ISSN 2070-0482, Mathematical Models and Computer Simulations, 2018, Vol. 10, No. 2, pp. 135–144. © Pleiades Publishing, Ltd., 2018.
Original Russian Text © S.A. Soukov, A.V. Gorobets, P.B. Bogdanov, 2017, published in Matematicheskoe Modelirovanie, 2017, Vol. 29, No. 8, pp. 3–16.

Portable Solution for Modeling Compressible Flows on All Existing Hybrid Supercomputers

S. A. Soukov[a], A. V. Gorobets[a, *], and P. B. Bogdanov[b]

[a] *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Moscow, Russia*

[b] *Institute of System Research, Russian Academy of Sciences, Moscow, Russia*

**e-mail: andrey.gorobets@gmail.com*

Received September 12, 2016

Abstract—A variant of a numerical algorithm for simulating viscous gasdynamic flows on unstructured hybrid grids and its software implementation for heterogeneous computations is described. The system of Navier–Stokes equations is approximated by the finite-volume method of an increased approximation order with the values of the variables being defined at the mass centers of the grid elements. The distributed software implementation of the numerical algorithm is adapted to running on hybrid computer systems of various architectures. Comparative implementations were created using the MPI, OpenMP, CUDA, and OpenCL software models permitting the use of multicore processors and various types of accelerators, including NVIDIA and AMD graphics processors, and Intel Xeon Phi multicore coprocessors. The data exchange between MPI processes and between processors and accelerators is carried out simultaneously with the execution of calculations (both in MPI + OpenMP mode and when using CUDA or OpenCL). The indicators of parallel efficiency and performance on systems with different types of computing devices are studied in detail. In the tests, up to 260 GPUs were successfully used.

Keywords: computational gas dynamics, heterogeneous computations, turbulent flows, MPI, OpenMP, CUDA, OpenCL

DOI: 10.1134/S2070048218020138

1. INTRODUCTION

The modern development stage of supercomputer technology has the following features that significantly affect the choice of the computer fluid dynamics (CFD) algorithm and the methods for its software implementation. The desired exaflop scale of performance is commonly achieved by using a hybrid (or, in other words, heterogeneous) architecture based on massively parallel accelerators that work as mathematical coprocessors together with central processors. This architecture is characterized by multilevel parallelism combining different types of parallel computing models. At the upper level, the nodes of the supercomputer are integrated within the parallel model with distributed memory and multiple instruction multiple data streams (MIMD) parallelism. At the hybrid node level, a model with shared memory and MIMD parallelism is employed for operation on multicore processors and/or fundamentally multicore Intel Xeon Phi accelerators. At the lower level, there is single instruction multiple data streams (SIMD) parallelism based either on vector extensions of the CPU cores or the GPU multiprocessors, respectively.

The key point in choosing an algorithm is the compatibility of all its basic operations with the most limited form of parallelism—stream processing. If an algorithm is fully compatible with SIMD and stream processing at the lower level, parallelization on the upper levels is not a major issue. Therefore, the main focus of software development shifts from the most effective implementation of the qualitatively best results of a numerical algorithm to the search for an algorithm that best suits the requirements of the hybrid architecture and simultaneously provides the results of acceptable accuracy.

Finite-volume and finite-difference CFD algorithms and software complexes for heterogeneous computations using accelerators were naturally developed on a simple-to-complex basis. In the early works, the simplest explicit algorithms on structured grids, for example [1, 2], were adapted to calculations on a single GPU. Then in later works (e.g., [3, 4]), multilevel parallelization appears for algorithms on structured grids enabling the use of multiple GPUs. A noteable domestic work for this class of algorithms on structured grids is, for instance, one of the earliest multi-GPU implementations [5] and also one of the

first successful domestic applications of the GPU to practical calculations of problems of the aircraft industry [6] (carried out several years before the publication of its results).

It should be noted that in most cases, as in all the above-mentioned works, the most frequently used development tool for accelerators is CUDA employed for the NVIDIA GPU. This is an evident choice for the early works when, in fact, nothing else had been available. At present, CUDA appears to be at a disadvantage compared to the open standard OpenCL, because CUDA is not portable. At the same time, implementations of CUDA and OpenCL are very similar and their performance and effectiveness are practically identical.

An example of early works applying OpenCL for multiGPU computations on structured grids is [7], which uses a rather complex algorithm of increased accuracy. OpenCL has also successfully been used in methods on unstructured grids. Among such works for single accelerators, there are, for instance, [8] where a numerical scheme with edge-based reconstruction is used and [9] where the universal OpenCL implementation for a scheme based on polynomial reconstruction is proposed and tested on different types of computing devices.

MPI + OpenCL implementation for multiple accelerators is presented, for instance, in [10], where the exchanges are effectively hidden behind computations for the case of the simplest first-order scheme on unstructured grids.

Among the few implementations of computations on Intel Xeon Phi accelerators, we should mention [11], which is the largest domestic application of such systems (up to 256 accelerators).

Our work is devoted to choosing and implementing the program of a numerical method, which is suited to the modern hybrid supercomputer architectures, i.e., easily implemented, fully compatible with stream processing and all existing types of accelerators, would have fundamentally unlimited scalability and would be capable of effectively using thousands of processors and accelerators.

This paper proposes a variant of a numerical algorithm of an increased accuracy order for modeling viscous gasdynamic flows on unstructured hybrid grids and its software implementation for heterogeneous computations.

The distributed implementation based on MPI is adapted to running on hybrid cluster systems of different architectures. The data exchange between MPI processes and between processors and accelerators is carried out simultaneously with the calculations. The performance when using up to 260 GPUs is demonstrated.

Implementations on OpenMP and CUDA were created in order to compare efficiency, performance, time, and effort, in addition to the main OpenCL implementation suitable for all types of devices. The possibility of direct comparison between all types of computing devices available in practice in the world at the time of publication (CPUs, NVIDIA GPUs, AMD GPUs, and Intel Xeon Phi), but between different development tools as well, is also an important advantage of this work.

2. MATHEMATICAL MODEL AND NUMERICAL METHOD

2.1. Mathematical Model

The nonstationary flows of a viscous compressible gas are described by a system of Navier–Stokes equations. In a Cartesian coordinate system, in a compact vector form, the system of equations is written as

$$\frac{\partial Q}{\partial t} + \nabla \times F = 0, \tag{1}$$

where Q is a vector of conservative variables and F is the flux vector, which is the sum of the convective F^{Φ} and diffusion F^{D} transfer.

2.2. Discretization of a System of Differential Equations

In choosing the method for the discretization of the system of equations (1), the main condition was the possibility of applying the method for calculation on condensing grids consisting of elements of an arbitrary type. From this viewpoint, the control volume method is the most effective. The computational domain is filled with a grid consisting of polyhedral cells C_i, which have the volume $|C_i|$ and the boundary surface ∂C_i. The following balance ratio is written for the grid cells:

$$\frac{d}{dt}\int_{C_i} Q d\Omega + \oint_{\partial C_i} F dS = 0. \tag{2}$$

The discrete values of the grid functions Q_i are determined at the mass centers of the control volumes (x_i, y_i, z_i) and are equal to the average integral value from the continuously distributed quantity

$$Q_i = \frac{1}{|C_i|}\int_{C_i} Q d\Omega.$$

The surface ∂C_i of the inner cell consists of plane faces ∂C_{ij}, which it shares with the cells from the set I_i. The flux through the face ∂C_{ij} is calculated by the quadrature formula as the product of the flux F_{ij} at the mass center of the face (x_{ij}, y_{ij}, z_{ij}) by the area S_{ij}. In this case, the discrete analog of Eq. (2) assumes the form

$$\frac{dQ_i}{dt} + \sum_{j\in I_i} F_{ij} S_{ij} = 0.$$

In order to approximate the convective flux, we use a set of basic schemes [12] for solving the Riemann problem on the decay of an arbitrary discontinuity $F_{ij}^{\Phi} = \Phi(Q_{Lij}, Q_{Rij}, \mathbf{n}_{ij})$.

The choice of the scheme depends on the characteristics of the simulated flow. As parameters of the function Φ, we define the direction $\mathbf{n}_{ij}$ of the unit vector of the outer normal to the face and the initial values of gasdynamic values Q_{Lij} and Q_{Rij} on both sides of its mass center. On substituting the averaged values of gasdynamic variables in the incident faces of the cell $Q_{Lij} = Q_i$ and $Q_{Rij} = Q_j$, the scheme only has the first order of approximation. The approximation order is increased by the method of polynomial reconstruction. The piecewise linear distribution of the function f over the volume C_i is described by a polynomial f_i^P of the form

$$f_i^P(x, y, z) = a_{0i}^f (x - x_i) + a_{1i}^f (y - y_i) + a_{2i}^f (z - z_i) + f_i.$$

Further, in the expressions for convective flux, we substitute the reconstructed values of the variables $Q_{Lij} = Q_i^P(x_{ij}, y_{ij}, z_{ij})$ and $Q_{Rij} = Q_j^P(x_{ij}, y_{ij}, z_{ij})$.

The components of the gradient vector of a linear function Δf_i^P are equal to the polynomial coefficients. The values of the coefficients are determined from the discrete analog of the formula of the integral gradient representation

$$\nabla f_i^P = (a_{0i}^f, a_{1i}^f, a_{2i}^f) = \sum_{j\in I_i} f_{ij} \mathbf{n}_{ij} S_{ij}.$$

The value of the function at the mass center of the face is calculated as the weighted average of the quantities f_i and f_j: $f_{ij} = g_{ij}^i f_i + g_{ij}^j f_j$, where the weighting factors g_{ij}^i and g_{ij}^j are inversely proportional to the distances from the mass centers of the cells to the face plane and their sum is equal to unity.

The diffusion flux F_{ij}^D is calculated as a half-sum of the diffusion fluxes from the averaged values of the grid functions and the gradients of reconstruction polynomials $F_{ij}^D = D(Q_i, \nabla Q_i^P, Q_j, \nabla Q_j^P, \mathbf{n}_{ij})$.

Time derivatives are integrated using the explicit Euler scheme of the first order. In the general form, the determination of the values in the inner grid cell on a new time layer $\overline{Q_i}$ with the step Δt can be described by the formula

$$\overline{Q_i} = Q_i - \frac{\Delta t}{|C_i|}\sum_{j\in I_i}\left[\Phi\left(Q_i^P(x_{ij}, y_{ij}, z_{ij}), Q_j^P(x_{ij}, y_{ij}, z_{ij}), \mathbf{n}_{ij}\right) + D(Q_i, \nabla Q_i^P, Q_j, \nabla Q_j^P, \mathbf{n}_{ij})\right] S_{ij}.$$

The algorithm for processing cells, the faces of which belong to the boundary of the computational domain Γ, is similar to the algorithm for processing the inner cells. The values of the functions $f_{i\Gamma}$ and the fluxes $F_{i\Gamma}$ are calculated or specified in accordance with the statement of the boundary conditions.

The presented algorithm is applicable to modeling subsonic flows on grids with polyhedral elements of an arbitrary type, including hybrid ones.

3. PARALLEL ALGORITHM AND SOFTWARE IMPLEMENTATION

3.1. Algorithm of Time Integration Step

The software implementation of the computational kernels includes the following three main functional blocks:

(1) calculation of coefficients for polynomial reconstruction;

(2) calculation of fluxes through faces;

(3) updating variables by summing the fluxes from the faces in order to pass to a new time layer.

A single sequential execution of the functional blocks corresponds to a time integration step. Proceeding from the equality $F_{ij} = -F_{ji}$, the fluxes through the cell faces are evaluated only in one direction and buffered into an array along the faces. The direction of the flux is taken into account as a sign on summing the fluxes through the surface of the control volume during the procedure for updating the time values.

The chosen method for organizing the calculations makes it possible to interpret each functional block as an execution of a set of identical data-independent elementary tasks on a different data set, which fits into the stream processing paradigm. At the algorithmic level, the procedure for determining the polynomial coefficients of the control volume is taken to be the unit of parallelism for the first function block; for the second block, it is the procedure for calculating the flux through the face; and for the third block, the procedure for calculating the total flux of the control volume together with updating the vector of the values of the grid functions is selected.

3.2. OpenMP Parallelization

In implementing the kernels for a multicore architecture, an invocation of the corresponding algorithmic units of procedural parallelism occurs in the loops with respect to the control volumes (the first and third functional blocks) and the faces (the second functional block). Since there is no data interdependency between the loop iterations, the loops are parallelized directly using dynamic planning with an empirically selected chunk, i.e.,

```
#pragma omp parallel for schedule (dynamic, 100).
```

3.3. Stream Processing on OpenCL and CUDA

In the implementations of the first and third functional blocks for the SIMD architecture, units of low-level parallelism are singled out. In the first block, for each control volume (cell), there are 15 threads computing 15 polynomial coefficients. One 32-thread local workgroup (that is, a block of threads that are executed in the SIMD mode on one stream multiprocessor and share its local memory, while the 32 multiplicity is due to the specific GPU architecture) processes two cells, with two idle threads. Accordingly, the total number of threads is a multiple of 16 of the number of cells. Similarly, in the third block, each cell is processed by five threads, which calculate the updated values of the five grid functions specified on the cell by summing the fluxes from the cell faces. The unit of parallelism of the second block does not change in comparison with the CPU as each thread computes the flux through one common grid face.

The choice of the size for the local workgroup significantly affects the performance and depends on the type of the accelerator. In the universal implementation on OpenCL, this selection is performed automatically at the initialization stage by the brute-force method (multiple of 32) and the timing of several test runs. No autoselection is required for CUDA, as only a single accelerator type can be used.

Special attention is paid to the organization of the data structure. The following three options for presenting data on the elements of the computational domain have been implemented:

(1) the array of structures (AOS) is the set of data specified on each cell (or face); it is grouped into a structure, the data are placed in a single AOS, the size of which is equal to the number of cells;

(2) the structure of arrays (SOA), in contrast, for the elements of the set separate arrays are allocated by cells and grouped into a structure;

(3) AOS + LDS (local data storage).

For example, suppose that the data set of each thread consists of m double-precision floating-point values, and the local workgroup has the size K. From the programmer's viewpoint, the first option seems more natural but it is less efficient, especially on the GPU, because it does not ensure the coalescing of the memory access. In the AOS format, the input data are K blocks with m values each, sequentially located in the memory. Requests from the neighboring threads of a group to the value from their set enter the memory at $8m$ bytes from each other, which results in a multifold reduction in the number of values that fall into one cache line.

In the SOA option, the data are arranged as blocks of size K in m different arrays. The threads of the local workgroup querying the value from the set access the sequentially stored values in one array. Thus, in a single transaction with a slow global memory, the data are delivered for several threads at once. If the size K is a multiple of the number of values in the cache line and the start positions of the arrays are aligned in memory accordingly, the number of transactions is minimized.

The third variant of AOS + LDS combines the convenience of the first and the performance of the second option. First, the threads of the local workgroup receive the complete linear section from K blocks of m values to the local shared memory reading in chunks consecutive values from K, as in the case of SOA. When a complete set of data for all K threads is placed in the local memory (which is ensured by a local barrier), the threads perform the calculations accessing their data in the local memory. The difference from the AOS option consists in the threads during reading accessing foreign data from other threads of the group.

The performance of options 2 and 3 proved to be identical but significantly higher than option 1: for the NVIDIA Tesla 2050 GPU, the difference was 37%, and on a newer model, GTX Titan, it was 24%, which is a noticeable drop. This suggests that the newer GPU models have become more tolerant towards suboptimal access to memory.

3.4. MPI Parallelization with Distributed Memory

The issue of load balancing between parallel processes interacting within the message transfer model is addressed based on the method of geometric parallelism. The problem of partitioning a domain into MPI-processed domains is reduced to a uniform decomposition of the weighted grid graph. The graph vertices are equivalent to grid elements, and their integer weights are equal to the number of faces. The edges of the graph represent the existence of a shared face of two elements and do not have weights. The relative deviation of the sum of the vertex weights, which is maximal among the domains, from the arithmetic mean is considered as a criterion for estimating the efficiency of the partition.

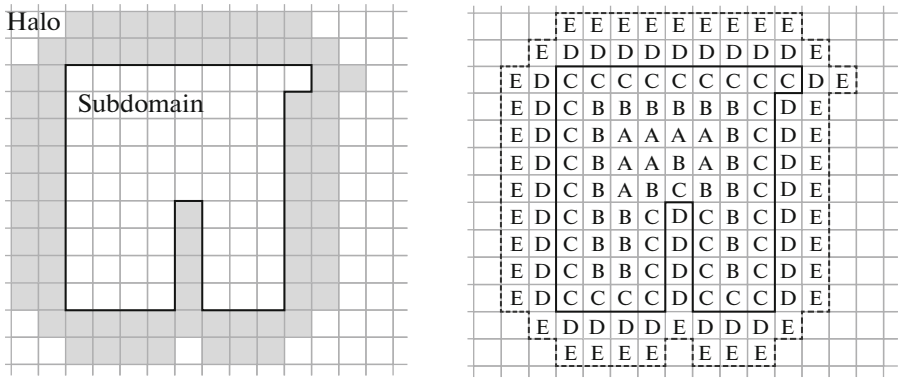
The computational domain of the MPI process is a union of a domain (subdomains of the computational domain) with a buffer zone. The buffer zone includes elements that do not belong to the domain but refer to the first two levels of connectivity of its elements along the grid graph (Fig. 1a). The configuration of the buffer zone corresponds to the computational template. The polynomial coefficients and values of the grid functions of the first-level connectivity elements are used to calculate the fluxes and gradients in the elements belonging to the domain. The grid functions of the second-level connectivity elements are taken into account in calculating the polynomial coefficients for the elements of the first level. At a single point of calculation synchronization, only the missing values of the grid functions are received/transmitted. The calculation of the polynomial coefficients for the elements entering the buffer zones is duplicated.

A special method for locally indexing the domain elements of the MPI process and the data ordering make it possible to hide the exchange of data behind the calculations. The elements are divided into five groups, as shown in Fig. 1b. Groups A, B, and C belong to the domain, groups D and E are formed from elements of the buffer zone. Group A consists of elements in the computational stencil of which there are no elements of the buffer zone. Stencils of group C elements contain buffer elements of the first-level connectivity, falling in turn in group D. The remaining elements of the domain and the buffer zone belong to groups B and E, respectively. During the MPI data exchange, the process receives the value vectors of the elements of groups D and E and passes the value vectors of the elements of groups B and C to the MPI processes requesting these values. MPI exchanges and exchanges between the CPU and the accelerator on OpenCL are implemented based on the same principle as in [10].

A block diagram of the execution algorithm of the time integration step is shown in Fig. 2. If the exchanges overlap with the calculations, each functional block is run twice on different sets of input data. The local indexing of the grid elements and the ordering of the data arrays are consistent with the grouping of the elements, as well as the buffer zone elements' belonging to the domains of MPI processes and the priority of data processing. The data on the cells of the computational domain of the MPI process are ordered as follows: A, B, C, D_1, E_1, ..., D_N, E_N, where N is the number of the neighboring domains. The data on the faces are as follows: A/A, A/B, B/B, B/C, C/C, C/D (showing the groups of cells divided by the face).

The sequential indexing of buffer elements excludes intermediate buffering during the reception of messages. Due to the fact that the value vector of the grid functions of one domain element can be requested simultaneously by several MPI processes, in the general case, it is impossible to avoid intermediate buffering of the transmitted data.

At the initialization stage of the calculation, the distributed procedures are performed for reading the data, constructing a grid graph, forming the computational domains, initializing the data exchange card, and grouping and reordering the elements. The weighted graph is decomposed using the GridSpiderPar



(a) Subdomain and buffer zone (halo) (b) Labelling groups of elements

Fig. 1. Subdomain of MPI process of computational domain.

[13] library, which can process grids with billions of elements. The operating speed of the library enables decomposition during the run of the computation without a separate preprocessing stage and without storing the decomposition in the file system.

4. STUDY OF PERFORMANCE AND PARALLEL EFFICIENCY

4.1. OpenMP Parallelization

Acceleration with OpenMP was tested on a hybrid grid consisting of 445000 cells, on various multicore processors, and an Intel Xeon Phi accelerator. The attained acceleration with respect to successive execution is shown in Table 1. The number of OpenMP threads was taken equal to both the number of cores and the hardware-supported number of parallel threads, which allowed us to estimate the gain from Hyper-Threading Technology. It should also be noted that using OpenCL on a CPU gives almost the same performance as OpenMP (according to the test on an Intel Xeon E5-2690v2 CPU with an Intel OpenCL driver). Thus, it was shown that the universal OpenCL implementation could effectively use not only accelerators but also multicore CPUs.

4.2. Performance on a Single Device

The performance of the universal software implementation was investigated on a wide range of computing devices (Fig. 3). For the tests, the same small grid of 445000 cells was used (so that the task could fit into the memory of all the considered devices). The resulting execution time of one time step varies from 90 ms on a 6-core Intel Xeon processor X5670 (Gulftown) up to 5 ms on the AMD Radeon R9 Nano (Fiji) GPU. The computational density of the algorithm is approximately two floating-point operations per byte of data from memory, the computational cost is approximately 2500 floating point operations per

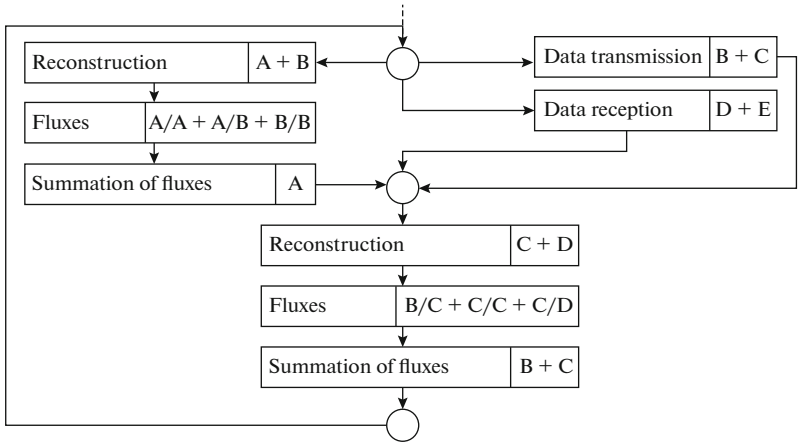


Fig. 2. Execution algorithm of time integration step.

Table 1. Acceleration with OpenMP on grid of 445000 cells

Computing device	Number of cores	Number of threads	Acceleration	Parallel efficiency, %
Intel Xeon X5670 (Gulftown)	6	6	5.05	84
Intel Xeon E5-2690 (Sandy Bridge)	8	8	6.2	78
		16	7.4	92
Intel Xeon E5-2697v3 (Haswell)	14	14	9.4	67
		28	11.6	83
Intel Xeon Phi SE10X Knights Corner	60	60	52	88
		240	99	165

grid cell, and the memory cost is about 1.1 KB per grid cell. It should be noted that due to the simplicity of the algorithm, the source code of the computational cores is less than 800 lines.

With regard to the results presented in Fig. 3, it should be noted that in the tests on MVS-10P, the Intel Xeon Phi SE10X accelerator still outperformed the 8-core CPU Intel Xeon E5-2690, albeit, only by 10%. This seems to be a very good result for that generation of accelerators. It is also interesting to compare the NVIDIA Kepler and AMD FirePro accelerators, which have approximately the same performance. At the same time, the performance of the new generation of the AMD GPU based on the Fiji architecture proves to be much higher. This is despite the fact that a low-budget game model with a small number of double-precision computing units was used, which is by several factors inferior to the peak performance of Kepler and FirePro. The gain is achieved due to the significantly higher bandwidth of the 3D-stack DRAM of the HBM memory. This again underlines the fact that the speed of RAM operation is of key importance and not the computational performance of the device.

4.3. Parallel Efficiency on Multiple Devices

The efficiency of the data transfer between the host and the accelerator and MPI exchanges between the processes are investigated for two types of systems: a single high-density computation module with 8 graphic processors and a large supercomputer with hundreds of computational modules, with one accelerator on each. For directly assessing the gain from hiding exchanges, measurements were made for exchanges in an asynchronous mode with the simultaneous data transfer and calculations (overlap mode) and for blocking exchanges executed sequentially with computations.

The results of the first test series on the node with 8 NVIDIA GTX Titan GPUs are shown in Fig. 4 for grids of various sizes. For a grid of 445000 cells, we observe both a significant gain in the overlap mode and this mode falling short of the line of ideal acceleration. For a more refined grid of 3.6 mln cells, the

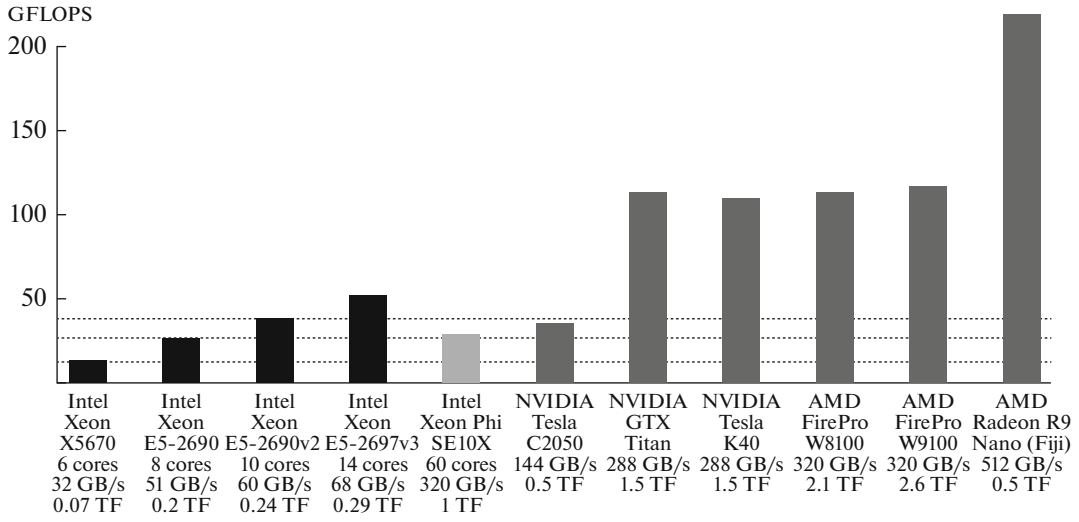


Fig. 3. Performance on various devices, grid of 445000 cells.

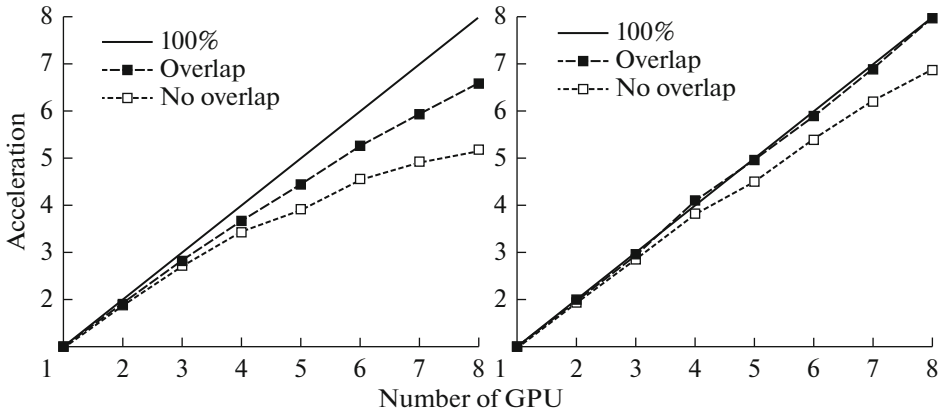


Fig. 4. Acceleration on one node (PCIe 3) with 8 NVIDIA GTX Titan GPUs, grid of 445000 cells (left) and of 3.6 mln cells (right).

overlap mode also gives a significant advantage but the obtained acceleration coincides with the ideal one eight times out of eight. This confirms that with a sufficient computing load on the GPU, the exchanges are completely hidden behind the calculations, and the bandwidth of the PCIe 3 bus is sufficient for this algorithm to fully use eight devices on one node. This implies that the data exchange capabilities of NVLink from NVIDIA available in CUDA are not critically necessary.

Figure 5 shows the results obtained for the Lomonosov-2 supercomputer using up to 260 NVIDIA K40 GPUs on a grid of 29 mln cells. The overlap mode, as in the first case, demonstrates a significant gain, and high parallel efficiency is maintained when the load is reduced to about 100000 cells per GPU. Figure 5 also presents the resulting acceleration on the CPU for MPI + OpenMP parallelization until loading approximately 10000 cells per core (2744 cores).

5. FURTHER WAYS TO IMPROVE EFFECTIVENESS

The presented algorithm and its heterogeneous implementation have a one-level decomposition of the computational domain with respect to MPI processes (Fig. 6a) with a single computing device per process. This is a simple approach readily enabling the use of resources on the petaflop level of performance—tens of thousands of processor cores and hundreds of accelerators. The loads between processors and accelerators are balanced statically at the stage of grid decomposition based on the actual performance ratio.

Attaining the exaflop level may require a more complex approach to organizing the data exchanges. Further improvement in parallel efficiency can be achieved by two-level decomposition, i.e., at the first level, the calculations are distributed between the supercomputer nodes, and at the second level, the subareas of the first level are further distributed between the processes that are performed on one node (Fig. 6b). This significantly reduces the external exchange of data between the nodes of the supercomputer since the data is only transmitted for one common external boundary of the first-level subdomain rather than across all the boundaries of the processes at the node. However, this approach also fails to make full use of hiding the exchanges. It is proposed to use two-level decomposition designed to ensure the simultaneous execution of three components: MPI exchanges, exchanges within the node, and computations. The implementation scheme of this approach is shown in Fig. 6c.

The second-level decomposition is performed so that the interface zone of the first level (the cells connected by a template of a numerical scheme with the cells from the other subdomains of the first level) should not be separated by the second-level decomposition, but should be located entirely in one interface subdomain of the second level (which can easily be done by simply excluding the interface from the second-level decomposition). This second-level subdomain connected with other first-level subdomains will be processed by central processors. It will apparently surround the subdomains of the accelerators that do not have connections with other subdomains of the first level and do not need MPI exchanges (Fig. 6c). Due to this, the MPI and intranode exchanges can be performed simultaneously with each other, significantly reducing the time required for data transmission. According to rough estimates, the performance of the processors with respect to the accelerators is more than sufficient for the processors to be able to process the entire minimal interface subdomain. Further, to ensure that the processors are not idle, the load can easily be balanced by simply expanding this subdomain according to the actual performance ratio.

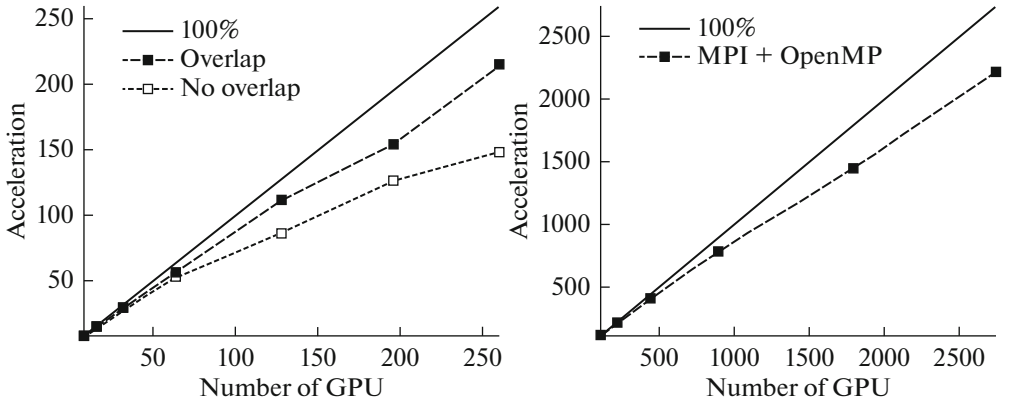


Fig. 5. Acceleration on supercomputer Lomonosov-2, GPU NVIDIA K40 (on left) and CPU Intel Xeon E5-2697v3 (on right), grid of 29 mln cells.

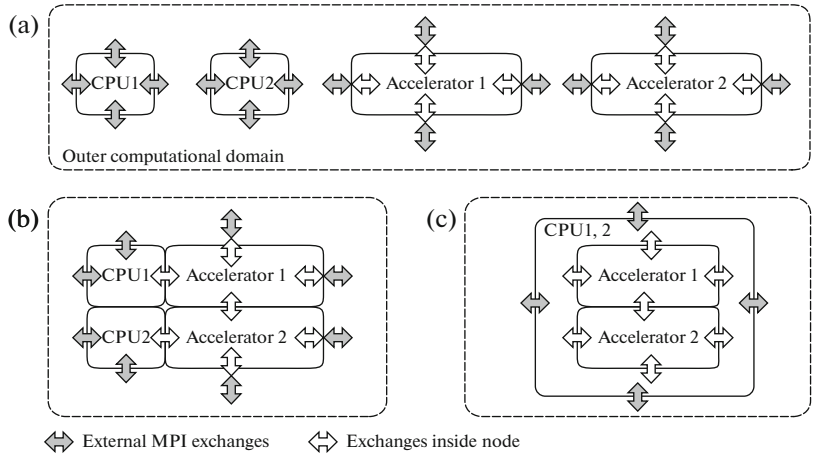


Fig. 6. Operation scheme of single-level (a), two-level (b), and improved two-level decomposition (c) with simultaneous execution of MPI exchanges, host-accelerator exchanges, and calculations (double hiding of exchanges).

It should be noted that implementing an implicit scheme will not be particularly difficult in terms of parallelization. In order to solve SLAEs with the Jacobi matrix, an implicit scheme with Newtonian linearization only requires a primitive version of the preconditioned method of biconjugate gradients consisting only of operations of a matrix-vector product, a scalar product, and a linear combination of vectors. These operations are ideally suitable for stream processing.

6. CONCLUSIONS

The paper presents a parallel algorithm for modeling compressible turbulent flows based on the finite-volume method of an increased approximation order on unstructured hybrid grids and describes its universal MPI + OpenCL implementation. Performance measurement on a wide range of computing devices, including multicore processors, GPU accelerators, and Intel Xeon Phi coprocessors, in addition to providing interesting comparative information, once again shows that memory bandwidth rather than peak performance is really of key significance. The demonstration tests were effective with up to several hundred GPUs and up to several thousand CPU cores. In addition, software implementations have been created using OpenMP and CUDA to compare various software models. Based on the results, conclusions are drawn about the almost complete identity of CUDA and OpenCL implementations in terms of both performance and labor (with the universality of the latter). Comparison of OpenMP and OpenCL implementations on multicore processors also demonstrated almost identical performance. With respect to the data exchange, a significant gain has been shown by hiding exchanges behind computations both in the distributed computations on a set of supercomputer nodes and on a single node with a set of GPUs. The

bandwidth sufficiency of the standard PCIe 3 bus is shown for running the algorithm on a node with eight GPUs. The obtained results show that the universal implementation based on OpenCL is suitable for large-scale calculations of nonstationary problems with the use of almost all hybrid supercomputer architectures currently available in the world.

ACKNOWLEDGMENTS

The work was supported by the Russian Scientific Foundation, project no. 15-11-30039 (development of discrete viscous part of flows, heterogeneous implementation of the algorithm on OpenCL), and by the Russian Foundation for Basic Research, project no. 15-07-04213 (development of decomposition tools for parallel calculations). The work used supercomputers K-100, KIAM RAS; MVS-10P, the Joint Supercomputer Center of the Russian Academy of Sciences; and Lomonosov-2, the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

REFERENCES

1. E. Elsen, P. le Gresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a gPU," J. Comput. Phys. **227**, 10148–10161 (2008).
2. P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceeding of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009.*
3. D. A. Jacobsen and I. Senocak, "Multi-level parallelism for incompressible flow computations on GPU clusters," Parallel Comput. **39**, 1–20 (2013).
4. P. Zaspel and M. Griebel, "Solving incompressible two-phase flows on multi-GPU clusters," Comput. Fluids **80**, 356–364 (2013).
5. A. A. Davydov, B. N. Chetverushkin, and E. V. Shil'nikov, "Simulating flows of incompressible and weakly compressible fluids on multicore hybrid computer systems," Comput. Math. Math. Phys. **50**, 2157–2165 (2010).
6. E. V. Koromyslov, M. V. Usanin, L. Yu. Gomzikov, A. A. Siner, and T. P. Lyubimova, "Numerical simulation of aerodynamic and noise characteristics of subsonic turbulent jets using graphic processing units," Vychisl. Mekh. Sploshn. Sred **9** (1), 84–96 (2016).
7. A. V. Gorobets, F. X. Trias, and A. Oliva, "A parallel MPI + OpenMP + OpenCL algorithm for hybrid supercomputations of incompressible flows," Comput. Fluids **88**, 764–772 (2013).
8. R. Rossi, F. Mossaiby, and S. R. Idelsohn, "A portable OpenCL-based unstructured edge-based finite element Navier-Stokes solver on graphics hardware," Comput. Fluids **81**, 134–144 (2013).
9. P. B. Bogdanov, A. V. Gorobets, and S. A. Sukov, "Adaptation and optimization of basic operations for an unstructured mesh CFD algorithm for computation on massively parallel accelerators," Comput. Math. Math. Phys. **53**, 1183–1194 (2013).
10. P. B. Bogdanov, A. A. Efremov, A. V. Gorobets, and S. A. Sukov, "Using a scheduler for efficient data exchange on hybrid supercomputers with massively-parallel accelerators," Vychisl. Metody Program. **14**, 122–134 (2013).
11. A. V. Chikitkin, V. A. Titarev, M. N. Petrov, and S. V. Utyuzhnikov, "Parallel technologies for aerodynamics problems solving in FlowModellium software complex," in *Proceedings of the Conference on Supercomputer Days in Russia, Sept. 26–27, 2016, Moscow.*
12. E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction,* 3rd ed. (Springer, Berlin, Heidelberg, 2009).
13. E. N. Golovchenko and M. V. Yakobovskii, "Parallel partitioning tool GridSpiderPar for large mesh decomposition," Vychisl. Metody Program. **16**, 507–517 (2015).

Translated by I. Pertsovskaya