New Features of DVM system

Valery Aleksakhin [1] [0000-0001-8385-8894], Vladimir Bakhtin [1,2] [0000-0003-0343-3859], Olga Zhukova [1] [0000-0002-1033-6371], Dmitry Zakharov [1] [0000-0002-6319-5090], Victor Krukov [1,2] [0000-0001-6630-964X], Nataliya Podderyugina [1][0000-0002-9730-1381] and Olga Savitskaya[1][0000-0002-2174-3212]

[1] Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, 125047, Moscow, Russia
[2] Lomonosov Moscow State University, GSP-1, Leninskie Gory, 11999, Moscow, Russia
dvm@keldysh.ru

Abstract. DVM-system is designed for the development of parallel programs of scientific and technical computations 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, in the nodes of which accelerators (graphic processors or Intel Xeon Phi coprocessors) can be used as computing devices along with universal multi-core processors. The article presents new features of DVM-system that have been developed recently.

Keywords: automation of development of parallel programs, DVM system, accelerator, GPU, Fortran, C, irregular grid, unstructured grid.

1 Introduction

To achieve high accuracy of computations the researchers are forced to considerably refine a computation grid. It leads to proportional growth of computer memory consumption and increase of computation time. Use of unstructured grids instead of structured ones allows to solve this problem partially. In this case there is an opportunity to vary a grid detailing on computation area, thereby to reduce both a time for excessively exact computations on some areas, and random access memory used to store not needed detailed values. Also it allows to abstract numerical methods from calculation area geometry and to remove most calculation area limitations. However, such programs have much more complicated structure. When operating with regular grids, it wasn't necessary to store the neighbourhood relations, as well as the space coordinates, explicitly, as these properties and quantities were directly bound with multidimensional index spaces of the arrays of quantities.

DVMH [1, 2] model is based on data parallelism. This model is based on the concept of a distributed multi-dimensional array. Each processor has not only a local part of distributed array, but also so-called shadow edges - the copies of elements from local parts of neighboring processors. Main interconnection of the processors is performed via these shadow edges. A distribution of computations is performed by their mapping on the distributed arrays. Due to index offsets of used arrays are pre-known, the accesses are performed either to own local part, or to shadow edges of known width, which are defined as a continuation of local part along certain dimension of a distributed array. For example, for "cross" template with 4 neighbors, an element with (i, j) indexes is calculated using the elements with indexes (i-1, j), (i, j-1), (i+1, j), (i, j+1) and shadow edges of width 1 for both dimensions are needed.

DVMH compilers transform references to distributed multidimensional arrays to a form independent from sizes and location of local part on each processor, but initial index expressions aren't changed. As a result each access to distributed data is performed in global (initial) indexes, but the coefficients and shifts calculated during execution are used for access to memory for each dimension. Such approach (unlike modifying of index expressions) allows to abstract from the contents of parallelizable loops, but it introduces serious restriction on a form of a distributed array part addressed by each processor. This part is called extended local part and is a union of the local part and shadow edges. Only block distributions with shadow edges are used in DVMH model. Thus the extended local part is subarray of source array of the form (A1:B1, A2:B2, A3:B3, ..., An:Bn).

The new capabilities of DVM system [3] to solve the problem of these restrictions are presented in the article. The second section describes the expansion of DVM languages in order to introduce new types of distributed arrays, parallel loops, and other auxiliary constructions that significantly simplify parallelization of applications with irregular grids on a cluster. The third section introduces tools that allow a programmer to manually distribute data using MPI or other parallel programming technologies, while leaving the ability to use DVM languages within a cluster node to map computations on CPU or graphics accelerator cores.

2 New Capabilities of Operation with Irregular Grids

For operation with irregular grids the new type of an array and template distribution – by-element distribution – is introduced. This type of distribution doesn't superimpose any restrictions on which array elements should be located on the same processor or which array elements should be located on adjacent processors. On the contrary, it allows to specify arbitrary belonging of each array element independently.

Two new rules of by-element distribution have been added: indirect and derived. The indirect distribution is specified by an array of integers, and its size is equal to the size of the indirectly distributed dimension, and the values specify the domain number. The number of domains can be both higher and lower than the number of processors. The DVM system ensures that all elements of the same domain belong to the same processor.

The derived distribution is specified by the rule, form of which is similar to the form of DVMH model's alignment rule (ALIGN). However, it has considerably more flexibility. The syntax can be described as it is shown in Fig. 1.

```
        indirect-rule ::= indirect ( var-name )
        derived-rule ::= derived (derived-elem-list with derived-templ)
        derived-elem ::= int-range-expr
        int-range-expr ::= arbitrary integer expression + ranges are al-
lowed in index expressions, use of align-dummy variables.
        derived-templ ::= var-name [ derived-templ-axis-spec ]...
        derived-templ-axis-spec ::= [ ] | [ @ align-dummy [ + shadow-
name ]... ] | [ int-expr ]
```

Fig. 1. BNF formula for new distribution rules.

All references to distributed arrays in *int-range-expr* must be available (the element belongs to extended local part) for the corresponding element of the template (search of template elements is performed in its local part and specified shadow edges). If according to derived rule the same element should be distributed on several processors at once, then DVM system selects one of them where element will actually be distributed, and adds it in shadow edge on remaining processors with "overlay" name. There shouldn't be elements not distributed on any processor. Such cases are runtime errors and cause the program abnormal termination. Calculated non-existent indexes of distributed array are ignored without error.

An overlay is introduced for possibility of consistent distribution of grid elements. For example, there are cells, edges, vertexes. In such a case there is an opportunity to build one distribution on the basis of another, and in any sequence.

As a result of such distribution an array has two types of element indexing: global (it is also the source in serial program) and local. Local indexing is continuous within one processor, i.e. there is such an order of local elements that their local indexes will completely fill some integer segment [Li, Hi].

By-element shadow edges are also introduced. The shadow edge is a set of elements, not belonging to the current process (the requirement to belong to adjacent process is removed), for which, first, an access from any point of the program is possible without special specifications, and, second, special tools of operation with them are introduced: updating by *shadow_renew* specification, expansion of parallel loop by *shadow_compute* specification, etc.

Unlike traditional ones, by-element shadow edges are added to templates during the program execution and have names to refer to them. They are specified in almost the same way as derived distribution, see Fig. 2.

```
        shadow-add ::= shadow_add ( templ-name [ shadow-axis ]... =
shadow-name ) [ include_to ( var-name-list ) ]
        shadow-axis ::= [ ] | [ derived-elem-list with derived-templ ]
```

Fig. 2. BNF formula for specification of by-element shadow edges.

Exactly one of shadow-axis should be non-empty brackets. All arrays from the list specified in *include_to* should be aligned with a template to which dimension the shadow edge is added. As a result of such directive execution the shadow edge is added to the template and included in specified distributed arrays. After this operation shadow elements of the arrays are available for reading from the program, and also can be renewed by *shadow_renew* directive.

To implement by-element shadow edges and derived distribution the compiler generates a special function according to specified expressions. The parameters to bypass local part of the template are passed to the function by runtime system. This function, bypassing the template, fills the buffer of element indexes according to expressions in left part of mapping rule, and then returns it back to runtime system. Then the buffer is analyzed by runtime system.

For experimental use of these possibilities an auxiliary directive of localization of index array values was introduced. It modifies the values of integer array, replacing global indexes of specified target array by local ones (Fig. 3).

```
       localize-spec ::= localize ( ref-var-name => target-var-name [
axis-specifier ].)
       axis-specifier ::= [ ] | [ : ]
```

Fig. 3. BNF formula for directive of localization of index array values.

After such operation it is possible to use available method of compilation of parallel loops: they will be executed wholly in local indexes.

Together with modification of the directive of shadow exchanges and implementation of exchanges for by-element shadow edges (that now are performed not only between adjacent processors, but with arbitrary subset of processors) this set of extensions allows to parallelize and launch the applications on irregular grids on a cluster with accelerators.

To illustrate the possibilities of the DVM system for work with unstructured grids, consider a small example of Fortran program that implements a three-dimensional Jacobi algorithm (Fig. 4). In this program one-dimensional arrays are used instead of three-dimensional arrays. Because of this, indirect addressing appears, the tools for working with it were not previously available in DVM.

```
             program JAC_INDIRECT
             parameter (L=100, itmax=5000)
             real*8:: tmp,eps, maxeps=0.005
             integer x_t,y_t,z_t,cur
             real*8, allocatable :: A(:),B(:)
             integer, allocatable :: ibstart(:), ibend(:), ib(:)
             integer, allocatable :: indir_x(:), indir_y(:),indir_z(:)
             allocate(A(L*L*L),B(L*L*L), ibstart(L*L*L), ibend(L*L*L))
             allocate(indir_x(L*L*L), indir_y(L*L*L), indir_z(L*L*L))
! Here a one-dimensional array that "emulates" a three-dimensional array
! in the usual three-dimensional Jacobi algorithm is created
             cur = 1
             do i = 1,L*L*L
               x_t = (i-1) / (L*L)
               y_t = mod((i-1) / L, L)
               z_t = mod(i-1, L)
               indir_x(i) = x_t
               indir_y(i) = y_t
               indir_z(i) = z_t
               ibstart(i) = cur
               if (x_t.gt.0) cur = cur + 1
               if (x_t.lt.L-1 cur = cur + 1
               if (y_t.gt.0)   cur = cur + 1
               if (y_t.lt.L-1) cur = cur + 1
```

```
                if (z_t.gt.0)   cur = cur + 1
                if (z_t.lt.L-1) cur = cur + 1
                ibend(i) = cur - 1
              enddo
              allocate(ib(cur-1))
              cur = 1
              do i = 1,L*L*L
                x_t = (i-1) / (L*L)
                y_t = mod((i-1) / L, L)
                z_t = mod(i-1, L)
                if (x_t.gt.0) then
                    ib(cur) = i - (L*L)
                    cur = cur + 1
                endif
                if (x_t.lt.L-1) then
                    ib(cur) = i+(L*L)
                    cur = cur + 1
                endif
                if (y_t.gt.0) then
                    ib(cur) = i-L
                    cur = cur + 1
                endif
                if (y_t.lt.L-1) then
                    ib(cur) = i+L
                    cur = cur + 1
                endif
                if (z_t.gt.0) then
                    ib(cur) = i-1
                    cur = cur + 1
                endif
                if (z_t.lt.L-1) then
                    ib(cur) = i+1
                    cur = cur + 1
                endif
              enddo
! A similar CSR (Compressed Sparse Row) format is used to pack the
! array. Each element can have up to 6 adjacent elements on the left
! and on the right for each of the three dimensions. For i-th element
! of array A the list of its neighbors are contained in the array ib,
! beginning from index ibstart(i)and ending by index ibend(i).
! Above this CSR-like structure is created.
! Also the arrays indir_x/y/z which contains indexes that an element had
! in three-dimensional array are filled.
! The arrays are filled before iterative loop. Since all elements now
! are stacked in the one-dimensional array it is required to check the
! three-dimensional indexes to exclude processing of boundary elements.
              do i = 1, L*L*L
                A(i) = 0
                if (indir_x(i) == 0 .or. indir_x(i) == L-1 .or.
             &      indir_y(i) == 0 .or. indir_y(i) == L-1 .or.
             &      indir_z(i) == 0 .or. indir_z(i) == L-1) then
                    B(i) = 0
                else
                    B(i) = 4 + indir_x(i) + indir_y(i) + indir_z(i)
                endif
              enddo
! After filling the modified Jacobi algorithm is applied
              do it = 1, itmax
                eps = 0
                do i = 1,L*L*L
```

```
                 if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
           &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
           &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
                     tmp  = ABS(B(i) - A(i))
                     eps = MAX(tmp, eps)
                     A(i) = B(i)
                 endif
              enddo
              do i = 1, L*L*L
                 if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
           &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
           &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
! Indirect addressing
                     B(i) = (A(ib(ibstart(i)))   + A(ib(ibstart(i)+1))
           &             + A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3))
           &             + A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5)))
           &               / 6.0
                 endif
              enddo
              print 200,  it, eps
      200     format(' it = ', i4, '   eps = ', e14.7)
              if ( eps .lt. maxeps )    exit
            enddo
            deallocate(ibstart,ibend)
            deallocate(ib)
            deallocate(A,B,indir_x,indir_y,indir_z)
            end program
```

Fig. 4. Serial version of the program that implements the Jacobi algorithm.

Let 's start to consider a parallel version of the program:

```
      program JAC_INDIRECT
      parameter (L=100, itmax=5000)
      real*8:: tmp,eps, maxeps=0.005
      integer x_t,y_t,z_t,cur
      real*8, allocatable :: A(:),B(:)
      integer, allocatable :: ibstart(:), ibend(:), ib(:)
      integer, allocatable :: indir_x(:), indir_y(:),indir_z(:)
      integer MAP(L*L*L)
!DVM$   TEMPLATE E(L*L*L)
!DVM$   TEMPLATE :: E2(:)
!DVM$   DISTRIBUTE :: E
!DVM$   DISTRIBUTE :: E2
!DVM$   ALIGN :: A,B
!DVM$   ALIGN :: indir_x, indir_y,indir_z, ibstart, ibend
!DVM$   ALIGN :: ib
      call fillMap(MAP,L,1)
      allocate(A(L*L*L),B(L*L*L), ibstart(L*L*L), ibend(L*L*L))
      allocate(indir_x(L*L*L), indir_y(L*L*L), indir_z(L*L*L))
!DVM$ REDISTRIBUTE E(INDIRECT(MAP))
!DVM$ REALIGN (I)  WITH  E(I) :: A,B,indir_x, indir_y,indir_z
!DVM$ REALIGN (I)  WITH  E(I) :: ibstart, ibend
```

The first update is to add the array *MAP*. This array will serve as the "distribution map" on the basis of which we will distribute the data. Two templates are also declared: the static template *E*, which will be distributed by-element, and the dynamic template *E2*, which will be discussed later. The distribute directive without parameters is specified for these templates. It means that the templates will be distributed later. Also an align directive without parameters is specified for all arrays. It means that these arrays will be aligned further with some template or already distributed array. The map filling function, fillMap, is then added. One of the possible implementations of this function looks as follows:

```
      subroutine fillMap(MAP,L,axis)
      integer  numproc
      integer i,L,axis
      integer MAP(L*L*L)
!This line is needed for program compatibility with usual compilers
      PROCESSORS_SIZE(axis) = 1
      numproc = PROCESSORS_SIZE(axis)
        do i = 1,L*L*L
           MAP(i) = ((i-1) * numproc) / (L*L*L)
        enddo
      end subroutine
```

PROCESSORS_SIZE (axis) is a utility function that returns the number of processors in specified axis of the processor grid on which the program was launched. Since this program is one-dimensional, axis is equal to 1, and below everything will be described taking into account that the launch grid is one-dimensional. The concrete implementation simulates block distribution - the map is divided into equal blocks, and all elements from the first block are located on the processor with index 0, all elements from the second block are located on the processor with index 1, and so on.

After filling the distribution map, it is immediately used in the redistribute directive. Here indirect - by-element distribution is specified as distribution type. In the case of by-element distribution, i-th element of the template appears on the processor whose index is specified in the map in the i-th position. It allows to distribute the data in any format: the block distribution can be used, as here, the elements can be allocated alternately when each next element is distributed on another processor, or they can be distributed randomly. The programmer has the ability to specify any mapping.

After that, all the needed arrays are aligned to the newly created template by the realign directive. After execution of this directive the elements with index i for all specified in it arrays will be distributed on the same processor as the i-th element of template *E*. Using templates to specify the initial by-element distribution is necessary in this case, the array cannot be directly distributed by-element.

The next update in the program appears after memory allocation for the array *ib*:

```
     allocate(ib(cur-1))
!DVM$ TEMPLATE_CREATE(E2(cur-1))
!DVM$ REDISTRIBUTE E2(DERIVED((ibstart(i):ibend(i)) with E(@i)))
!DVM$ REALIGN (I)  WITH  E2(I) :: ib
```

Here another new type of data distribution appears - derived distribution. The derived distribution is a variant of by-element distribution whose idea is that it is just "derived" from any other distribution. It is worth remembering how indirect addressing looked in serial program:

```
      B(i) = (A(ib(ibstart(i)))   + A(ib(ibstart(i)+1)) +
         A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3)) +
         A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5))) / 6.0
```

We can notice that on the same processor together with *B(i)* which is already distributed by-element, we should have the elements of the array *ib* with indexes from *ibstart(i)* to *ibstart(i)+5*, that is, all elements-neighbors. Taking into consideration the data storage format, the end index will actually be *ibend(i)*, which for all non-boundary elements is just equal to *ibstart(i)+5*. We can to ensure the presence of all necessary elements via derived distribution. For the distribution of the array *ib*, the *E2* template will be used. It is created dynamically, since at the start of the program we do not know the size of the array *ib*, and therefore the size of the template. Immediately thereafter, a redistribute directive with derived type of distribution is applied to the template. This directive means that in the new template *E2* the elements with indexes beginning with *ibstart (i)* and ending with *ibend(i)* must be on the same processor as the i-th element of the template *E*. Instead of specifying the *ibstart(i):ibend(i)* range, a comma-separated list of indexes (or even a single index) can be specified in the directive. The array *ib* is then aligned to the newly created template, thereby ensuring that the element *B(i)* and all its neighbors will be located on the same processor. For all non-boundary elements of the array *B* this means that all elements from *ib(ibstart(i))* to *ib(ibstart(i)+5)* will be located on the same processor together with element *B(i)*. It should be noted that if several processors want to get the same element, when creating a derived template, this element will be placed on one of the processors, and for all other processors it is placed in automatically created shadow edges.

The next update appears after the array *ib* was filled:

```
       if (z_t.lt.L-1) then
            ib(cur) = i+1
            cur = cur + 1
        endif
      enddo
!DVM$ LOCALIZE(ibstart => ib(:))
!DVM$ LOCALIZE(ibend => ib(:))
!DVM$ SHADOW_ADD(E((ib(ibstart(i):ibend(i))) with E(@i)) = "nei1") in-
clude_to A
!DVM$ LOCALIZE(ib => A(:))
```

The localize directive is a utility directive that transforms global indexes to local ones, that is necessary for correct addressing of the arrays. The directive must be applied to all arrays that are used to index by-element distributed arrays. The array to be localized is specified on the left side of the directive and the array to be indexed by the localized array is specified on the right side. For arrays with 2 or more dimensions, it is necessary also to specify the dimension to be localized. The directive must be used after the localized array has been fully filled and will not be more modified, but before its usage to index a distributed array in a parallel loop or in a *shadow_add* directive. In this case the *ibstart* and *ibend* arrays have already been filled, and will be used for indexing immediately in the *shadow_add* directive.

Let's remember again how indirect indexing looked in the main loop:

```
        B(i) = (A(ib(ibstart(i))) + A(ib(ibstart(i)+1)) +
         A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3)) +
         A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5))) / 6.0
```

We took care of the array *ib*, but we still have the array *A*, which is indexed by the array *ib*. In order to ensure that the needed elements of array *A* are placed on the processor where element *B(i)* is located, it is necessary to add a shadow edge to array *A*, using the *shadow_add* directive. This instance of the directive specifies, that on the one processor, together with the i-th element of template *E* (part "*with E (@ i)*"), we must add to the shadow edge all elements of template *E* (the first occurrence of *E* in the directive) whose indexes are in the array *ib* from *ibstart (i)* to *ibend (i)*. Then this shadow edge is called "nei1", and it is specified that this shadow edge should be added for the array *A*. Thus, we have created a shadow edge, that for each element *A(i)* contains all its neighbors. The shadow_add directive ensures that there are no duplicate elements in the shadow edge. If an element already presents on the processor, it will not be added to the shadow edge. It should be noted that the array *ib* is localized after the *shadow_add* directive. Since it is used to index the array *A* - it is localized on it.

After that it remains only to specify the parallel directives and regions:

```
!DVM$ REGION
!DVM$ PARALLEL (i) ON B(i)
      do i = 1, L*L*L
        A(i) = 0
        if (indir_x(i) == 0 .or. indir_x(i) == L-1 .or.
     &      indir_y(i) == 0 .or. indir_y(i) == L-1 .or.
     &      indir_z(i) == 0 .or. indir_z(i) == L-1) then

            B(i) = 0
        else
            B(i) = 4 + indir_x(i) + indir_y(i) + indir_z(i)
        endif
      enddo
!DVM$ END REGION
      do it = 1, itmax
!DVM$ REGION
        eps = 0
!DVM$   PARALLEL (i) ON B(i), REDUCTION(MAX(eps)), PRIVATE(tmp)
        do i = 1,L*L*L
           if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
     &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
     &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
```

```
                tmp  = ABS(B(i) - A(i))
                eps = MAX(tmp, eps)
                A(i) = B(i)
            endif
        enddo
!DVM$   PARALLEL (i) ON B(i), SHADOW_RENEW(A)
        do i = 1, L*L*L
            if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
     &          indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
     &          indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
                B(i) = (A(ib(ibstart(i)))   + A(ib(ibstart(i)+1))
     &                + A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3))
     &                + A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5)))
     &                / 6.0
            endif
        enddo
!DVM$ END REGION
!DVM$ GET_ACTUAL(eps)
        print 200,  it, eps
200     format(' it = ', i4, '   eps = ', e14.7)
        if ( eps .lt. maxeps )    exit
      enddo
```

In this case the parallel directive distributes the loop iterations by-element based on the array *B* distribution. The i-th loop iteration is executed on the processor where *B(i)* element is located, and therefore on the processor whose index was written to *MAP(i)* at the time when the distribution directive for the template *E* was executed.

In this case a *shadow_renew* clause for *A* will update all shadow edges that are bound to the array *A*. In this example there is only one such shadow edge - *nei1*, that was declared by *shadow_add*. Other directives/clauses do not differ from standard DVM without extension. The clause *reduction(max(eps))* ensures that on each processor we will have the maximum value of *eps* as the result of execution of all loop iterations, not just the iterations, executed of this processor. The clause *private (tmp)* declaired that the variable *tmp* is private, so its value on one iteration does not affect other iterations. The region and end region directives show code areas to be executed on the graphics accelerator if such one is assigned to the program, and the *get_actual (eps)* directive specifies that the actual value of the *eps* variable is on the graphics accelerator and it must be copied to CPU memory.

The resulting program can be executed on a heterogeneous computational cluster with accelerators.

3 New Possibilities for Additional Parallelization of Existing Programs

Now, when parallel computers are exploited for more than one decade for calculation performing, there are many programs which have already been parallelized on a cluster, but don't have parallel versions for CPU cores and also don't use GPU.

Traditionally in DVM approach programming process (or parallelization of available serial programs) begins with distribution of arrays, and then parallel computations are mapped on them. It means that to use DVM system tools, it is necessary to convert the programs, parallelized, for example, using MPI, back in serial ones and to replace manually distributed data and computations by distributed arrays and parallel loops described in DVM language.

However, firstly, an author doesn't always want to discard his parallel program, and secondly, it isn't always possible to realize the source data and computation distribution schemes in DVM language. In particular, the transformation of the tasks on irregular grids to DVMH model may require non-trivial decisions and tricks and is not always possible.

One of the ways to solve both problems is a new operating mode of DVM system: DVM system doesn't participate in inter-processor interaction, but works locally on each process.

This mode is turned on by specifying a specially created MPI library when DVM system is built. The library doesn't perform any communications and doesn't conflict with real MPI implementations. As a result an illusion of a program running on 1 processor is created for DVMH runtime system.

In addition to such mode, a notion of non-distributed parallel loop is introduced in Fortran-DVMH and C-DVMH languages. For such loop it isn't needed to specify mapping on a distributed array. For example, the three-dimensional parallel loop may look like this (Fig. 5):

```
!DVM$ PARALLEL(I,J,K) REDUCTION (MAX(EPS))
!For Fortran-DVMH
DO I = L1,H1
   DO J = L2, H2
      DO K = L3, H3
...
#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++)
...
```

Fig. 5. Non-distributed parallel loop.

By definition such loop is executed by all processors of current multiprocessor system, but since in described new mode DVM system thinks that the multiprocessor system consist of only one process, such construction doesn't lead to replication of computations but only allows to use parallelism within one process – to use cores of

CPU or GPU. As a result, it is possible to avoid specifying any distributed array in terms of DVMH model and at the same time to use following DVM system capabilities:

- use parallelism on shared memory (use CPU cores): with OpenMP use or without, a possibility to bind threads;
- use GPU: not only "naive" porting of a parallel loop on the accelerator, but also execution of automatic reorganization of data, simplified management of data movements;
- select optimization parameters;
- use convenient tools of performance debugging.

This mode can be used in particular to obtain the intermediate results in a process of full parallelization of a program in DVMH model. It allows to create programs for multi-core CPU and GPU faster and noticeably easier (there is a set of restrictions for work with distributed arrays, but it is optional to create them in such approach),

It allows quickly and noticeably easier (there is a set of restrictions for work with distributed arrays, but it is optional to create them in such approach) to obtain the program for multi-core CPU and GPU, and also to evaluate the perspectives of target program speedup on a cluster with multi-core CPUs and accelerators.

Conclusions and Outlook

DVM system automates a process of parallel program development.

Obtained DVMH programs without any change can be efficiently executed on clusters of different architectures that use multi-core universal processors, graphics accelerators, and Intel Xeon Phi coprocessors. This is achieved through various optimizations that are performed both statically, when compiling DVMH programs, and dynamically.

The article introduced new possibilities of the DVM system, which allow to expand the scope of its applicability and allow to parallelize not only tasks on structured grids, for which the DVM system was designed initially [4], but also the tasks on unstructured grids.

Recently, adaptive grids have been actively used for numerical solution of mathematical physics problems. It is a method that allows to locally rebuild the grid. The adaptation is required to refine the grid elements in the areas where they are most needed, and to leave the grid less detailed elsewhere. Such grids with maximum precision allow to represent shock waves, phase transitions and other areas of large gradients of functions. The authors of the project are working to expand the capabilities of the DVM system to support adaptive grids.

This work was partially supported by Presidium RAS, program I.26 "Fundamentals of creating algorithms and software for advanced ultra-high performance computing".

References

1. 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 2019/11/21..
2. 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 2019/11/21.
3. System for automating the development of parallel programs (DVM-system), http://dvm-system.org, last accessed 2019/11/21.
4. Bakhtin, V.A., Zaharov, D.A., Kolganov, A.S., Krukov, V.A., Podderyugina, N.V., Pritula, M.N.: Development of Parallel Applications Using DVM-system. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering, vol. 8, no. 1, pp. 89-106. (2019), https://doi.org/10.14529/cmse190106.