New Features of DVM-System for Additional Parallelization of MPI Programs

Vladimir Bakhtin[1] [0000-0003-0343-3859], Dmitry Zakharov[1] [0000-0002-6319-5090],
Alexander Kolganov [1] [0000-0002-1384-7484], Victor Krukov[1] [0000-0001-6630-964X],
Nataliya Podderyugina[1] [0000-0002-9730-1381], Olga Savitskaya[1][0000-0002-2174-3212],
Alexander Smirnov[1][0000-0002-2971-4248]

[1] Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, 125047, Moscow, Russia
dvm@keldysh.ru

Abstract. DVM-system is designed for the development of parallel programs of scientific and technical calculations in C-DVMH and Fortran-DVMH languages. These languages use a single parallel programming model (DVMH model) and are extensions of the standard C and Fortran languages with parallelism specifications, written in the form of directives to the compiler. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters. The article presents new features of DVM-system for additional parallelization of MPI programs for clusters where the nodes can use graphics accelerators as computing devices along with universal multi-core processors.

Keywords: automation of development of parallel programs, DVM-system, accelerator, GPU, Fortran, C, MPI, OpenMP, OpenACC, DVMH.

1 Introduction

At present, the following programming models are widely used for the development of programs for high-performance calculations on modern clusters: MPI (to map the program on cluster nodes), POSIX Threads (to map the program on processor cores), CUDA and OpenCL (to map the program on accelerator cores). All these programming models are low-level. To map the program to all levels of parallelism, a programmer has to use a combination of the listed models, for example, MPI + POSIX Threads + CUDA. Technically, it is easier to combine low-level programming models implemented through libraries than to combine high-level models implemented through the languages and corresponding compilers. But it is much more difficult to program, debug, support and transfer such programs to other computers. For example, when switching from NVIDIA GPUs to AMD GPUs, CUDA will have to be replaced with OpenCL. Therefore, it is important to use high-level models and programming languages.

Among high-level programming models, a special place is occupied by the models implemented by adding to the programs in standard serial languages the specifications that control the mapping of the programs to parallel computers. These specifications in the form of comments in Fortran programs or directives to the compiler (pragmas) in C and C++ programs, are not visible to ordinary compilers, and it greatly simplifies the usage of new parallel programming models. An example of such a model for multi-core processors and SMP systems is the OpenMP model. The latest versions of the OpenMP standard [1] propose an extension of the model to use accelerators. This extension makes it possible to describe parallelism inside one node of the cluster: to use the cores of the central processor (CPU) and the cores of accelerators. A similar approach was implemented jointly by Cray, NVIDIA and AMD companies. The OpenACC standard [2] has been developed; it describes a set of compiler directives designed to simplify the creation of heterogeneous parallel programs that use both central and graphics processors. The use of high-level specifications should allow the programmer to abstract from the features of the graphics processor (GPU), data communications, etc. With the appearance of new high-level models (OpenMP version 4.+ and OpenACC), the process of parallel program developing can be significantly simplified. Thus, instead of 3 programming models, only two can be used, for example, MPI+OpenMP or MPI+OpenACC.

Recently, many compilers have appeared that support mapping OpenMP and OpenACC programs to graphics accelerators, for example, GNU GCC, Cray CCE, IBM XL, PGI, Clang/Flang and many others [3]. The main difficulties encountered by application programmers when using these compilers are the follows:

– to obtain an efficient parallel program, as a rule, a lot of experience of one or another OpenMP/OpenACC compiler usage is required. There are parallelism specifications that give the compiler some freedom when generating a parallel version of the program. For example, the kernels specification in a OpenACC defines a fragment in a program that can run on an accelerator. Such a fragment can be executed sequentially by a single thread, or it can be executed in parallel. In this case, the compiler creates a set of threads that are combined into blocks, and the number of blocks and threads is chosen by the compiler and the runtime system at its discretion. The experience of the compiler using allows to better understand the logic of its work, learn how to analyze the compiler's optimization reports, understand how to control the compiler's work and "force" it to generate more efficient code;

– not all of the features described in the standards are currently implemented in the compilers. For example, today there is no any compiler with full support for OpenMP version 5.0. Many new features of the OpenMP standard version 4.5 (published in November 2015) that are necessary for working on accelerators have not yet been implemented ("partially supported," "limited support," "offloading to GPU devices is available with some limits") [3]. When switching from one version of the compiler to another one, it is needed to transform the code to make its works correctly. Some recommendations on what to do and what not to do in the programs when using different compilers can be found, for example, in [4];

– efficient mapping of the program on accelerators requires a significant change of the program code. Moreover, such modifications depend on the target architecture for which the program is being developed. For example, the optimizations that will work well for the Intel Xeon PHI coprocessor may not work for the graphics accelerator and vice versa. Some researchers say about the portability of OpenMP/OpenACC programs on various computing devices, but to ensure the "effective" portability of parallel programs, it is necessary to support several versions of the program: for the CPU, coprocessor, and GPU;

– currently, there are no convenient, specialized tools for debugging and performance analyzing of OpenMP and OpenACC programs for accelerators. As a rule, programmers have to use standard tools, for example, to use Nvidia Visual Profiler to analyze a program performance on a graphical accelerator and understand the logic of the compiler (what computing CUDA-kernel corresponds to one or another loop in the program, what parallelism specifications led to one or another operation of copying data from central processor memory to accelerator memory, etc.). An important question is how to understand which parallelism specifications need to be corrected in the source program if an incorrect execution of the CUDA or OpenCL version of the program obtained after compilation is detected?

Thus, the development of parallel programs for clusters with accelerators in MPI+OpenMP and MPI+OpenACC models is difficult today.

The Distributed Virtual Memory for Heterogenic Systems (DVMH) model can be recommend to develop parallel programs as an alternative to OpenMP and OpenACC models. This high-level model was proposed in 2011 in Keldysh Institute of Applied Mathematics RAS. The model [5, 6] makes it possible to create programs for heterogeneous computing clusters with different accelerators. Applied programs implemented in this model (hereinafter referred to as DVMH programs) do not require changes when they are transferred from one computer system to another. The task of efficient mapping the program on all computing devices of the supercomputer is solved by DVMH compilers and the DVMH runtime system. There are implemented specialized debuggers and performance analyzers of DVMH programs, which issue error diagnostics and performance characteristics in terms of the user's program.

The article presents new features of DVM system [7], that allow to map existing MPI programs on clusters with accelerators.

The specifications of parallelism that are necessary for additional parallelization of MPI programs are considered in Section 2. The process of MPI/DVMH program launching is described and main environment variables to control the execution of a parallel program are given in Section 3. The possibilities of the instrumentation tools implemented in the DVM system for the performance analysis and functional debugging of MPI/DVMH programs are shown in Sections 4 and 5. The results of additional parallelization of Himeno program are presented in Section 6.

2 DVM specifications of parallelism for additional parallelization of existing MPI programs

Currently, when parallel computers have been used for calculations for several decades, there are many programs that were already parallelized on a cluster, but they have not parallelization on the cores of the central processor, and also do not use graphics accelerators.

Traditionally, in the DVM approach the process of programming or parallelizing of existing serial programs begins with distribution of arrays, and then parallel calculations are mapped on them. It means that to use the DVM system tools, for example, parallel MPI programs have to be converted back into serial ones and manually distributed data and calculations have to be replaced with distributed arrays and parallel loops described in DVM language.

However, firstly, the author does not always want to abandon his parallel program, and secondly, it is not always possible to transform the original data distribution and calculation scheme into DVM language. In particular, the transformation of some tasks on irregular grids into the DVMH model may require non-trivial solutions and tricks, and is not always possible.

One of the ways to get rid of both problems is to implement a new mode of the DVM system operation, when it isn't participate in inter-processor interaction, but works locally in each process.

This mode is enabled by specifying a specially created MPI library when building DVM system. This library does not perform any communications and does not conflict with real MPI implementations. As a result, an illusion of running a program on 1 processor is created for DVMH runtime system.

In addition to this mode, in the C-DVMH and Fortran-DVMH languages a notion of non-distributed parallel loop was introduced: for such loop it is not necessary to set the mapping on a distributed array. By definition, such a loop is performed by all processors of the current multiprocessor system, but since the DVM system in the described new mode considers exactly one process as a multiprocessor system, this construction does not lead to multiplication of calculations, but only allows to use parallelism within one process - to use the cores of the central processor or graphics accelerator. As a result, it is possible not to specify any distributed arrays in terms of the DVMH model and at the same time use the capabilities of the DVM system:

- to use parallelism on shared memory (use CPU cores) with using threads (OpenMP or POSIX Threads);
- to use graphics accelerators: not only "naive" porting of the parallel loop to an accelerator, but also performing of automatic data rearrangement, simplified management of data movements;
- to select optimization parameters;
- to use convenient tools to debug and analyze the performance of parallel programs.

Consider the basic parallelism specifications that can be used in the development of MPI/DVMH programs.

2.1 Computational Region

The computational region specifies a part of the program (with one entrance and one exit) for possible execution on a multi-core processor/coprocessor or graphics accelerator:

```
region-directive ::= **region** [ in-out-local-clause ]
in-out-local-clause ::= **in** ( array-range-list )
                                   | **out** ( array-range-list )
                                   | **local** ( array-range-list )
                                   | **inout** ( array-range-list )
                                   | **inlocal** ( array-range-list )
array-range ::= var-name[ subscript-range ]
subscript-range ::= [ int-expr [ : int-expr ] ]
```

In-out-local-clause specifications are intended to indicate the direction of data usage in a region: **in** - input data; **out** - output data; local - local data: the values of the specified variables are updated in the region, but these changes will not be used anywhere else; **inout** - abbreviated notation of two specifications **in** and **out** simultaneously; **inlocal** - abbreviated notation of two **in** and **local** specifications simultaneously. Not all used in region variables should be specified in in-out-local-clause. For the variables used in a region, but not specified in specification list, the following rules are applied by default:

– all used arrays are considered to be fully used (subarrays aren't selected);
– **in** attribute is assigned to any variable used only for reading;
– **inout** attribute is assigned to any variable used for writing;
– **inout** attribute is also assigned to any variable, whose direction of usage isn’t determinable;
– **local** and **out** attributes are not assigned.

In in-out-local-clause specification, a section (subscript-range) can be specified for each dimension of an array by specifying a range of indexes. The composite specifications, for example, **out**(s[1:5]), **out**(s[7:10]) or **in**(s[1:5]), **out**(s[6:10]) and the crossed specifications, for example, **out**(s[1:6]), **out**(s[3:10]) or even **out**(s[1:6]), **out**(s[3:5]) are allowed. The conflicting specifications, such as **out**(v), **local**(v), aren't allowed.

2.2 Parallel loop

A computational region usually consists of one or more parallel tightly nested loops:

```
parallel-directive ::= **parallel** parallel-map [ parallel-clause-list ]
parallel-map ::= ( int-constant )
                        | ( do-variable-list )
parallel-clause ::= private-clause
                        | reduction-clause
                        | across-clause
                        | tie-clause
```

The parallel-map specification specifies the number of loops of the nest, associated with this directive.

The **private** specification declares a private variable. A variable is called private if its use is localized within the one iteration of a loop:

```
private-clause ::= private ( var-name-list )
```

Very often programs contain loops where so called reduction operations are performed: the array elements are accumulated in some variable, or the maximum (minimum) value is calculated. The iterations of such loops may be distributed and executed in parallel, if to use the **reduction** specification:

```
reduction-clause ::= reduction ( red-spec-list )
red-spec ::= red-func ( red-variable )
                        | red-loc-func ( red-variable , loc-variable [, size ] )
red-variable ::= array-name
                        | scalar-variable-name
loc-variable ::= array-name
                        | scalar-variable-name
size ::= int-constant
red-func ::= sum
                        | product
                        | max
                        | min
                        | and
                        | or
                        | xor
red-loc-func ::= maxloc
                        | minloc
```

Consider the following loop:

```
for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++)
                A[i][j] =(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
```

Data dependence (information connection) exists between loop index *i1* and *i2* (*i1<i2*), if both these iterations refer to the same array element by write-read or read-write scheme. If iteration i1 writes the value and iteration i2 reads the value, there is a flow dependence between the iterations. If iteration i1 reads the "old" value, and iteration i2 writes the "new" value, then there is anti-dependence between these iterations. In both cases, iteration i2 can be executed only after the iteration i1. The value of i2-i1 is called a range or length of the dependence. If for any iteration i there is a dependent iteration i + d (where d is a constant), then the dependence is called a regular one or constant-length dependence. A loop with regular dependencies can be distributed using the **parallel** directive, using **across** specification:

```
across-clause ::= across ( across-spec-list )
```

```
across-spec ::= var-name [ subscript-range ]
subscript-range ::= [ flow-dep-length [ : anti-dep-length ] ]
flow-dep-length ::= int-expr
anti-dep-length ::= int-expr
```

All the arrays that have a regular data dependency are listed in **across** specification. The length of flow dependence (flow-dep-length) and the length of anti-dependence (anti-dep-length) are specified for each dimension of the array. There is no data dependence, if the length of dependence is equal to zero.

The **tie** specification is used to set a correspondence between loop dimensions and an array dimensions:

```
tie-clause ::= tie( tie-array-list )
tie-array ::= var-name [ tie-expr ]
tie-expr::= do-variable
                        | –do-variable
                        | *
```

In the DVMH model, the data distribution is performed using alignment (**align** directive) which for each element of array A brings in accordance the element or section of an array B. If an element of the array B is distributed on the processor, the element of A, corresponding to this element of B via alignment, will be also distributed on the same processor. The distribution of calculations in the DVMH model is performed taking into account the distribution of data. The parallel loop mapping rule is set by the **on** specification, which allows to associate loop dimensions with distributed array dimensions. This information allows to the compiler and runtime system to perform some optimizations that can significantly increase a parallel program performance. For example, knowing how arrays are related to each other and knowing the loop mapping rules, the DVMH runtime system can determine for a given loop optimal representation of arrays in the memory of the computing device and perform dynamic rearrangement of arrays before and after the loop execution. As a result, for example, on a graphical accelerator, all accesses to global memory performed by CUDA threads of one warp will be combined, adjacent threads of the block will access to neighboring cells of the global memory and the loop can be performed up to 10 times faster [8].

In the MPI/DVMH program, **align** and **on** specifications are not used, because the programmer himself performs the distribution of data and the distribution of calculations among the cluster nodes using MPI tools. The new optimizing specification **tie** allows to set a correspondence between the loop dimensions and the dimensions of the arrays and reorder them. Using this specification, the programmer passes the information to the runtime system - which dimension of the loop has connection with given dimension of the array (and the connection direction: direct or reverse), and if there is no connection, then "*" is used.

For parallel execution of loops with regular data dependencies on graphic accelerators, the method of hyperplanes is implemented in the DVM system. All elements lying on the same hyperplane can be calculated independently of each other. In this execution order of the loop iterations, a problem of effective access to global memory again occurs because of non-adjacent elements of arrays are processed in parallel. To efficiently perform such loops, the so-called diagonal transformation is implemented in the runtime system. As a result of such transformation neighboring elements of arrays on diagonals (in the plane of the necessary two dimensions) are placed in neighboring memory cells, that allows to apply the technique of performing the loop with dependencies on hyperplanes without significant loss of performance on the operations of an access to the global memory of the graphics accelerator. A precondition for efficient execution of the loops with a dependency in MPI/DVMH program is using of **tie** specification for all across-arrays for their dimensions with non-zero dependency lengths.

Figure 1 shows an example of a parallel loop that can be executed on a multi-core processor or accelerator.

```
#pragma dvm parallel(3) reduction (max(eps)) // For C-DVMH
for (int i = L1; i <= H1; i++)
    for (int j = L2; j <= H2; j++)
        for (int k = L3; k <= H3; k++)
...
!DVM$ PARALLEL(I,J,K) REDUCTION (MAX(EPS))   ! For Fortran-DVMH
DO I = L1,H1
   DO J = L2, H2
      DO K = L3, H3
...
```

Fig. 1. Not-distributed parallel loop.

2.3 Data actuality

Program fragments outside the computational regions are always executed on CPU. As already noted, for each region, the data necessary for its execution (input, output, local) are specified, and data movements between computing regions are performed in accordance with information about used by the region data, contained in the region description. To control data movements between regions and program fragments outside regions, special directives are provided:

getactual-directive ::= **get_actual** (array-range-list)
actual-directive ::= **actual** (array-range-list)
array-range ::= var-name [subscript-range]
subscript-range ::= [int-expr [: int-expr]]

The directive **get_actual** performs all necessary updates in order to the actual (i.e. the newest) values of data in subarrays and scalars specified in the list were in CPU memory.

The **actual** directive declares that the subarrays and scalars specified in the list have the newest values in CPU memory. The values of specified variables and the elements of arrays in memory of accelerators are considered outdated and if necessary will be updated before use.

2.4 Function called from computational region

The functions called from computational regions and parallel loops should have no side effects and contain exchanges between processors (so-called transparent functions). As a consequence of this, transparent functions should not contain I/O statements, calls of MPI library functions, and DVMH directives (for example, nested parallel loops).

It is necessary to place **routine** specification before declaration and definition of the functions called from a computational region:

routine-directive ::= **routine**

This specification tells the compiler to generate for this function a code that can be executed on a multi-core processor, coprocessor, and GPU.

3 Running MPI/DVMH programs

To compile and run MPI/DVMH programs, it is needed to use a special version of the DVM system. This version does not perform any communications and can be started in each MPI process.

Before running MPI/DVMH program, it is necessary to set the following environment variables (for example, they can be set in the dvm script):

1. **DVMH_PPN** is the number of MPI processes that will be launched on a cluster node. A positive integer or a list of non-negative integers.
2. **DVMH_NUM_THREADS** is a looped-back list of non-negative integers that specifies a number of threads in each of the MPI processes. Indexed by MPI process number.
3. **DVMH_NUM_CUDAS** is a looped-back list of non-negative integers that specifies the number of graphics accelerators in each of the MPI processes. Indexed by process number. In the current implementation, the maximum number of graphics accelerators that MPI process can use is 1.

Setting all these variables is mandatory for MPI/DVMH programs. These variables allow correctly distribute the computing resources of a cluster node among various MPI processes, bind the generated threads to the cores of the central processor, and distribute graphic accelerators among MPI processes.

For example, configuration:

```
export DVMH_PPN='3'
export DVMH_NUM_THREADS='0'
export DVMH_NUM_CUDAS='1'
```

allows to run three MPI processes on each node of the cluster, and each of them will use its own graphics card.

For a cluster node with 2 octa-core processors, the following configuration can be used:

```
export DVMH_PPN='2'
export DVMH_NUM_THREADS='8'
export DVMH_NUM_CUDAS='0'
```

As a result, 2 MPI processes will be launched on each node of the cluster, and each of them will create 8 threads and they will be bound to their own cores.

The absence or incorrect definition of any of the variables above may result in inefficient execution of the program. For example, several MPI processes will try to use the same graphics card.

The ability to use a list of non-negative integers as values for these variables allows to set its own values for each node of the cluster and for each MPI process.

For example, configuration:

```
export DVMH_PPN='2'
export DVMH_NUM_THREADS='16,0'
export DVMH_NUM_CUDAS='0,1'
```

launches 2 MPI processes on each node of the cluster, one of them will use 16 threads, and the other - a graphics card. In this case, the MPI programmer is responsible for loading balance between different computing devices.

After setting environment variables listed above, the MPI/DVMH program can be started by the command:

./dvm run < MPI-process number > < program name > [< task parameters >]

As a result of this command, the required number of MPI processes will be created, and in each of them the DVMH runtime system will distribute calculations among the cores of the central processor or graphics accelerator.

4 Analysis of MPI/DVMH program performance

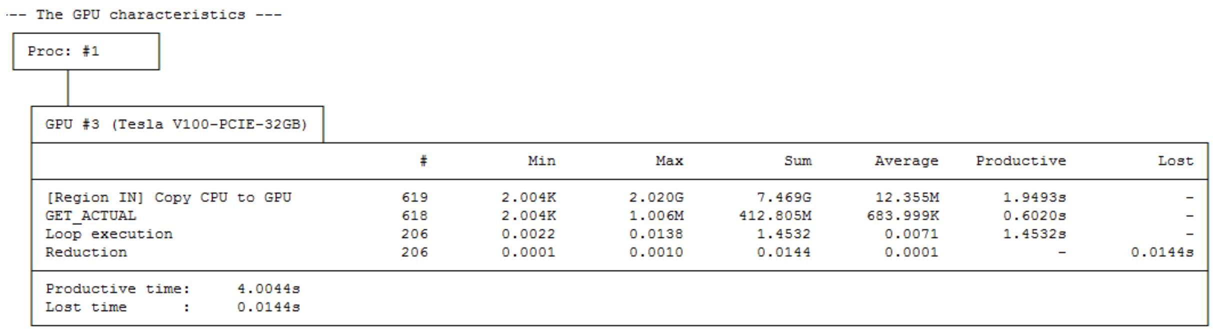
To analyze and debug the performance of DVMH programs, the tools have been created that function as follows. DVMH runtime system accumulates information with time characteristics of a program execution in RAM. When the program is finished, this information is written to a file, which is then processed by a special tool - performance visualizer.

Using the performance visualizer, the user can obtain time characteristics of his program execution with various degrees of detail (the whole program; parallel and serial loops; and also any sequences of statements marked by the programmer).

When the program is executed on accelerators, in addition to the losses due to execution of inter-processor exchanges, there are accumulated the characteristics that allow to estimate the execution times of computational regions, and the losses due to:

- data copying from CPU memory to the accelerator memory and back (when entering and exiting the computational region);
- bringing the variables in the memory of CPU and the accelerator into a consistent state (operations actual/get_actual);
- data copying for shadow, reduction, remote, across operations;
- performing of various dynamic optimizations implemented by the DVMH runtime system for more efficient use of accelerator resources (for example, rearrangement of arrays in the accelerator memory).

The results obtained by the performance visualizer for the program will be described in chapter 6 are shown in Fig. 2.



```
--- The GPU characteristics ---

Proc: #1

GPU #3 (Tesla V100-PCIE-32GB)

                                          #        Min        Max        Sum    Average   Productive      Lost
[Region IN] Copy CPU to GPU             619     2.004K     2.020G     7.469G    12.355M      1.9493s         -
GET_ACTUAL                              618     2.004K     1.006M   412.805M   683.999K      0.6020s         -
Loop execution                          206     0.0022     0.0138     1.4532     0.0071      1.4532s         -
Reduction                               206     0.0001     0.0010     0.0144     0.0001            -   0.0144s

Productive time:     4.0044s
Lost time       :    0.0144s
```

Fig. 2. The statistics of program execution on the graphics accelerator.

In the table in Fig.2 there are shown a number of performed operations, the execution times for these operations, and the amount of copied data. As already noted, such information can be accumulated for any fragment of the parallel program. For example, if the -e4 key is specified when compiling a program, such information will be obtained for each parallel loop in the program.

After completion of the MPI/DVMH program, each process keeps the information with time characteristics in its own file, and then each of them can be analyzed separately. A new dialog shell for working with statisticians is currently being developed, it should simplify the process of analyzing of DVMH program performance. Key capabilities of the developed system: simultaneous work with several statisticians, the possibility to compare characteristics for various launches, the possibility of by-interval analysis, sorting the intervals by significance, as well as graphical representation of results: charts, graphs, etc. The implementation of this system should significantly simplify the analysis of MPI/DVMH programs performance (for example, all statistics obtained for different MPI processes can be displayed/analyzed together).

5 Debugging MPI/DVMH programs

The DVMH model has an important advantage over other high-level programming models for accelerators. In the DVMH model, the movement of data between the CPU memory and the accelerator memory is not explicitly specified, but is performed automatically in accordance with the specifications of data usage in computational regions. For example, the **copyin**(A) specification in the OpenACC model means that for further execution of the program it is necessary to copy array A from the CPU memory to the accelerator memory; a similar specification **in**(A) in the DVMH model means that if the next program fragment will be executed on the accelerator and the array A is not located in the accelerator memory, it is necessary to copy array A to the accelerator memory. If array A is already located on the accelerator, or the next fragment of the program will be executed on the CPU, then no copy operations occur. This approach has the following advantages:

– allows to decide dynamically where it is more profitable to perform a particular region;
– allows to repeatedly execute the region to find the optimal mapping of calculations on the graphics accelerator;
– allows to compare the results of the region execution on the CPU and on the GPU to detect discrepancies in the results of the execution.

This made it possible to implement a special mode of operation of the DVMH program, when all calculations in the regions are simultaneously performed on the CPU and on the GPU.

The comparison of data at the entrance in the region allows to detect the absence of **out** specification for previously executed regions or the **actual** directive.

The comparison of the output data obtained in the region during execution on GPU with the output data obtained in the same region during execution on CPU allows to detect and localize errors that occur during execution on accelerators.

All output data of the computational region are included in the comparison. The integer data are compared for coincidence, and real numbers are compared with given accuracy by absolute and relative error. If the discrepancies are found, the user is informed about them. Further the version of data obtained on CPU is used in the program.

Thus, the following technique can be used to debug MPI/DVMH programs:

– in the first step, the program is debugged as an MPI program using TotalView, Intel Trace Analyzer and Collector, or other debugging tools for MPI programs. This is possible because of DVMH directives are not visible to ordinary compilers;
– in the second step, the program is started in a special mode when intermediate results of parallel execution of the program on the central processor and the graphics accelerator are compared in order to detect discrepancies in the execution results.

The following types of errors are detected:

1. Incorrect parallelization not suitable for array-parallel execution in shared memory was performed by a programmer.

2. The programmer incorrectly specified private or reduction variables in a parallel loop.
3. Arithmetical operations or mathematical functions were executed on the accelerator with the result different from the result, obtained on CPU. It can occur due to the command system distinctions leading to different results (within the limits of accuracy of the rounding).
4. The programmer specified incorrect directives of data actualization **get_actual** and **actual**, and as a result, the data processed on the central processor and accelerator became different.

Enabling and using this comparative debugging mode does not require from a programmer to make any changes in his program, tool it, or re-compile. To enable the mode of comparative debugging, it is required to set the environment variable DVMH_COMPARE_DEBUG to 1, or use the command **./dvm cmph** to launch the program.

If errors were detected the information about them is issued in standard error output stream or in a file. The name of the file can be specified in environment variable DVMH_LOGFILE. The information about error detecting and a set of indexes of the array elements with discrepancies in compact form are issued.

An accuracy of variable comparison can be changed, if to set values of the environment variables
DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS,
DVMH_COMPARE_LONGDOUBLES_EPS.

6 Testing the approach

Consider the process of additional parallelization of programs in the MPI/DVMH model using, as example, Himeno program, which solves the Poisson task in a three-dimensional area using the iterative Jacobi method [9]. There are many different versions of this program: on Fortran 77, Fortran 90, C; MPI, OpenMP; with static, dynamic arrays. For this experience, we used MPI version of the program in Fortran 90. The source code of the program is 814 lines.

Additional parallelization of this program required:

1. Declare 2 parallel loops in Jacobi procedure:

```
!DVM$ PARALLEL (K,J,I), PRIVATE(S0,SS), REDUCTION(SUM(WGOSA))
!DVM$ PARALLEL (K,J,I)
```

2. Join these 2 parallel loops into one computational region:

```
!DVM$ REGION
!DVM$ END REGION
```

3. Bring to a consistent state in CPU and accelerator memory the value of the reduction variable WGOSA, which is used to control the convergence of the iterative method:

```
!DVM$ ACTUAL(wgosa)
!DVM$ GET_ACTUAL(wgosa)
```

These specifications are used after zeroing the wgosa variable on the CPU at each iteration and before executing the mpi_allreduce function.

4. Copy the necessary elements of array P (shadow edges) before sending them and after they are received from neighboring processors:

```
!DVM$ GET_ACTUAL(P(:,:,2),P(:,:,kmax-1))
!DVM$ ACTUAL(P(:,:,1),P(:,:,kmax))
!DVM$ GET_ACTUAL(P(:,2,:),P(:,jmax-1,:))
!DVM$ ACTUAL(P(:,1,:),P(:,jmax,:))
!DVM$ GET_ACTUAL(P(2,:,:),P(imax-1,:,:))
!DVM$ ACTUAL(P(1,:,:),P(imax,:,:))
```

For this program, it is possible to specify: the dimensions of arrays that should be distributed and on how many parts (DDM pattern parameter is "1 1 2"). Depending on the distribution of the arrays, 3 different procedures passing the necessary data are called. In each of these procedures, one ACTUAL directive and one GET_ACTUAL directive were added. If to omit this parameter, for example, always to distribute arrays along 1-st dimension, then only 2 directives instead of 6 will be enough.

Thus, to parallelize this test in the MPI/DVMH model, it was necessary to add 12 DVMH directives in the program. The most difficult thing was to determine the data that needs to be copied from the accelerator memory to the CPU memory and back to perform inter-processor exchanges. Table 1 shows the execution times of 100 iterations of the program for the arrays 1025x1025x525 size on the computing cluster K60 (Keldysh Institute of Applied Mathematics, RAS) using a different number of cores and graphics cards.

Table 1. The execution times of Himeno program on hybrid cluster K60

Configuration	1 core	2 cores	4 cores	8 cores	16 cores	1 GPU	2 GPU	4 GPU	8 GPU
Time, in sec.	297.73	279.13	146.46	78.49	43.31	5.94	6.43	3.44	1.97

When using 8 nVidia Volta GV100GL GPUs, the program has accelerated almost 22 times in comparison with its execution on 16 cores of the Intel Xeon Gold 6142 CPU. When using 1 graphics accelerator, the program is accelerated almost 50 times than on one core of the CPU.

Conclusion

The advent of hybrid clusters with accelerators has seriously complicated the process of parallel program development. In addition to data distribution, distribution of calculations, and the performing of inter-processor exchanges between cluster nodes, additional parallelization is now required. It is necessary to additionally distribute data and calculations among computing devices of the cluster node (graphics accelerators, coprocessors, multi-core processors, etc.) and organize their parallel processing within a particular computing device.

The paper presented new features of the DVM system that can be used for such additional parallelization of existing MPI programs, showed the following advantages of the DVMH model in comparison with the OpenMP and OpenACC models:

1. High performance of the obtained programs, achieved due to various optimizations, which are performed both during a compilation of DVMH programs and during their execution. For example, dynamically rearrangement of data, dynamic compilation of CUDA handler code during the program runtime, and others. Some of these optimizations are made possible due to additional information in the DVMH program: about dependencies, about the correspondence of loop dimensions to array dimensions.
2. The possibility to parallelize the loops with data dependency on graphics accelerators.
3. The availability of performance analysis tools, which work in the terms understandable to a user, accumulates the characteristics of parallel program performance that are associated with DVMH language constructions.
4. The availability of tools for automated debugging of parallel programs.
5. The simplicity of DVMH parallelism specifications. All main DVMH constructions to parallelize MPI programs were presented in this article. For comparison, the description of the latest version of the OpenACC standard takes about 150 pages, and the full description of the OpenMP standard takes more than 600 pages.

Thus, the DVMH model can be safely recommended for additional parallelization of MPI programs as an alternative to OpenMP and OpenACC models.

References

1. OpenMP Application Programming Interface. Version 5.0. November, 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, last accessed 2020/11/25.
2. The OpenACC Application Programming Interface. Version 3.0. November, 2019. https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf, last accessed 2020/11/25.
3. OpenMP Compilers & Tools. https://www.openmp.org/resources/openmp-compilers-tools/, last accessed 2020/11/25.
4. Rahulkumar, Gayatri, Charlene, Yang: Optimizing Large Reductions in BerkeleyGW on GPUs Using OpenMP and OpenACC. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9626-optimizing-large-reductions-in-berkeleygw-with-cuda-openacc-openmp-and-kokkos.pdf, last accessed 2020/11/25.
5. C-DVMH language, C-DVMH compiler, compilation, execution and debugging of DVMH programs. http://dvm-system.org/static_data/docs/CDVMH-reference-en.pdf, last accessed 2020/11/25.
6. Fortran DVMH language, Fortran DVMH compiler, compilation, execution and debugging of DVMH programs. http://dvm-system.org/static_data/docs/FDVMH-user-guide-en.pdf, last accessed 2020/11/25.
7. System for automating the development of parallel programs (DVM-system). URL: http://dvm-system.org, last accessed 2020/11/25.
8. Bakhtin, V.A., Kolganov, A.S., Krukov, V.A., Podderugina, N.V., Pritula, M.N.: Methods of dynamic tuning of DVMH programs on clusters with accelerators. Russian Supercom-

puting Days: Proccedings of International conference (28–29 september 2015, Moscow), Moscow: Moscow University Press, 2015, P. 257–268.

9. Himeno benchmark. http://accc.riken.jp/en/supercom/documents/himenobmt/, last accessed 2020/11/25.