduces communication costs due to the need to exchange only with two neighboring computing nodes.

After that, each vertex of the graph is assigned its own GPU or CPU. On each computing unit, the vertices are divided into internal and boundary. The latter are connected with at least one vertex belonging to another subgraph.

After partitioning, each block A_k of the original matrix A contains the following submatrices:

- $A_k^{[i_k,i_k]}$ – matrix associated with the internal vertices;
- $A_k^{[i_k,b_k]}$, $A_k^{[b_k,i_k]}$ – matrices associated with the internal and boundary vertices;
- $A_k^{[b_k,b_l]}$ – matrix associated with the boundary vertices of the k-th and l-th blocks.

Then the matrix A can be written in the following form:

$$A = \begin{pmatrix} A_1^{[i_1,i_1]} & A_1^{[i_1,b_1]} & \cdots & 0 & 0 \\ A_1^{[b_1,i_1]} & A_1^{[b_1,b_1]} & \cdots & 0 & A_1^{[b_1,b_d]} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & A_d^{[i_d,i_d]} & A_d^{[i_d,b_d]} \\ 0 & A_d^{[b_d,b_1]} & \cdots & A_d^{[b_d,i_d]} & A_d^{[b_d,b_d]} \end{pmatrix}.$$

We divide the matrix-vector product $n = Am$ into two components by using the obtained partition:

$$n_k^b = A_k^{[b_k,i_k]} m_k^i + \sum_{l=1}^{l \le d} A_k^{[b_k,b_l]} m_l^b, \quad n_k^i = A_k^{[i_k,i_k]} m_k^i + A_k^{[i_k,b_k]} n_k^b. \tag{1}$$

Here k corresponds to the computing device. The block representation of the vectors involved in the algorithm is inherited from the matrix partitioning. For example, the vector m has the form $m^T = \left(m_1^i, m_1^b, \ldots, m_k^i, m_k^b, \ldots, m_d^i, m_d^b\right)$. The implementation of the matrix-vector product reduces the cost of communication between blocks at each iteration of conjugate gradient method. To perform this operation, an exchange of vectors m_k^b is required, the size of which is less than the dimension of the initial vector m.

The partitioning of the preconditioner M is carried out in a similar way.

B. Block Pipelined Algorithm

The matrix blocks were mapped on the available CPU and GPU with the block partitioning of the matrix and vectors. The number and size of blocks let on to map the load in accordance with the performance of the computing units, including the allocation of several blocks to one.

Let us represent parallel block scheme of the method CGwO that is performed each k-th computing unit in the form of Algorithm 2. Two parallel branches of this algorithm are executed accordingly on the CPU and CPU/GPU. Operations performed in parallel are shown in one line of the algorithm. Vector operations on each computing unit occur in two stages, for internal and boundary nodes. The designations of the internal and boundary nodes for vectors are omitted, with the exception of the matrix-vector multiplication. Dot products are performed independently by each computing unit on its parts of vectors. The summation of intermediate scalars occurs in parallel threads responsible for communication, which is the synchronization point at each iteration of the algorithm.

In block CGwO, compared to Algorithm 1, the preconditioning step has been moved (line 7 to line 20). This is done in order to combine vector operations on the computing unit and the assembly of the vector parts of the right hand side to perform matrix-vector multiplication in preconditioning. The 31 line on the right uses the ternary operator: if $j = 0$, then $\beta = 0$, in other cases $\beta = \gamma_1/\gamma_0$. The subscript h is used for vectors that are stored only in CPU memory.

C. MPI+OpenMP+CUDA Programming Model

Numerical experiments on the Algorithm 2 were carried out on heterogeneous platform with various configuration

Algorithm 2 Block Algorithm CGwO Performed on k-th Device

Require: Matrix partitioning into blocks $A_k^{[i_k,i_k]}$, $A_k^{[i_k,b_k]}$, $A_k^{[b_k,i_k]}$, $A_k^{[b_k,b_l]}$.

1 $r = b$
2 $u = M^{-1}r$
Parallel algorithm branches
$(\textbf{CPU} \vee \textbf{GPU})_k$
3 $w_k^i = A_k^{[i_k,i_k]} \cdot u_k^i + A_k^{[i_k,b_k]} \cdot u_k^b$
4 $w_k^b = A_k^{[b_k,b_k]} \cdot u_k^b + A_k^{[b_k,i_k]} \cdot u_k^i$
5 $w_k^b = w_k^b + w_h^b$
6 $m = M^{-1}w$
7 $\gamma_{1k} = (r_k, u_k)$; $\delta_k = (w_k, u_k)$
8 **while** $||r||_2/||b||_2 > \varepsilon$ **do**
9 $\quad n_k^i = A_k^{[i_k,i_k]} \cdot m_k^i + A_k^{[i_k,b_k]} \cdot m_k^b$
10 $\quad n_k^b = A_k^{[b_k,b_k]} \cdot m_k^b + A_k^{[b_k,i_k]} \cdot m_k^i$
11 $\quad n_k^b = n_k^b + n_h^b$
12 $\quad z = n + \beta z$
13 $\quad w = w - \alpha z$
14 $\quad q = m + \beta q$
15 $\quad s = w + \beta s$
16 $\quad p = u + \beta p$
17 $\quad x = x + \alpha p$
18 $\quad r = r - \alpha s$
19 $\quad u = u + \alpha q$
20 $\quad m = M^{-1}w$
21 $\quad \gamma_0 = \gamma_1$
22 $\quad \gamma_{1k} = (r_k, u_k)$; $\delta_k = (w_k, u_k)$
23 **end while**

CPU
24 Assembly of the vectors u_k^b
25 $w_h^b = \sum_{l=1,l\neq k}^{l\leq d} A_k^{[b_k,b_l]} \cdot u_k^b$
26 Copying w_h^b on the GPU$_k$
27 Assembly of the vectors m_k^b
28 Assembly $\delta = \sum_k \delta_k; \gamma_1 = \sum_k \gamma_{1k}$
29 $n_h^b = \sum_{l=1,l\neq k}^{l\leq d} A_k^{[b_k,b_l]} \cdot m_k^b$
30 Copying n_h^b on the GPU$_k$
31 $\beta = ((j = 0)\,?\,0 \,:\, \gamma_1/\gamma_0)$
32 $\alpha = \gamma_1/(\delta - \beta\gamma_1/\alpha)$
33 Assembly of the vectors w_k^b
34 Assembly vectors m_k^b
35 Assembly $\delta = \sum_k \delta_k; \gamma_1 = \sum_k \gamma_{1k}$

of computing nodes containing several CPUs and GPUs. In the general case, the parallel computing on several heterogeneous computing nodes containing one or more CPUs and several GPUs is implemented by the combination of several technologies: MPI, OpenMP and CUDA. In this article, the approach is to properly divide the computational workload between the CPU and the GPU, so that the CPU can aid the GPU in sharing the computational costs. Our programming strategy is based on implementation strategy, where a hybrid MPI+CUDA+OpenMP programming model is used to realize concurrent CPU+GPU computations. The principal concept for the strategy is to overlap computation with communication using OpenMP's nested parallelism capability to generate two independent groups of threads. The first thread group handles the CUDA, MPI communication and computation of the halo boundary points on the CPU using OpenMP threads. The second thread group computes the interior points on the CPU.

Let us consider the software organization of computations using as example some cluster, which includes two computing nodes (8-CPU cores and 2-GPUs). Each computing node is associated with a parallel MPI process. In a parallel process, 9 parallel OpenMP threads are generated, which is one more than the available CPU cores. The eighth OpenMP thread is responsible for communications between different computing nodes (using MPI technology, vector assembly using the `Allgatherv` function, adding scalars `Allreduce`) and various GPUs. In the 2 Algorithm, the operations performed by this thread are presented to the right. Zero and first OpenMP threads are the host threads for one of the available GPU devices and are responsible for transfer data between the GPU/CPU (calls to asynchronous copying functions) and auxiliary computations. Each available GPU device (further considered as a computing unit) is associated with one of the parallel OpenMP threads, which is responsible for transferring data between the GPU and CPU (calls to asynchronous copy functions) and participates with the eighth treads in matrix-vector product on boundary vertices (lines 25, 29 right column). The remaining parallel threads (second to seventh) perform the calculations as a separate computing unit for their matrix block. The operations performed by computing units in the 2 Algorithm are shown on the left.

The preconditioning in lines 2, 6 and 20 implies the use of block matrix-vector multiplication of the form (1) considered above.

IV. Numeric Results

To evaluate the presented algorithm, two series of tests were carried out: on synthetically generated matrices from HPCG test [19] and matrices taken from the University of Florida Sparse Matrix Collection. [20].

World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:16, No:10, 2022

A. Benchmarking with HPCG Matrices

The High Performance Conjugate Gradient (HPCG) [19] is a benchmark program that solves a sparse linear system arising in solving a three-dimensional heat diffusion problem. HPCG intends to solve the linear system generated from the finite difference discretization of the Poisson equation: $-\triangle u = b$, with homogeneous Dirichlet boundary conditions applied along the boundary of a three-dimensional cubic domain Ω. Based on a semistructured mesh with equidistant mesh spaces in the x, y and z directions, respectively, the discretization employed in HPCG leads to a second-order accurate 27-point stencil.

The resulting sparse linear system has the following properties: A — sparse matrix with 27 nonzero entries per row for interior equations and 7 to 18 nonzero terms for boundary equations; A — symmetric, positive definite, nonsingular linear operator.

We generate a synthetic symmetric positive definite (SPD) matrix A using an array-of-pointers-style compressed sparse row format, an exact solution vector of all 1.0 values, a corresponding right-hand-side vector b, and initial guess for x of all 0.0 values. The sparsity pattern of the synthetic matrix is really a regular 27-point 3-dimensional stencil pattern.

We tested one CPU performance on the Uran cluster based on three typical data sizes, including:

1) $125 \times 125 \times 160$, $nnz = 66503662$;
2) $160 \times 160 \times 201$, $nnz = 137318884$;
3) $200 \times 200 \times 250$, $nnz = 267487792$;
4) $250 \times 250 \times 310$, $nnz = 519219712$;
5) $310 \times 310 \times 390$, $nnz = 1005862912$;
6) $390 \times 390 \times 485$, $nnz = 1986735009$;

In all variants, the pipelined version of CG converged in 43 iterations with ε equal to 10^{-6}. For example, the solving times of the first three data sizes on one OpenMP thread were 8.39, 17.41, 33.22 seconds, respectively. We were able to obtain the result only for the first two sizes: 1.33, 2.72 when using single GPU. The remaining data sizes are not placed in the memory of one graphics accelerator. These results allow us to estimate that the performance of one OpenMP thread is approximately 6.5 times lower than the performance of single GPU for the linear system solving by the conjugate gradient method.

Performance matrices are executed and compared through scalability studies and absolute runtime results. To estimate the computational performance and the impact of MPI and OpenMP communications, our numerical experiments were executed with different numbers of CPUs and GPUs. The results are shown in Figs. 1 and 2. Each diagram corresponds to the number of cluster nodes (n-CPU) involved in the computations and the number of graphics accelerators on each node (m-GPU).

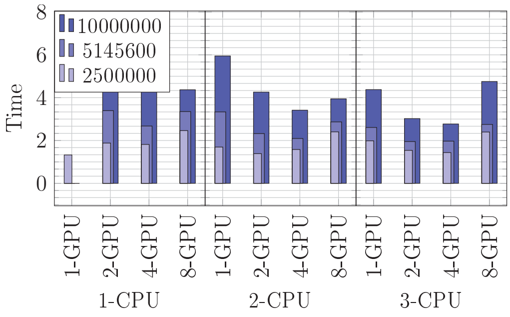

Fig. 1 Scalability of the block algorithms CGwO by CPU and GPU

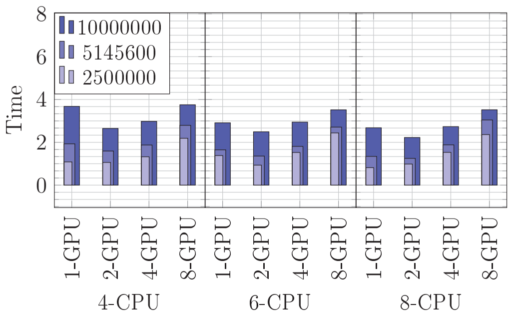

Fig. 2 Scalability of the block algorithms CGwO by CPU and GPU(continue)

Fig. 3 shows the results by subdomains for the case when 2 GPUs are used per computational node. Here, the matrix size of the linear system is approximately doubled. It can be noted that, starting from a size equal to 10,000,000, there is a good scalability of the algorithm. The problem execution time remains practically unchanged by doubling the problem size and doubling the number of subdomains.

B. Benchmarking on the SuiteSparse Matrix Collection

The results of comparing two algorithms of the conjugate gradient method on SLAEs containing test matrices [19] are considered. The problems range from small matrices, used as counter-examples to hypotheses in sparse matrix research, to large test cases arising in large-scale computation.

In the standard PCG algorithm, three dot products need to be done per iteration, with each one requiring a global collective communication that may substantially degrade the scalability at scale. In order to reduce the global communication overhead, we employ a reformulated but mathematically equivalent variant of the basic PCG algorithm, the pipelined Block PCG.

As shown in Algorithm 2, the pipelined PCG method has two advantages. First, only one global reduction is required for

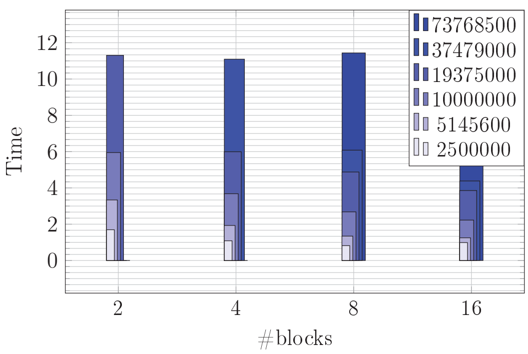

Fig. 3 Scalability of the block algorithms CGwO by number blocks

World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:16, No:10, 2022

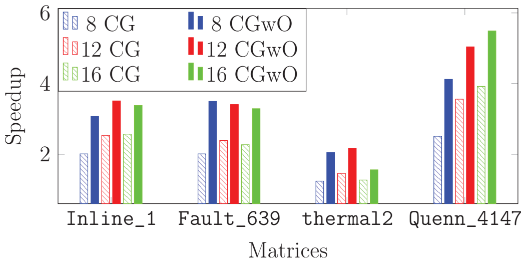

Fig. 4 Speedup of the block algorithms CG and CGwO

each iteration. Second, the global reduction can be overlapped with the matrix-vector product and with the application of the preconditioner.

Fig. 4 presents the results of accelerating the block algorithms of the conjugate gradient method, when divided into a larger number of blocks, accordingly 8, 12 and 16. To compute the speedup, parallel application was run repeatedly with different mapping of subdomains to several CPUs and GPUs. For example, in the case of 12 subdomains, variants were considered: 2 CPUs with 6 GPUs, 3 CPUs with 4 GPUs, 6 CPUs with 2 GPUs. The best time is shown.

The results of comparing two algorithms of the conjugate gradient method on SLAEs containing test matrices are presented in Table I. The results are given for several types of computing nodes using a single graphics accelerator.

The matrices are ordered by increasing the order of the system of equations (N) and the number of nonzero elements (nnz). Bold indicates the best time to solve the system in each case. The pipelined algorithm CGwO showed a reduction in execution time on small SLAEs which are characterized by a small computing load, due to which a reduction in communications provides less time. Note that the classic CG algorithm was implemented based on CUBLAS, while the CGwO variant uses matrix and vector operations of its own GPU implementation.

For systems `Inline_1` and `Fault_639`, the execution time of the pipelined algorithm is 10 and 13.5% longer than the block version of CG, which is associated with additional vector operations that are not blocked by reduced communications. With a decrease in the number of iterations, for example, for solving a large system with `G3_circuit`) with an approximately equal number of equations with `thermal2`, the execution time of the CG and CGwO algorithms on one GPU increases slightly. For the system (`thermal2` and `G3_circuit`) the increase in costs becomes more significant.

Table II presents the results of the block variant of the algorithms for computing on several computing nodes for systems with small dimension matrices. Here are the results for 2 and 3 subdomains. Each subdomain was considered on a separate computing node. Communications were carried out using MPI technology. A significant influence of network characteristics on the performance of block methods can be seen in Table II for system of equations with matrix `1138_bus`. Computations for these SLAEs were performed at various computing nodes with different throughput and latency of the network. In numerical experiments on the CNs (partition "debug") connected by a Gigabit network, communication costs significantly increase the execution time of the CG algorithm. For example, in the variant `1138_bus` on the cluster partition "debug", the execution time of the pipeline algorithm is 3.6 times less (the line "debug" in Table II and any row in Table I). Using the Infiniband 20 Gb/s communication network reduces the execution time for all presented systems of equations (lines "M2090" and "K40m").

When reducing the computational load, a decrease in the number of synchronization points and the consolidation of transfers per transaction is more pronounced. This shows a comparison of systems with matrices `Kuu` and `Muu`. Both systems have an equal number of equations and nonzero elements, but the conditionality of these matrices is significantly different and, as a consequence, the number of iterations in the conjugate gradient method is different. Table I shows that using the pipeline algorithm for the matrix `Muu` gives speedup by 70 times, compared with the matrix `Kuu`, where the speedup is only 2.8.

The speedup was considered relative to the option on one GPU from Table I. An application that implements this algorithm was executed in the exclusive mode of the computing node but not of the network.

As can be seen from the presented results, the pipelined CG shows the speedup greater than the classic version of conjugate gradient method. Wherein, for the largest of the considered matrices `Quenn_4147`, the speedup achieves 5.49 times, while the classical version gives 3.92 as maximum. For the strongly sparse matrix `thermal2`, block algorithms do not give high speedup (maximum is 1.56), since the computational load depends mainly on the number of the nonzero elements.

An analysis of the results showed that reducing the data size due to the matrix partitioning and reducing the synchronization points slightly decrease the impact of communication costs on the total algorithm performance. Only the use of computing nodes connected by Infiniband allowed us to get speedup when computing on several computing nodes. The matrix partitioning into blocks allowed to decrease the execution time of the pipelined block algorithm in comparison with the conjugate gradients on one node on the matrices `Inline_1`, `Fault_639` by reducing the computational load on one GPU.

Large systems `thermal2`, `G3_circuit`, solved by the block of the CGwO algorithm, as well as the reduction in communications costs and synchronization points, do not overlap the increasing costs of additional vector operations.

V. Conclusion

The heterogeneous computing platforms containing and sharing CPU + GPUs provide an effective solution to a wider range of problems with high energy efficiency when CPU and GPUs are uniformly loaded.

The parallel implementation of the solution of systems of linear algebraic equations on a heterogeneous platform was considered. The performance of parallel algorithms for classical conjugate gradient method is significantly limited by synchronization points when using the CPU and GPU together. A pipelined algorithm of the conjugate gradient method with one synchronization point was proposed. Also,

TABLE I
STATISTICS OF THE TEST PROBLEMS: PROBLEM NAMES, DIMENSIONS (N), NUMBER OF NONZEROS (nnz), DEVICE TYPE (DT) AND PROBLEM ANALYSIS IN TERMS OF THE TIMING IN SECONDS

Matrix	N	nnz	# iter.	DT	Time, s	
					CG	CGwO
Plat362	362	5786	991	M2090	6.88E-01	**3.07E-01**
				K40m	4.13E-01	3.12E-01
1138_bus	1138	4054	717	M2090	3.81E-01	**1.84E-01**
				K40m	5.31E-01	2.01E-01
				debug	6.82E-01	1.90E-01
Muu	7102	170134	12	M2090	2.64E-01	4.68E-03
				K40m	3.31E-01	**4.55E-03**
Kuu	7102	340200	378	M2090	4.31E-01	**1.31E-01**
				K40m	4.39E-01	1.35E-01
Pres_Poisson	14822	715804	661	M2090	6.72E-01	3.13E-01
				K40m	6.346E-01	**2.73E-01**
Inline_1	503712	36816342	5642	M2090	4.74E+01	5.17E+01
				K40m	**3.06E+01**	3.37E+01
Fault_639	638802	28614564	4444	M2090	3.83E+01	4.32E+01
				K40m	**2.44E+01**	2.77E+01
				debug	2.44E+01	2.77E+01
thermal2	1228045	8580313	2493	M2090	1.35E+01	1.82E+01
				K40m	**8.33E+00**	1.18E+01
G3_circuit	1585478	7660826	592	M2090	3.43E+00	4.32E+00
				K40m	**1.94E+00**	2.92E+00
Quenn_4147	4147110	399499284	8257	M2090	5.46E+02	5.78E+02
				K40m	**3.55E+02**	3.75E+02

TABLE II
TIME OF SOLVING BY THE BLOCK ALGORITHMS CG AND CGWO ON CPU/GPU, S

Matrix/DT	CG/#blocks		CGwO/#blocks	
	2	3	2	3
Plat362/M2090	1.55E+00		**1.22E+00**	
/K40m	1.92E+00	1.56E+00	1.28E+00	1.31E+00
1138_bus/M2090	1.84E+00		**9.28E-01**	
/K40m	1.90E+00	1.85E+00	1.03E+00	1.04E+00
/debug	1.25E+01		**5.36E+00**	
Muu/M2090	6.12E-01		2.29E-01	
/K40m	6.59E-01	5.64E-01	2.89E-01	**2.88E-01**
Kuu/M2090	1.30E+00		**6.43E-01**	
/K40m	1.29E+00	1.36E+00	6.81E-01	7.95E-01
Pres_Poisson/M2090	1.55E+00		**9.57E-01**	
/K40m	1.60E+00	1.66E+00	1.02E+00	1.19E+00
G3_circuit/M2090	4.27E+00		3.99E+00	
/K40m	4.04E+00	3.510E+00	3.27E+00	**2.77E+00**

it is provided the possibility of asynchronous computations, load balancing between several GPUs located both on the same computing node and for a GPU cluster when solving systems of large-dimensional equations. To further increase the efficiency of calculations, it is supposed to study not only the communication load of the algorithms but also the distributing of the computational load between the CPU and GPU. To obtain more reliable evaluation of communications costs, it is necessary to conduct a series of computational experiments on supercomputer with a completely exclusive mode of operation and a large number of heterogeneous nodes.

The following conclusions can be drawn from the analysis of data obtained during numerical experiments: the use of a pipeline algorithm reduces communication costs, but increases computational ones. For systems of small sizes or with a small number of iterations, this reduces the execution time of the algorithm when using a single GPU. For systems of large dimensions, a reduction in execution time, in comparison with CG, is possible only with a sufficiently small partition of the matrix into blocks, in which the increased computing costs overlap the communication decrease.

The proposed block algorithms, in addition to reducing the execution time, allow solving large linear systems that requires memory resources not provided by one GPU or computing node. At the same time, the pipelined block algorithm reduces the overall execution time by reducing synchronization points and combining communications into one message.

ACKNOWLEDGMENT

The research was performed using computing resources of the collective use center of IMM UB RAS ”Supercomputer center of IMM UB RAS”.

This research was funded by the Ministry of Science and Higher Education of the Russian Federation in the framework of state assignment No. 075-00928-21-01, project FEWS-2020-0010.

REFERENCES

[1] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, Jul. 2015. [Online]. Available: https://doi.org/10.1145/2788396

[2] N. Bosner, Z. Bujanović, and Z. Drmač, "Parallel solver for shifted systems in a hybrid CPU–GPU framework," *SIAM Journal on Scientific Computing*, vol. 40, no. 4, pp. C605–C633, 2018.

[3] E. Agullo, L. Giraud, A. Guermouche, and J. Roman, "Parallel hierarchical hybrid linear solvers for emerging computing platforms," *Comptes Rendus Mecanique*, vol. 339, no. 2, pp. 96–103, Feb. 2011.

[4] J. Gaidamour and P. Hénon, "A parallel direct/iterative solver based on a Schur complement approach." in *IEEE 11th International Conference on Computational Science and Engineering*, IEEE, Ed., Sao Paulo, Brazil, Jul. 2008, pp. page 98–105, 8 pages double colonnes. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00353547

[5] L. Giraud, A. Haidar, and Y. Saad, "Sparse approximations of the Schur complement for parallel algebraic hybrid linear solvers in 3D," INRIA, Research Report RR-7237, Mar. 2010. [Online]. Available: https://hal.inria.fr/inria-00466828

[6] S. Rajamanickam, E. G. Boman, and M. A. Heroux, "Shylu: A hybrid–hybrid solver for multicore platforms," in *In Proc. of 26th IEEE Intl. Parallel and distributed Processing Symp. (IPDPS'12). IEEE*, 2012.

[7] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov, "Domain decomposition preconditioners for communication-avoiding krylov methods on a hybrid cpu/gpu cluster," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC 14)*, IEEE. New Orleans, LA: IEEE, 2014-11 2014.

[8] A. Jamal, M. Baboulin, A. Khabou, and M. Sosonkina, "A hybrid CPU/GPU approach for the Parallel Algebraic Recursive Multilevel Solver pARMS," in *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2016, pp. 411–416.

[9] N. Kasmi, M. Zbakh, and A. Haouari, "Performance analysis of preconditioned conjugate gradient solver on heterogeneous (multi-CPUs/multi-GPUs) architecture," *Lecture Notes in Networks and Systems*, vol. 49, pp. 318–336, 2019.

[10] S. Kopysov, I. Kuzmin, N. Nedozhogin, A. Novikov, and Y. Sagdeeva, "Scalable hybrid implementation of the schur complement method for multi-gpu systems," *J. Supercomput.*, vol. 69, no. 1, p. 81–88, Jul. 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1209-7

[11] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, pp. 409–436, 1952.

[12] J. Cornelis, S. Cools, and W. Vanroose, "The communication-hiding conjugate gradient method with deep pipelines," 2019.

[13] E. D'Azevedo, V. Eijkhout, and C. Romine, "A matrix framework for conjugate gradient methods and some variants of cg with less synchronization overhead." 01 1993, pp. 644–646.

[14] J. Wilkinson and C. Reinsch, *Linear Algebra*, Berlin, Heidelberg, 1971.

[15] A. Chronopoulos and C. Gear, "s-step iterative methods for symmetric linear systems," *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153 – 168, 1989.

[16] P. Ghysels and W. Vanroose, "Hiding global synchronization latency in the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014, 7th Workshop on Parallel Matrix Algorithms and Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819113000719

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," vol. 38, no. 1, 2011.

[18] I. R. Kadyrov, S. Kopysov, and A. K. Novikov, "Partitioning of triangulated multiply connected domain into subdomains without branching of inner boundaries," *Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, vol. 160, no. 3, pp. 544–560, 2018, (In Russian).

[19] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016. [Online]. Available: https://doi.org/10.1177/1094342015593158

[20] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. Dongarra, "Matrix market: a web resource for test matrix collections," in *Quality of Numerical Software*, 1996.