Fortran DVMH language.
Fortran DVMH compiler.
Compilation, execution and debugging of DVMH programs.

Table of contents

1 Introduction

In recent years many computing clusters with accelerators in their nodes have been appeared. Most of them are graphics processors of NVIDIA company. The clusters with accelerators of other architecture – Xeon Phi by Intel Corporation – was appeared in 2012.

The model of parallel programming for clusters with accelerators was developed at Keldysh Institute of Applied Mathematics, Russian Academy of Sciences in 2011. The model is extension of DVM model and is called DVMH (DVM for Heterogeneous systems).

The Fortran-DVMH (FDVMH) language developed within the model simplifies a process of parallel program writing, and also allows to convert the program for cluster (DVM program) to the program for cluster with accelerators (DVMH program) with small changes.

The FDVMH language is FORTRAN 95 language extended by parallelism specifications. These specifications are implemented as special comments called directives. The directives "are invisible" to standard compilers. It allows to have single version of the program for sequential and parallel execution.

The FDVMH language allows to write the program (DVMH program) which can be executed both as sequential and as parallel one for clusters with accelerators and without accelerators.

The tools for functional debugging and performance debugging are developed for DVMH programs.

The main work on implementation of parallel program execution model is performed by DVMH runtime system (RTS – RunTime System).

To control RunTime System functioning, parallel program debugging and statistics accumulation the parameters are used. DVMH system settings and methods to correct parameters necessary for functioning are described in the document.

2 Glossary

The terms and abbreviations used in the document are described in this section.

2.1 Description of hardware computational systems

Computational cluster – a set of interconnected computational nodes (computers). The cluster node can contain some specialized processor devices – accelerators in addition to the central processor unit (CPU).

CPU, central processor unit – several universal multi-core processors with common random access memory (RAM).

Accelerator – specialized processor device connected to CPU and oriented on high-performance execution of some programs. If the accelerator has its own memory, the central processor unit moves the program and required for it data from RAM of CPU to random access memory of the accelerator to execute the program on the accelerator.

Computational device, calculator – CPU or an accelerator.

GPU - graphic processor, one of accelerator types.

CUDA device - graphic processor of NVIDIA Corporation.

Intel Xeon Phi coprocessor – specialized processor which can be used as CPU or as accelerator.

Constraint:

The current version of DVM system allows to use as the accelerator only CUDA device, and Intel Xeon Phi coprocessor - only as CPU.

2.2 The description of virtual computational systems where DVMH program can be executed

Virtual multiprocessor system (or **array of virtual processors**) – the machine provided to user DVMH program by hardware and basic system software. For distributed computer an example of such machine is MPI machine. In this case the multiprocessor system is a group of MPI processes created when the parallel program is launched. One or several MPI processes can be mapped on the same node of hardware cluster. The number of processors

of virtual multiprocessor system and its representation as multidimensional grid is specified in command line when the program is launched.

In the FDVMH model **virtual processor became heterogeneous** – it consists of a virtual host processor (MPI process is started on it and the program fragments out of regions are executed on it), virtual multiprocessor (regions with OpenMP target architecture are executed on it), and several virtual accelerators of different architectures. In this case the virtual host processor and the virtual multiprocessor are mapped on CPU and work on common RAM, but each virtual accelerator has its own memory and is mapped on the separate hardware accelerator of the appropriate architecture.

3 DVMH model and different architectures

DVMH programming model and FDVMH language allow to develop parallel programs for clusters which nodes contain the accelerators of the NVIDIA Corporation and Intel Xeon Phi coprocessors in addition to the universal multi-core processors. The model supports the usage of all listed architectures both separately, and simultaneously within one program.

DVM [1] model was taken as a basis of DVMH model. The constructions for organization of computations on clusters with accelerators and specifications of data streams, for control of data movement between RAM of central processor and memories of accelerators were added in DVM model.

The parallelism model is based on the special form of parallelism by data: one program – multiple data streams. In the model the same program is executed on each virtual processor, but each processor executes its own subset of statements according to data distribution.

First, the programmer defines arrays (***distributed data***) and iterations of the loops, that may be distributed between processors.

The distributed arrays are specified by data mapping directives (see sections 4.1.1 and 4.1.2), and parallel loops - by directives of computation distribution (see section 4.1.3).

The other variables (distributed by default) are mapped by one copy per each processor (***replicated data***). The replicated variable must have the same value on each processor except for reduction variables (see section 4.3.5) and private variables (see section 4.2) in a parallel loop.

Data distribution defines a set of local or ***own variables*** for each processor. The set of own variables defines ***the rule of own computations***: the processor assigns values only to its own variables.

During the program execution the processor may need in values of as own as other (***remote***) variables. All remote variables must be specified in directives of remote data access (see section 4.3).

Then the programmer defines code fragments which can be executed on accelerators. These fragments are called ***computational regions*** or simply ***regions***.

Program fragments outside regions are always executed on the central processor.

For each region data necessary for its execution (input, output, local) are specified.

Data movements between the central processor and accelerators are performed in general automatically according to information about used by region data, contained in the region description. To control data movements between accelerators and the central processor special directives are provided (see section 4.5).

There are several levels of parallelism in DVMH programs:

- Distribution of data and computations on MPI processes. This level is set by directives of distribution and redistribution of data and by specifications of parallel subtasks and loops.
- Distribution of data and computation on computational devices at the entrance into the computational region.
- Parallel processing within the certain computational device. This level appears at entrance into the parallel loop inside the computational region and for parallel loops outside the region in special mode that is set by an option - Opl (see **Application 3**).

4 Parallelization of Fortran-DVMH programs

Program parallelism is described in FDVMH language using the directives. Each directive is specified as a special comment:

!DVM$ <DVMH-directive>

The formal syntax of directives and the rules of directive record in free and fixed forms are given in **Appendix 1**. All examples in this document are written in the fixed form.

Program parallelization in DVMH model can be separated into the following stages:

1. Distribution of data (arrays) and computations (parallel loops) on an array of virtual processors.
2. Determining and specification of remote data.
3. Determining of regions for execution on accelerators.
4. Control of data movement between CPU memory and memories of accelerators.

4.1 Distribution of arrays and parallel loops

At the first stage **DISTRIBUTE**, **ALIGN** and **PARALLEL** directives are used. If to consider these directives at abstract level, they set correspondence between points of index (discrete) spaces of two objects. The index spaces of the following objects are used:

P – index space of virtual processor array. A user defines the array of virtual processors and set it when launches the program.

$\mathbf{A_i}$ - index space of i-th data array.

$\mathbf{L_j}$ - index space of j-th parallel loop. The parallel loop is considered as an array which elements are loop iterations. The number of dimensions of the array is defined by quantity of the loop headers.

Directives set correspondence between points (elements) of the following objects:

DISTRIBUTE: $\mathbf{A_i \Rightarrow P}$. To each point (the virtual processor) P a subset of points (array elements) of $\mathbf{A_i}$ is matched.

ALIGN: $\mathbf{A_{i1} \Rightarrow A_{i2}}$. To each point (array element) of $\mathbf{A_{i2}}$ the subset of points (array elements) of $\mathbf{A_{i1}}$ is matched.

PARALLEL: $\mathbf{L_j \Rightarrow A_i}$. To each point (array element) of $\mathbf{A_i}$ the subset of points (loop iterations) of $\mathbf{L_j}$ is matched.

Set of these directives defines a subset of elements of arrays and iterations of parallel loops for each virtual processor.

Data and statements that aren't specified by these directives are automatically distributed on each virtual processor (replicated data and not parallelizable computations).

4.1.1 Distribution of arrays. DISTRIBUTE and REDISTRIBUTE directives.

FDVMH language supports distribution by blocks (equal and non-equal) and distribution via alignment.

Distribution of array **A** is described by the following directive

!DVM$ DISTRIBUTE A ($f_1 \ldots f_k$ **)**

where f_i - distribution format for i-th dimension:

f_i	=	**BLOCK**	-	distribution by equal blocks
	=	**WGT_BLOCK** (*WB, NBL*)	-	» distribution by blocks according to their relative "weights"
	=	*****	-	non-distributed dimension

k – number of dimensions

If ***fi* = BLOCK**, the array dimension is distributed mainly by equal blocks. If the number of the array elements (**N**) doesn't exceed number of processors (**P**), on each processor either one array element is located, or no one. If the number of array elements (**N**) is more than number of processors, the number of elements per processor is determined by a formula:

[k * N / P] - [(k - 1) * N / P],

where: **k** – processor number, **N** – a number of elements of an array, **P** – number of processors.

If ***fi* = WGT_BLOCK**(*WB, NBL*), dimension is distributed according to their relative "weights". *WB* is a one-dimensional array of real numbers of *NBL* size.

For $1 \le i \le NBL$, *WB(i)* defines weight of *i*-th block. The blocks are distributed on *P* processors with balancing of sums of block weights on each processor. The condition

$P \leq NBL$

must be satisfied.

The processor weight is defined as a sum of weights of all blocks distributed on it. The array dimension is distributed proportionally to the weights of processors.

The **BLOCK** format is a special case of the **WGT_BLOCK**(*WB, P*) format, where *WB(i)* = 1 for $1 \leq i \leq P$ and *NBL = P*.

If ***f****i* = *, the whole dimension (local dimension) is distributed on each processor.

The number of **BLOC**K and **WGT_BLOCK**(*WB, NBL*) formats defines number of dimensions of virtual processor array. If we enumerate the **BLOCK** and **WGT_BLOCK***(WB, P)* formats as $\boldsymbol{fb_1,\ldots,fb_n}$, then dimension with $\boldsymbol{fb_i}$ format is mapped on i-th dimension of virtual processors array, and the number of blocks is defined by the size of i-th dimension of virtual processor array. In this case at the program startup it is possible to set any n-dimensional array of virtual processors.

Several arrays (*A1, A2,…*) can be distributed at the same mode by the single directive of the form:

!DVM$ DISTRIBUTE ($\boldsymbol{f_1 \ldots f_k}$ **) :: A1, A2,…**

where $\boldsymbol{f_i}$ - distribution format for i-th dimension.

In this case the arrays must have the same rank, but may have different sizes of dimensions.

The array specified by **DISTRIBUTE** directive, can be redistributed by the following directive:

!DVM$ REDISTRIBUTE A ($\boldsymbol{f_{12} \ldots f_{k2}}$ **)**

REDISTRIBUTE directive can be applied only to arrays with **DYNAMIC** specification:

!DVM$ DYNAMIC A

For redistributed array initial distribution may not be specified, it is so-called postponed distribution. The **DISTRIBUTE** directive will have the form:

!DVM$ DISTRIBUTE :: A

in this case **DYNAMIC** specification is optional.

If the array with ALLOCATABLE attribute is specified in **DISTRIBUTE** directive, the distribution is postponed until execution of ALLOCATE statement that allocates the array in the memory.

REDISTRIBUTE directive for dynamically distributed array can be performed only after ALLOCATE statement execution.

Example. Distribution by blocks.

!DVM DISTRIBUTE (BLOCK):: A,B,C

real A (12), B(11), C(5)

At startup on four processors the distribution will be the following:

Processsor	**A**	**B**	**C**
R(1)	**1**	**1**	**1**
	2	**2**	
	3		
R(2)	**4**	**3**	**2**
	5	**4**	
	6	**5**	
R(3)	**7**	**6**	**3**
	8	**7**	
	9	**8**	
R(4)	**10**	**9**	**4**
	11	**10**	**5**
	12	**11**	

Example. Distribution by weights of blocks.

```
      DOUBLE PRECISION WB(12)
      REAL A(12)
!DVM$ DISTRIBUTE A ( WGT_BLOCK( WB, 12 ))
      DATA WB / 2., 2.,1., 1., 1., 1., 1., 1., 1., 2., 2. /
```

Processor	**A**
R(1)	**1**
	2
	3
	4
R(2)	**5**
	6
	7
	8
R(3)	**9**
	10
R(4)	**11**

12

Distribution by the **WGT_BLOCK** format can be performed for any number of processors in the range from 1 to *NBL*. For this example the size of processor array R can be from 1 to 12.

Example. Distribution of dynamically allocated arrays.

```
      SUBROUTINE SP(K)
      REAL, ALLOCATABLE, DIMENSION (:) :: A
!DVM$ DISTRIBUTE (BLOCK) :: A
      READ(*,*) N
!     allocation and distribution of array A
      ALLOCATE (A(N))
      . . .
      END
```

4.1.2 Data localization. ALIGN and REALIGN directives.

Alignment of array **A** with the distributed array **B** brings in accordance to each element of array **A** an element or section of array **B**. In case of distribution of array **B** the array **A** will be distributed simultaneously. If element of array **B** is distributed on the processor, the element of **A**, corresponding to element **B** via alignment, will be also distributed on the same processor.

The method of distribution via alignment performs the following two functions.

1) Identical distribution of arrays of same shape to the same processor array does not always guarantee that the correspondent elements will be located on the same processor. It forces to specify remote access (see section 4.3) where it perhaps isn't exist. Only alignment of corresponding elements of the arrays guarantees their location on the same processor.

2) Several arrays can be aligned with the same array. Redistribution of the base array by **REDISTRIBUTE** directive will cause corresponding redistribution of the array group.

The main method of data localization (i.e. remote data decreasing) is joint distribution of several arrays. Joint distribution of two arrays **A** and **B** is described by the array_alignment diirective.

!DVM$ ALIGN A ($a_1 \ldots a_n$ **) WITH B (** $b_1 \ldots b_m$ **)**
where:
a_i - parameter of i-th dimension of aligned array **A**,
b_j - parameter of j-th dimension of aligned array **B**,
n - number of dimensions of array **A**,
m - number of dimensions of array **B**.

This directive brings in accordance to each array **B** element some subset of elements of array **A**. This subset of array **A** elements will be distributed on the same processor where the corresponding element of array **B** will be located. Parameters of the aligned array **A** and the basic array **B** can have the following form:

$a_i = ID_i$ $b_j = c*ID_j+d$

$= *$ $= *$

where:

ID_i , ID_j - identifiers
c, d - integer constants

Consider the semantics of these designations.

If a_i = *, i-th dimension of array **A** is entirely distributed on each processor where at least one element of array **B** is located (replication, local dimension).

If b_j = *, execution of **ALIGN** directive doesn't depend on j-th dimension of array **B** (collapse, i.e. dimension would not exist when correspondence is established).
If $a_i = ID_i$, always exists one and only one $b_j = c*ID_j+d$, where $ID_i=ID_j$. The equality $ID_i=ID_j$ means that i-th dimension of array **A** is bringing in correspondence to j-th dimension of array **B**. The correspondence of elements is set by $c*ID_j+d$ function. The indexes of the basic array must not exceed the limits of index space of the aligned array, otherwise it is necessary to perform alignment with template (see section 4.3.1).

Examples of ALIGN directives and semantics.

!DVM$ ALIGN A(I) WITH B(2*I+1)
To distribute the **A(I)** and **B (2*I+1)** elements on the same processor.

!DVM$ ALIGN A(I,J) WITH B(J,I)
To distribute **A (I, J)** and **B (J, I)** elements on the same processor.

!DVM$ ALIGN A(I) WITH B(*, I)
To distribute **A(I)** element on those processors where at least one element of **I**-th column **B** is located.

!DVM$ ALIGN A(I, *) WITH B(I)
To distribute the I-th line of **A** and the **B(I)** element on the same processor, i.e. to replicate the second dimension of array **A**.

Several arrays (*A1, A2,…*) can be aligned with the same array **B** in the same manner by the single directive of the form:

!DVM$ ALIGN (a_1… a_n) WITH B (b_1… b_m) :: A1, A2, …
In this case arrays *A1, A2* … must have the same number of dimensions (*n*), but can have different sizes of dimensions.

The parameters of alignment and/or basic array can be changed by the directive

!DVM$ REALIGN A (a_1… a_n) WITH C (c$_1$… c_k)

The aligned array should be described as **DYNAMIC**.

Initial alignment may not be specified, this is so-called postponed alignment. The **ALIGN** directive will have the form:

!DVM$ ALIGN :: A

in this case **DYNAMIC** specification is optional.

If in **ALIGN** directive the variable with ALLOCATABLE attribute is specified as the aligned array, the directive execution is postponed until execution of ALLOCATE statement. The **REALIGN** directive can be executed only after execution of ALLOCATE statement.

Example. Alignment of dynamically allocated arrays.

```
      SUBROUTINE  SBP(N)
      REAL, ALLOCATABLE, DIMENSION (:,:) ::  X, Y
!DVM$ ALIGN  Y( I, J )  WITH  X( I, J )
!DVM$ DISTRIBUTE  X ( BLOCK , BLOCK )
!DVM$ DYNAMIC  Y
      . . .
      ALLOCATE(X(N,N))
      ALLOCATE(Y(N,N))
      . . .
!DVM$ REALIGN  Y( I, J )  WITH  X( J, I )
      . . .
      END
```

Note, that ALLOCATE statements can't be executed in reverse order.

4.1.3 Distribution of parallel loop iterations. PARALLEL directive.

The parallel loop in the DVMH model is considered as an array of the loop iterations. Number of dimensions of such array is equal to the number of the parallel loop headers. The size of each dimension is defined by parameters of the corresponding header of the loop. For such representation to be correct, the following conditions must be satisfied:

- the parallel loop headers must not be separated by other statements (tightly nested loop);
- the parameters of the parallel loop headers must not be modified during the loop execution (rectangular index space);
- the loop iteration must be indivisible object and it should be executed on one processor. Therefore left parts of assignment statements of one loop iteration should be distributed on same processor (according to the rule of own computations).

Distribution of a parallel loop iterations is performed by the following directive:

!DVM$ PARALLEL (I_1 **,…** I_n **) ON A (** e_1**,…** e_m **)**

where: $_j$ - a variable (index) of j-th header of the parallel loop,

n - number of the loop headers,

A - array identifier,

m - number of the array dimensions,

$e_i = a * I_k + b$, ***a, b*** – integer variables

I_k – a variable (index) of k-th header of the loop.

This expression means the following:

- k-th dimension (the loop header) of the array of the loop iterations is matched with i-th dimension of data array,
- correspondence between loop iteration and array element is set by the linear function of $a^* I_k+b$.

PARALLEL directive matches each iteration of the parallel loop with some array element. It means that the loop iteration will be executed on the processor where the corresponding array element is located. The semantics of **PARALLEL** directive is similar to semantics of **ALIGN** directive. The difference is that instead of aligned data array the array of parallel loop iterations is used.

Example.

```
!DVM$   PARALLEL  ( I, J )  ON  A( I, J)
        DO  I = 1, N
          DO  J = 1, M-1
            A(I,J)  =  …
            B(I,J)  =  …
          ENDDO
        ENDDO
```

For the left parts of assignment statements of one loop iteration to be distributed on one processor, it is necessary to apply to the array **B** description the following directive:

```
!DVM$ ALIGN B( I,J )   WITH A( I,J )
```

If it is impossible to locate the left parts of statements on the same processor, the loop must be split on several loops for which the conditions of loop iteration array are satisfied.

Example.

```
DO    I = 1, N
    A(2*I) = . . .
    B(3*I) = . . .
ENDDO
```

```
!DVM$  PARALLEL  ( I )  ON  A( 2*I )
       DO  I = 1, N
         A(2*I) = . . .
       ENDDO
!DVM$  PARALLEL  ( I )  ON  B( 3*I )
       DO   I = 1, N
         B(3*I) = . . .
       ENDDO
```

The loop is split on 2 loops, each of them satisfies to condition of parallel loop.

The parallel loop must additionally satisfy to following conditions:

- distributed dimensions of arrays are indexed only by regular expressions of the form *a*I + b*, where I - is the loop index;
- left part of the assignment statement is a reference to the distributed array, reduction variable (see section 4.3.5) or the variable described in the loop body;
- there are no DVMH-directives inside the loop body.

4.2 Private variables

A variable is called private if its usage is localized within one iteration of a loop.

The distributed arrays can't be private variables. Value of the private variable isn't defined at the beginning of the loop iteration and isn't used after the loop iteration, therefore own copy of the private variable can be used in each iteration of the loop.

If private variables are used in a parallel loop, the additional **PRIVATE** specification in **PARALLEL** directive is needed:

```
!DVM$   PARALLEL  ( I, J )  ON  A( I, J ) , PRIVATE ( X )
        DO  I = 1, N
        DO  J = 1, N
            X = B(I,J) + C(I,J)
            A(I,J) = X
        ENDDO
        ENDDO
```

If several private variables are used in the loop, they should be listed after **PRIVATE** keyword in parentheses through a comma.

4.3 Remote data. Their types and specifications.

Data calculated on one processor but used on the others, are called ***remote***.

Detection of remote data is performed by means of assignment statement analysis. The assignment statement is always executed on the processor where data of its left side are located. If data in left and right sides of the assignment statement are located on the same processor, there are no remote data for this statement. Otherwise it is necessary to define the form and the size of remote data and to describe them by appropriate directives (see below). Such analysis will be called data localization analysis.

The purpose of parallelization is maximum parallelism with remote data minimizing (maximum of localization).

In the following sections we will use the program fragment

```
!DVM$   DISTRIBUTE  A ( BLOCK)
        . . .
!DVM$   PARALLEL  ( I )  ON  A ( I )
        DO  I =1,N
          A(I) = expr
        ENDDO
```

where *expr* – expresion.

Modifying expression *expr*, consider the main methods of data localization and remote data specifications for one-dimensional arrays (one dimension of a multidimensional array).

4.3.1 Data localization. Alignment with template.

Let

A(I) = B(I) + C(I)

If A(I), B (I) and C (I) are distributed on the same processor for every I, there are no remote data for this statement. Data localization can be performed using **ALIGN** directives:

```
!DVM$ ALIGN B( I,J )  WITH  A( I,J )
!DVM$ ALIGN C( I,J )  WITH  A( I,J )
```

Consider the statement

A(I) = B(I + d1) + C(I - d2)

where d1, d2 – positive constants.

It is impossible fully localize data for this statement, using array A, because of +d1 and -d2 offsets violate limits of index space of array A. Therefore it is necessary to perform alignment with a template in following manner:

```
!DVM$ TEMPLATE  TABC ( N+d1+d2 )
!DVM$ ALIGN B( I )  WITH  TABC( I )
!DVM$ ALIGN A( I )  WITH  TABC( I + d2 )
!DVM$ ALIGN C( I )  WITH  TABC( I + d1+d2)
!DVM$ DISTRIBUTE TABC ( BLOCK )
```

In this case A(I), B(I+d1) and C(I-d2) will be distributed on the same processor for every I. The template TABC defines some index space which is an intermediary between data array and the array of virtual processors. The template elements have no real location in the memory. They specify processors the corresponding elements of data arrays should be distributed on.

4.3.2 Remote data of SHADOW type

Let

A(I) = B(I - d1) + B(I + d2)

In this case full localization of data is impossible. Nevertheless it is necessary to perform the partial localization of the data using the directive

```
!DVM$ ALIGN B( I )  WITH  A( I )
```

After this directive execution the remote data location is defined precisely. To compute all A(I) on one processor d1 elements of array B from the processors with smaller element indexes and d2 elements of array B from the processors with bigger indexes will be used. Such data will be called remote data of **SHADOW** type (***shadow edges***).

To specify the size of shadow edges **SHADOW** directive is used:

```
!DVM$ SHADOW  B( d1:d2 )
```

For several arrays (*A1, A2,…*) with the same sizes of shadow edges following single directive can be used

```
!DVM$ SHADOW ( d1:d2 )  ::  A1, A2, …
```

The size of shadow edges is equal 1 by default.

In each parallel loop where array B remote data of **SHADOW** type are used, additional specification in **PARALLEL** directive is needed:

!DVM$ PARALLEL (I) ON A (I), SHADOW_RENEW (B)

4.3.3 Remote data of ACROSS type

Let

A(I) = A(I - d1) + A(I + d2)

As in the previous section, it is necessary to describe the size of remote data by the directive:

!DVM$ SHADOW A(d1:d2)

But in **PARALLEL** directive **ACROSS** specification is added:

!DVM$ PARALLEL (I) ON A (I), ACROSS (A(d1:d2))

Difference between **ACROSS** type data and **SHADOW** type data is in the following: independent execution of loop iterations is impossible, since before calculating A(I) it is necessary to calculate A(I-d1). In **ACROSS** specification all distributed arrays with regular data dependency are listed.

4.3.4 Remote data of REMOTE type

Let

A(I) = C(5) + C(I + N)

where C - distributed array.

In this case the following **REMOTE_ACCESS** specification in **PARALLEL** directive is needed:

!DVM$ PARALLEL (I) ON A (I), REMOTE_ACCESS (C(5), C(I+N))

If there are the statements

A(I)=C(5)

or

A(I)=C(N)

outside a parallel loop, before the statements the directive

!DVM$ REMOTE_ACCESS (C(5))

or

!DVM$ REMOTE_ACCESS (C(N))

is needed correspondently.

4.3.5 Remote data of REDUCTION type

Let in a loop the statements

A(I) = B(I) + C(I)
S = S + A(I)

are calculated.

For the first statement it is necessary to localize data as in section 4.1.2. For the second statement in **PARALLEL** directive **REDUCTION** specification is needed:

!DVM$ PARALLEL (I) ON A (I) , REDUCTION (SUM(S))

where:

SUM – name of summing reduction operation,

S – reduction variable.

Reduction operations are: SUM, PRODUCT, AND, OR, MAX, MIN, EQV, NEQV, MAXLOC, MINLOC.

4.3.6 Multidimensional arrays

When localizing data by **ALIGN** directive for multidimensional arrays it is necessary to specify indexes of the aligned array elements for all dimensions.

Regarding existence of remote data it is enough to analyze only distributed dimensions of arrays. Local dimensions are fully distributed on each processor and there are no remote data on them. Each distributed dimension which has remote data, should be considered in remote data specification (see section 4.3).

4.4 Specification of regions for execution on accelerators

The computational region specifyes a part of the program (with one entrance and one exit) for possible execution on one or several computational devices.

The region is specifyed by the pair of directives marking the beginning and the end of the region and has the following form:

!DVM$ REGION [*specification-list*]

< region inner >

!DVM$ END REGION

Region inner is a fragment of the program containing arbitrary quantity of parallel loops, separated, perhaps, by sequential groups of statements. The region may be empty.

Specification-list in **REGION** directive contains, if necessary, an information about the directions of data usage in the region: input, output, local.

The specifications follow the word **REGION** and are separated by commas. Data – variables, arrays, subarrays are listed after the specification name in parentheses through a comma.

There are following specifications:

IN - input data: latest values of these data should be in the region;

OUT - output data: the values of specified variables are updated in the region and may be used further;

LOCAL - local data: the values of specified variables are updated in the region, but these modifications won't be used further;

INOUT - abbreviated notation of two specifications **IN** and **OUT**;

INLOCAL - abbreviated notation of two specifications **IN** and **LOCAL**.

The records **IN** (A, B) and **IN**(A),**IN**(B) are allowed.

Subarray sections are written through a comma, for example, **IN** (s(1:5,2:6)).

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.

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 for reading;
- **INOUT** attribute is assigned to any variable used for writing;
- **INOUT** attribute is assigned to any variable, whose direction of usage isn’t determined;
- **LOCAL** and **OUT** attributes aren't assigned.

If only **IN** attribute is specifyed for a variable (**OUT** or **LOCAL** isn’t specifyed), it means that there are no any writing in such variable in the region and it doesn't modifyed during the region execution.

Example.

```
!DVM$      REGION IN (B,C), OUT(A)
!DVM$      PARALLEL ( I, J ) ON A( I, J)
           DO  I = 1, N
           DO  J = 1, N
                A(I,J)= B(I,J) + C(I,J)
           ENDDO
           ENDDO
!DVM$      END REGION
```

DVMH arrays are distributed among calculators, undistributed data are replicated. Iterations of parallel DVMH loops inside the region are shared between calculators according to the rule of parallel loop mapping specifyed in the parallel loop directive.

Each statement of sequential group of statements is executed on all calculators, except the case of distributed data modification - then the rule of own computation works.

Constraints:

- Nested regions aren't allowed.
- There should not be input-output operations in the region if it is executed on CUDA device.
- There should not be **ALLOCATE** and **DEALLOCATE** statements in the region.
- there shouldn't be statements to exit from a region inside the region.
- Inside sequential group of statements there shouldn't be statements to exit from the group.
- The statements of conditional and unconditional jump to bypass parallel loops are forbidden in a region, all parallel loops should be executed.

4.5 Control of data movement between CPU memory and memory of accelerators

The control of data movement outside computational regions between a random access memory of CPU and memories of accelerators is specified by actualization directives - **GET_ACTUAL** and **ACTUAL**.

GET_ACTUAL directive 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. If there are no parameters in the directive all data in CPU memory become actual.

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 elements of arrays located in memory of accelerators are considered outdated and if necessary will be updated before use. If there are no parameters in the directive all data are considered actual only in CPU memory.

The parameters of directives – the list of variables, arrays and subarrays - are specified in parentheses through a comma.

It isn't recommended to use **ACTUAL** and **GET_ACTUAL** directives without parameters because of increasing of error probability (**ACTUAL**), and also a danger of unnecessary data movements (**GET_ACTUAL**).

Example.

```
!DVM$      ACTUAL (B , C)

……..

!DVM$      REGION IN (B,C), OUT(A)
!DVM$      PARALLEL  ( I, J )  ON  A( I, J)
           DO  I = 1, N
           DO  J = 1, N
                A(I,J)= B(I,J) + C(I,J)
           ENDDO
           ENDDO
!DVM$      END REGION
……..
!DVM$      GET_ACTUAL ( A )
           print *, A
```

4.6 The distributed arrays in COMMON blocks and EQUIVALENCE statements

The arrays, distributed by default, can be used in COMMON blocks and EQUIVALENCE statements without restrictions.

The arrays, distributed by **DISTRIBUTE** and **ALIGN** directives, can't be used in EQUIVALENCE statements. Moreover, these arrays can't be associated with other data objects. Explicitly distributed arrays may be components of COMMON block under following conditions:

- COMMON block must be described in main program unit.
- Each description of the COMMON block must have the same quantity of components, and corresponding components - sequences of memory of the same size.
- If explicitly distributed array is the component of COMMON blocks, then the array declarations in different program units must specify the same data type and the same configuration. **DISTRIBUTE** and **ALIGN** directives for the array must have identical parameters.

Example. Explicitly distributed array in COMMON block.

Declaration in the main program.

```
           PROGRAM  MAIN
!DVM$      DISTRIBUTE  B ( *, BLOCK )
           COMMON /COM1/  X, Y(12), B(12,30)
```

Declaration in subroutine. The error is another number of components.

```
           SUBROUTINE  SUB1
!DVM$      DISTRIBUTE  B1 ( *, BLOCK )
           COMMON /COM1/  X, Y(12), Z, B1(12,30)
```

Declaration in subroutine. The error is other distribution of the array.

```
           SUBROUTINE  SUB2
!DVM$      DISTRIBUTE  B2 ( BLOCK, BLOCK )
```

```
      COMMON /COM1/  X, Y(12), B2(12,30)
```

Declaration in subroutine. The error is other configuration of the array.

```
      SUBROUTINE  SUB3
!DVM$ DISTRIBUTE  B3 ( *, BLOCK )
      COMMON /COM1/  X, Y(12), B(30,12)
```

Declaration in subroutine. There is no errors.

```
      SUBROUTINE  SUB4
!DVM$ DISTRIBUTE  B4 ( *, BLOCK )
      COMMON /COM1/  X, Y(12), B(12,30)
```

4.7 Procedures in parallel program

Procedure call inside parallel loop

The procedure called inside parallel loop must not have side effects and contain exchanges between processors (*purest procedure*). As a consequence, the purest procedure doesn't contain:

- input-output statements;
- FDVMH directives;
- assignment of values to COMMON blocks variables;
- variables from module program unit.

Procedure call outside parallel loop

If the actual argument is explicitly distributed array (**DISTRIBUTE** or **ALIGN**), it should be passed without shape changing. It means, that the actual argument is the reference to the array beginning, and the corresponding formal argument has the same configuration.

The formal arguments

If the actual argument is a distributed array, then corresponding formal argument must have *explicit* or *inherited* distribution.

Explicit distribution is described by **DISTRIBUTE** and **ALIGN** directives with the following restriction: the formal argument can be aligned only with other formal argument. Explicit distribution of the formal argument means that before the procedure call a user must provide the distribution of the actual argument in exact correspondence with distribution of the formal argument.

Inherited distribution of array C (the formal argument) is described by the directive:

```
!DVM$ INHERIT  C
```

Inherited distribution means that the formal argument inherits distribution of the actual argument for each procedure call. Inherited distribution doesn't require from a user to distribute the actual argument in correspondence with the formal argument.

Local arrays

In a procedure local arrays may be distributed by **DISTRIBUTE** and **ALIGN** directives. The local array can be aligned with formal argument. **DISTRIBUTE** directive distributes a local array on the processor subsystem the procedure was called on (*current subsystem*).

For distributed local array with SAVE attribute **DISTRIBUTE** and **ALIGN** directives have identical parameters in each procedure call.

Example. Distribution of local arrays and the formal arguments.

```
      SUBROUTINE  DIST( A, B, C, N )
      DIMENSION  A(N,N), B(N,N), C(N,N), X(N,N), Y(N,N)
!     explicit distribution of the formal argument
!DVM$ DISTRIBUTE  A ( *, BLOCK )
!     aligned formal argument
!DVM$ ALIGN  B( I, J )  WITH  A( I, J )
!     inherited distribution of formal argument
!DVM$ INHERIT  C
!     aligning  local array with  formal argument
!DVM$ ALIGN  X( I, J )  WITH  C( I, J )
!      distribution of local array
!DVM$ DISTRIBUTE  Y ( *, BLOCK )
      . . .
      END
```

4.8 Input-output

The statements of Fortran77 standard are used for organization of data input/output in FDVMH program.

FDVMH allows only restricted form of input/output statements for distributed arrays:

- An input-output list must contain only one name of distributed array and can't contain other input-output items.
- Only «*» format is allowed in formatted input-output statements.
- A control information list may not contain the ERR, END and IOSTAT specifiers.
- Only replicated variables is allowed in control information list.

The statements of distributed array input/output cann't be used in a parallel loop.

Input/output statements for replicated data have the following restrictions:

- The control information list should not contain ERR and END specifiers.
- Only the following simplified form of explicit loop is allowed:
  (A (i1, i2..., I), I = n1, n2)
  when inputting replicated assumed-size array.

Input statement, INQUIRE statement, and also any other input-output statement with IOSTAT controlling parameter may not be used in a parallel loop.

FDVMH program performing unformated input/output of distributed arrays isn't in general compatible with serial FORTRAN 77 program. The data written by one program may not be read by other one, because of difference in record lengths.

5 DVMH program execution scheme

DVMH program execution can be considered as the execution of sequence of computational regions and code fragments between them, that we will call outregion space. The code in outregion space is executed on the central processor whereas computational regions can be executed on heterogeneous computational devices. Both inside and outside regions parallel loops and serial fragments of the program can exist.

DVMH program execution begins synchronously by all launched processes. For execution of sequential fragments of the program one main sequential execution thread is created for each process.

When entering the computational region each process independently performs additional distribution of the data used by this computational region among computational devices. At this stage dynamic planning to balance loading and minimize the time costs for data movements connected with the data redistribution is performed.

6 Compilation, execution and debugging of DVMH programs

6.1 What is DVMH program?

The parallel program is the ordinary serial program, in which DVMH directives specifying its parallel execution are inserted.

DVMH directives are written as special comments

!DVM$ < DVMH directive >

that in the serial program is considered as the comment.

DVMH program is one or several files with source codes in FDVMH language, having extension **fdv, f, for, f90, f95, f03**. If the files have extensions **f90, f95, f03**, it is considered that they are in the free form, otherwise using compilation option (**– FI**) it is neccesary to specify that the record form is fixed. When compiling it is necessary to specify option **-f90** if the files have extension **fdv, f, for**, but are written in free form.

If there is module program unit among the files, it is nessesary to integrate the files in the uniform program using INCLUDE.

If there is a lot of files, but there are no modules, then to combine the files it is possible to use **makefile** or to list all the files in compilation line.

6.2 DVM system tuning

To compile and launch DVMH program it is necessary to copy to the working directory where the program is, the file of dvm commands start (**dvm**) from **dvm_sys/user** directory of DVM system.

In this file environment variables that can be modifyed by the user are defined. The environment variables for DVMH programs are described in **Appendix 3**.

Instructions how tune environment variables, compile and launch programs on different platforms are available on DVM site.

6.3 Method of DVMH programs debugging

DVMH program debugging implies two different kinds of activity:

- functional debugging, its purpose is to achieve correctness of functional execution of the program;
- debugging of performance, its purpose is to achieve required level of parallel program performance.

It is recommended to debug DVMH programs on test data in the following sequence of steps:

1. Serial execution and debugging using standard Fortran compiler and standard debugging tools.
2. Functional debugging of the parallel program.
3. Debugging of the parallel program performance.

6.3.1 Serial execution and debugging using standard Fortran compiler and standard debugging tools

DVMH directives are Fortran language comments for standard compilers therefore DVMH program is processed by them as usual serial program. It allows to debug the program as usual serial program (in the mode of DVMH directive *ignoring*) using ordinary debugging tools.

6.3.2 Functional debugging of parallel program

Functional debugging of DVMH program is performed in the following sequence of steps:

1. Compilation. Obtaining the ready-to-run program.
2. Dynamic control of DVMH directives.
3. Comparison of execution results on a cluster without accelerators.
4. Comparison of execution results in region on CPU and accelerators.

6.3.2.1 Compilation. Obtaining the ready-to-run program.

The command of conversion and compilation of DVMH program has the following form:

dvm f < DVMH-program_name>

where:

dvm	–	prefix (name of DVMH commands startup file);
< DVMH-program_name >	–	name (with extension) of the file with source program code. The file is searched only in the current directory;

Processing result: executable file <**DVMH-program_name**> in the current directory. If the converter detected errors, the executed file isn't created.

6.3.2.2 Dynamic control of DVMH directives

Dynamic control allows to detect the errors of the following types:

1. Undeclared data dependency in a parallel loop.
2. Using non-initialized private variables inside or outside of a parallel loop.
3. Modification of read-only variables.
4. Use of reduction variables after asynchronous reduction start, but before its completion.
5. Undeclared access to non-local elements of distributed array.
6. Writing to shadow edges of distributed array.
7. Reading shadow elements of array before their update completion.
8. Modification of non-local element of the distributed array in sequential part of the program.
9. Violation of a distributed array bounds.
10. Writing to remote access buffer.

For dynamic control a program should be compile at first in the mode of obtaining *debug version of parallel program*.

The command to obtain debug parallel vesion has the form:

dvm fpdeb <DVMH-program_name >

Processing result: executable file < **DVMH-program_name** >**_p**.

The command to start debug parallel version of DVMH program performing dynamic control of DVMH directives has the form:

dvm err < DVMH-program_name >

Processing result: errors, detected in DVM-directives (if they exist).

If any errors were detected **error.dbg** file containing list of all found errors is created in the current directory. The short message about existence of errors appears after output of task execution results.

The structure and the list of error messages of dynamic control see in **Application 4**.

Lack of dynamic control errors doesn't guarantee the correct operation of the parallel program due to following reasons:

- dynamic control doesn't check correctness of the description of reduction operations;

- the procedures called from DVMH-program, but written in other languages and not dynamically controlled can be a source of errors;
- dynamic control doesn't check a correctness of a region execution on accelerators, and also absence of **OUT** or **LOCAL** specifications for variables modifyed in the region;
- the tested sequential program can contain errors not appeared during sequential execution, but these errors could occur during parallel execution.

Therefore program debugging should be continued.

6.3.2.3 Comparison of execution results on a cluster without accelerators

To search such errors the method of trace accumulation and comparing of sequential and parallel execution of the program is used. It allows to localize the program point and moment, when the results are beginning to differ.

The program is executed on the CPU, accelerators aren't used.

When tracing the information about all readings and modifications of variables, entering each loop iteration, entering and exiting parallel loop is accumulated.

To perform comparison it is necessary to perform the following sequence of commands.

1. The command to obtain debug parallel version:

dvm fpdeb < DVMH-program_name >

Processing result: the executable file < **DVMH-program_name**> **_p**.

2. The command to obtain debug serial version:

dvm fsdeb < DVMH-program_name >

Processing result: the executable file < **DVMH-program_name** > **_s**.

3. The command to start debug serial version of DVMH-program, accumulating reference trace on one processor:

dvm trc < DVMH-program_name**>**

Processing result: the file **0.trd**, containing accumulated trace. If trace accumulation errors were detected – an error message after output of task execution results and **error.trd** file is created in the current directory.

4. The command to start debug parallel version of DVMH programs comparing computation results of the program execution on one processor in the special mode of reduction operation checking with the accumulated earlier reference trace.

dvm red < DVMH-program_name **>**

Processing result: If the errors are detected – error message output after output of task execution results and appearance of **error.trd** file in the current directory.

5. The command to start debug parallel version of DVMH program comparing computation results of the program execution on several processors with previously accumulated reference trace.

dvm dif [**N1** [**N2** [**N3 [N4]**]]] **< DVMH-program_nam**e>

where **N1, N2, N3, N4 -** the sizes of a processor matrix (by default – 1 1 1 1).

Processing result: If the errors are detected – error message output after output of task execution results and appearance of **error.trd** file in the current directory.

If no differences are detected in trace, it is possible to execute the program in parallel with real data.

6. If differences are detected, but the error in program was not detected using reference trace and diagnostics of trace comparison, the user can accumulate trace on each processor starting debug parallel version of the program on the required processor matrix. The following command is used for this purpose.

dvm ptrc [**N1** [**N2** [**N3 [N4]**]]] **< DVMH-program_nam**e **>**

where **N1, N2, N3, N4 -** the sizes of processor matrix (by default – 1 1 1 1).

Processing result: For each processor trace is accumulated in the separate file with names: **0.trd**, **1.trd**, **2.trd**, etc. If errors are found the appropriate message is issued after output of the task execution results.

The structure of trace accumulation files and the message list of result comparison error see in **Application 4**.

Note. All steps of debugging described in sections 6.3.2.2 and 6.3.2.3 can be started by one command:

dvm ftest [**N1** [**N2** [**N3 [N4]**]]] **< DVMH-program_nam**e **>**

6.3.2.4 Comparing of execution results in regions on CPU and accelerators

Comparison of execution results in regions on CPU and accelerators is a special mode of DVMH program operation when all computations in regions are executed simultaneously on CPU and GPU.

In this mode the output data obtained in region during execution on GPU are compared with the data obtained in the region during execution on CPU with a given degree of accuracy.

Such mechanism allows to detect and localize the errors which occur during execution on accelerators.

If the mode of comparative debugging of the region is turned on the same iterations of the same parallel loop will be executed twice – once on CPU and other – on GPU.

All output data of the computing region are included in comparison. Integer data are compared on equality, and real numbers are compared with the given accuracy by absolute and relative inaccuracy. If discrepancies were found the information about it is issued. Further in the program the version of data obtained on CPU is used.

Such comparison is used to check a correctness of a region execution on accelerators, and also absence of **OUT** or **LOCAL** specifications for variables that are modifyed in the region.

When region is executed on the accelerator the errors can arise for several reasons:

1. Incorrect parallelization not suitable for array-parallel execution in shared memory was performed by programmer.
2. The programmer incorrectly specified private or reduction variables in a parallel loop.
3. Arithmetical operations or mathematical functions are executed on the accelerator with the result different from result, obtained on CPU. It can occur due to command system distinctions leading to different results (within the limits of precision of the rounding).
4. The programmer specified incorrect data actualization directives **GET_ACTUAL** and **ACTUAL** owing to what processed data on CPU and the accelerator were different.

Turning on and use of comparative debugging mode doesn't require from the programmer to make any changes in the program, and also again to compile it.

The program will be executed in the mode of comparative debugging if it is launched by the command:

dvm cmph [N1 [N2 [N3 [N4]]]] < name of executable file of DVMH-program >

where **N1, N2, N3, N4** - the sizes of processor matrix (by default – 1 1 1 1).

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 specifyed in environment variable **DVMH_LOGFILE**.

Accuracy of variable comparison can be changed, if to set values of environment variables **DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS, DVMH_COMPARE_LONGDOUBLES_EPS**.

It is recommended to perform comparative debugging on test data.

6.3.3 Compilation and execution of DVMH programs on cluster with accelerators

Compilation and execution of DVMH programs is performed using the following commands:

dvm f <compilation options> <DVMH-program_name>

dvm run [N1 [N2 [N3 [N4]]]] < name of executable file of DVMH-program >

where **N1, N2, N3, N4** - the sizes of processor matrix (by default – 1 1 1 1).

When DVMH-program is launched the rank and sizes of virtual processor matrix determine the configuration and number of processes (**N1*N2*N3*N4**), where DVMH-program will be executed in parallel.

Compilation options are described in **Appendix 3**.

Instructions how to setup environment variables, to compile, to start programs on different platforms are available on DVM site.

6.3.4 Debugging parallel program performance

To debug performance performance analyzer is used. It allows to obtain information about main characteristics of the program performance (or its parts) on parallel system. The performance of parallel program execution on multiprocessor computers with distributed memory is determined by the following main factors:

- degree of program parallelism - a part of parallel calculations in total volume of calculations;
- balance of processor loading during parallel calculations;
- time necessary for interprocessor communications;
- degree of overlapping of interprocessor communications and calculations.

6.3.4.1 Main characteristics of performance

Main characteristics and their components

- **Efficiency coefficient** (**Parallelization efficiency**) is ratio of productive time to total processor time.
- **Time of execution** (**Execution time**) - the maximum value among times of execution of the program on all used processors.
- The number of used processors (**Processors**).
- Total processor time (**Total time**) is production of the time of execution (Execution_time) by the number of used processors (Processors).
- **Productive time** (**Productive time**) is the sum of three components - productive processor time (**CPU**), input/output time (**I/O**) and productive system time (**Sys**).
- **Lost time** (**Lost_time**) - a difference between the total time of processor usage and productive time. Components of lost time are insufficient parallelism, communications and idle time.
- **Insufficient parallelism** (**Insufficient par**) and its components.
- **Communications** and all their components (**Communication**).
- **Idle** (**Idle time**) of a processor because of its insufficient loading.
- **Potential losses because of disbalance** (**Load_Imbalance**).
- **Potential synchronization losses** (**Synchronization**) in case of execution of collective operations and all their components.
- **Potential losses because of variation of times** (**Time_variation**) because of execution of collective operations and all their components.
- **Overlapping time** (**Overlap**) and its components. This characteristic reflects potential reducing of communication expenditures due to overlapping of interprocessor communications with computations.

Characteristics of program execution on each processor

- Lost time (**Lost time**) is the sum of insufficient parallelism losses (**User Insufficient par**), system insufficient parallelism losses (**Sys Insufficient par**), communications losses (**Communication**) and idle (**Idle time**).
- Insufficient parallelism losses (**User insufficient par**).
- System insufficient parallelism losses (**Sys insufficient par**).

- Idle on the given processor idle (**Idle time**) is difference between maximal time of interval execution (on any processor) and the time of interval execution on the given processor.
- Total communication time (**Communication**).
- Real time of losses because of dissynchronization (**Real synchronization**).
- Potential time of losses because of dissynchronization (**Synchronization**).
- Potential time of losses because of time variation (**Variation**).
- Time of asynchronous operation overlapping (**Overlap**).
- Losses because of load imbalance (**Load Imbalance**) is difference between maximal processor time (**CPU** + **Sys**) and the time on the given processor.
- Time of interval execution (**Execution time**).
- Productive processor time (**User CPU time**).
- Productive system time (**Sys CPU time**).
- Input/output time (**I/O time**).
- Number of processors used for interval (**Processors**).
- Communication times for all types of collective operations (**Reduction**, **Shadow**, **Remote access**, **Redistribution** and **I/O**).
- Real dissynchronization losses for all types of collective operations.
- Potential dissynchronization losses for all types of collective operations.
- Potential time variation losses for all types of collective operations.
- Time of overlapping for all collective operations (**Overlap**).

Note 1. The last three characteristics are issued only if in start parameters **IsTimeVariation=1** parameter is set;

To obtain value of real losses because of dissynchronization it is necessary to set **IsSynchrTime=1** parameter.

Note 2. The performance analyzer outputs to the user the execution characteristic both for all program, and on each processor.

6.3.4.2 Representation of program as a hierarchy of intervals. Execution with statistics accumulation.

Program execution is considered as an interval of the highest level (zero level). This interval can include several intervals on the next (first) level. Such intervals can be parallel loops, sequential loops as well as any sequence of operations marked by user for which the execution starts from the first statement and completes with the last statement. The intervals of the first level can in turn include intervals of the second level etc.

All above characteristics are computed not only for the whole program but also for each its interval. In FDVMH the interval is defined as follows:

!DVM$ INTERVAL [<integer expression>]

<sequence of statements>

!DVM$ END INTERVAL

For example, marking loop body as an interval and specify loop counter as integer expression each loop iteration will be represented as separate interval. In the same manner characteristics of even and odd loop iterations or characteristics of procedure execution with given parameters can be obtained.

To accumulate statistics about DVMH program performance when it is launched on multiprocessor computer or on workstation network the parameter **Is_DVM_STAT** (the sign of statistics accumulation in the file **usr.par**) should be equal to 1.

After the program completion with statistics accumulation the statistics file with name **sts.gz+** (or **sts**, or <**task name**>.**sts.gz**+) should be created in the current directory. If during data accumulation the errors were detected, the file can still be created and an error message will be output on the screen or to a file. The list of messages is given in **Appendix 5**.

Constraint:

- Interval may not be inside the loop, it must to include the loop entirety.

6.3.4.3 Start of performance analyzer. Representation of task characteristics for intervals.

To get time characteristics for intervals user should execute the following command:

dvm pa <statistics file name> < file name with characteristics>

All characteristics are written in text form into specified file. For each interval the following information is saved:

- name of file with source code of DVMH program and number of first statement of interval (SOURCE, LINE);
- interval type – whole program, parallel loop (PAR), sequential loop (SEQ) or marked by user sequence of statements (USER);
- interval level number (LEVEL);
- the number of entrances (and exits) in the interval (EXE_COUNT);
- value of expression defined when describing interval (EXPR);
- main execution characteristics and their components (Main characteristics);
- minimal, maximal and average program execution characteristics on every processor (Comparative characteristics);
- program execution characteristics on every processor (Execution characteristics on processors).

When characteristics are issued their components are placed in the same line (to the right in brackets), or in the next line (to the right of symbols “*” or “-“).

The components of some characteristics connected with collective operation execution are issued as columns of table where lines correspond to the type of collective operation and columns correspond to characteristics. One of columns (Nop) of this table contains a number of operations of every type, that are characteristics not depending on the number of processors used for the program execution.

Information about minimal, maximal and average values of such characteristics is saved in the table in the same way. Some characteristics aren't issued at all if their values are equal to zero.

6.3.4.4 Recommendations on characteristics analysis

The main criterion is **efficiency coefficient** of parallelization. If **efficiency coefficient** is low, it is necessary to analyze lost time and its components.

At first it is necessary to evaluate three components of lost time for main interval (as a rule, iterative loop in the program is selected as such interval). It is most probable that the main share of lost time is the share of one of first two components (insufficient parallelism or communications).

If **insufficient parallelism** is the reason, it is necessary to detect, on what sections it is found – sequential or parallel ones. In the last case the reason may be very simple – incorrect specification of processor matrix when the program is started or wrong data and computation distribution. If insufficient parallelism is found on sequential sections, the existence of the sequential loop executing large volume of computation is the reason of it most likely. The removal of this reason may require much efforts.

If the main losses are due to **communications** it is necessary, first of all, to pay attention to real losses because of **dissynchronization** (**Real synchronization**). If its value is close to amount of communication losses, it is necessary to consider **potential losses because of imbalance** (**Load_Imbalance**) as iust imbalance of parallel loop calculations is the most probable cause of dissynchronization and great communication losses. If imbalance value is much less than value of **potential losses because of synchronization** (**Synchronization**), it is necessary to pay attention to value of **potential losses because of times variation** (**Time_variation**) for collective operations. If dyssynchronization isn't a consequence of time variation of completion of collective operations, it may be caused by imbalance of some parallel loops which in the considered interval of the program execution may be compensated mutually. So it makes sense to consider imbalance characteristics in intervals of lower level.

The second probable cause of great dissynchronization losses may be processor dissynchronization that can occur due to input/output operations starts. This happens because the main job (operation system input/output function calls) is executed on input/output processor while the rest of processors are waiting for data from I/O processor or information about collective operation completion. This cause can be easily revealed if user considers the corresponding **communication** component – losses because of input/output communications.

Large number of reduction operations or operations loading data from other processors (renewing shadow edges or remote access) may be also the main cause of communication losses. In this case user should check specifications of remote data. Existence of excess specifications is one of the causes of losses because of communications.

There is another approach for characteristic analysis when first, efficiency coefficients and lost time in first level intervals are analyzed and then they are analyzed in second level intervals etc. As a result a critical fragment of the program will be found. It is

necessary to take into considerations that interval dissynchronization losses and interval idle losses may be caused by not only imbalance and time variation on this interval but by imbalance and time variation on other previously executed intervals.

Note. As upon transition from sequential execution of the program to its parallel execution on one processor efficiency losses are possible, it is recommended to compile the sequential program with calls of information accumulation functions to estimate performance and to run obtained sequential program on one processor. For this purpose it is necessary to execute following compilation command:

dvm f -s < DVMH program name>

and command to start the program:

dvm run 1 < DVMH program name >

Then it is necessary to compare the obtained statistics with statistics of parallel execution on one processor.

7 References

1. DVM system[Electronic resource] - : [web-site] – DVM
   http://dvm-system.org/

8 The example of Jacobi program in the Fortran-DVMH language

We will illustrate of Fortan DVMH capabilities on the example of Jacobi algorithm program.

```
            PROGRAM    JAC
            PARAMETER    (L=8,  ITMAX=10)
            REAL     A(L,L), EPS, MAXEPS, B(L,L)
!DVM$   DISTRIBUTE     ( BLOCK,   BLOCK)   ::   A
!DVM$   ALIGN  B(I,J)  WITH  A(I,J)
!               arrays A and B  with block distribution

          PRINT *,  '**********  TEST_JACOBI   **********'
          MAXEPS  =  0.5E – 7
!DVM$  region
!DVM$  PARALLEL    (J,I)   ON   A(I, J)
!               nest of two parallel loops, iteration (i,j) will be executed on
!               processor, which is owner of element A(i,j)
          DO   J  =  1, L
          DO   I  =  1, L
             A(I,  J)  =  0.
             IF(I.EQ.1 .OR. J.EQ.1 .OR. I.EQ.L .OR. J.EQ.L) THEN
                 B(I,  J) = 0.
             ELSE
```

```
                B(I,  J)  = ( 1. + I + J )
              ENDIF
            END DO
            END DO
!DVM$  end region
          DO  IT  =  1,  ITMAX
               EPS  =  0.
 !DVM$  actual(EPS)
 !DVM$  region
 !DVM$  PARALLEL  (J,  I)   ON  A(I,  J),  REDUCTION ( MAX( EPS ))
 !              variable EPS is used for calculation of maximum value
               DO  J  =  2, L-1
               DO  I  =  2, L-1
                    EPS = MAX ( EPS,  ABS( B( I, J)  -  A( I, J)))
                    A(I, J)  =  B(I, J)
               END DO
               END DO
 !DVM$  PARALLEL  (J,  I)   ON  B(I,  J),  SHADOW_RENEW  (A)
 !              Copying shadow elements of array A from
 !              neighbouring processors before loop execution
               DO  J = 2,  L-1
               DO  I = 2,  L-1
                    B(I, J) =  (A( I-1, J ) + A( I, J-1 ) + A( I+1, J)+
         *                     A( I, J+1 )) / 4
               END DO
               END DO
 !DVM$ end region
 !DVM$ get_actual(EPS)
               PRINT 200,  IT, EPS
 200           FORMAT(' IT = ',I4, '   EPS = ', E14.7)
               IF ( EPS . LT . MAXEPS )    EXIT
          END DO
!DVM$ get_actual(B)
          OPEN(3,FILE='JAC.DAT',FORM='FORMATTED', STATUS='UNKNOWN')
          WRITE (3,*)  B
          CLOSE (3)
          END
```

As a result of the directive execution

!DVM$ DISTRIBUTE (BLOCK, BLOCK) :: A

the array A will be distributed among calculators. The quantity and type of used calculators is set using environment variables and command line options when program is launched.

Directive

!DVM$ ALIGN B(I,J) WITH A(I,J)

specifyed joint distribution of two arrays A and B. The array B elements will be distributed on the same calculator where the corresponding elements of array A will be located.

Directive

!DVM$ PARALLEL (J,I) ON A(I,J)

specifyes the distribution of computations. The loop iterations will be executed on that calculator where the corresponding elements of array A are located.

Specification **REDUCTION (MAX(EPS))** organizes effective execution of reduction operation - global operation with data located on different calculators (maximum value finding).

Specification **SHADOW_RENEW (A)** indicates the need of remote data (shadow edges) swapping from other calculators before the loop execution. As there are no any additional specifications in **REGION** directives, the compiler automatically defines the directions of variable use - as **INOUT** (A, B,EPS).

When executing the first computing region (initialization loop) necessary memory will be allocated for the distributed parts of arrays A and B on accelerators.

When entering to the second computing region (in iterative loop) check is carried out, whether there are actual representatives for arrays A and B on the calculator. As such representatives are already present, no additional operations of actual data copying to calculators are performed.

When exiting from the computing region data in the host memory isn't updated. Before the array output in a file, it is required to copy last modifications of the array B from calculator memory using the directive **GET_ACTUAL** (B).

Annex 1. Syntax of FDVMH directives

The syntax of FDVMH directives is described by the following Backus-Naur form:

is	is by definition
or	an alternative construct
[ ]	encloses optional construct
[ ]…	encloses an optionally repeated construct which may occur zero or more times
x-list	x [, x]…

Syntax of directive

directive-line	**is**	**!DVM$** *dvm-directive*
dvm-directive	**is**	*specification-directive*
	or	*executable-directive*
specification-directive	**is**	*align-directive*
	or	*distribute-directive*
	or	*template-directive*
	or	*shadow-directive*
	or	*inherit-directive*

executable-directive **is** *realign-directive*
or *redistribute-directive*
or *parallel-directive*
or *remote-access-directive*
or *region-begin-directive*
or *region-end-directive*
or *get-actual-directive*
or *actual-directive*
or *host-section-begin-directive*
or *host-section-end-directive*

Constraints:

- A *specification-directives* may appear only in specification section.
- An *executable-directive* may appear among executable statements.
- Any expression, included in specification directive, must be the specification expression.
  A *specification expression* is an expression where each primary must be one of the following forms:
  1) constant,
  2) a variable which is the formal argument,
  3) a variable from COMMON block,
  4) reference to intrinsic function where each argument is a specification expression,
  5) a specification expression enclosed in parentheses.

No statements may be interspersed within a continued directive. A *directive-line* can't appear within a continued statement. An example of a directive with continuation in free form follows. Note that column 6 must be blank, except when signifying continuation.

```
!DVM$      ALIGN  SPACE1( I, J, K )
!DVM$*        WITH  SPACE(J , K, I )
```

Example of directive with continuation in free form:

```
!DVM$      ALIGN  SPACE1( I, J, K )   &
!DVM$         WITH  SPACE(J , K, I )
```

The following example shows to universal directive with continuation i.e. satisfying to rules as the free, as the fixed form:

```
!DVM$      ALIGN  SPACE1( I, J, K )                                  &
!DVM$&        WITH  SPACE(J , K, I )
```

Note that the sign **&** in the first line is in 73 position of the line.

DISTRIBUTE and REDISTRIBUTE directives

distribute-directive **is** *dist-action distributee dist-directive-stuff*
or *dist-action* [*dist-directive-stuff*] **::** *distributee-list*

dist-action **is** **DISTRIBUTE**
or **REDISTRIBUTE**

dist-directive-stuff **is** *dist-format-list*

Distribute **is** *array-name*

	or *template-name*
dist-format	**is BLOCK** **or WGT_BLOCK (** *block-weight-array* , *nblock***)** **or ***

Constraint:

- A length of *dist-format-list* must be equal to the array rank. That is, distribution format must be specified for every array dimension.

ALIGN and REALIGN directives

align-directive	**is** *align-action alignee align-directive-stuff* **or** *align-action* [*align-directive-stuff*] **::** *alignee-list*
align-action	**is ALIGN** **or REALIGN**
align-directive-stuff	**is (** *align-source-list* **)** *align-with-clause*
Alignee	**is** *array-name*
align-source	**is *** **or** *align-dummy*
align-dummy	**is** *scalar-int-variable*
align-with-clause	**is WITH** *align-spec*
align-spec	**is** *align-target* **(** *align-subscript-list* **)**
align-target	**is** *array-name* **or** *template-name*
align-subscript	**is** *int-expr* **or** *align-dummy-use* **or** *
align-dummy-use	**is** [*primary-expr* *****] *align-dummy* [*add-op primary-expr*]
primary-expr	**is** *int-constant* **or** *int-variable* **or** (*int-expr*)
add-op	**is +** **or -**

Constraint:

- A length of *align-source-list* must be equal to the rank of aligned array.

TEMPLATE directive

template-directive **is** **TEMPLATE** *template-decl-list*

template-decl **is** *template-name* [(*explicit-shape-spec-list*)]

Distribution of loop iterations. PARALLEL directive.

parallel-directive **is** **PARALLEL** **(** *do-variable-list* **)**
ON *iteration-align-spec*
[, *private-clause*]
[**,** *reduction-clause*]
[**,** *shadow-renew-clause*]
[**,** *remote-access-clause*] [**,** *across-clause*]

iteration-align-spec **is** *align-target* **(** *iteration-align-subscript-list* **)**

iteration-align-subscript **is** *int-expr*
or *do-variable-use*
or ***

do-variable-use **is** [*primary-expr* *****] *do-variable*
[*add-op* *primary-expr*]

Private variables. PRIVATE clause.

private-clause **is** **PRIVATE (***private_variable-list***)**

private_variable **is** *array-name*

or *scalar*

Reduction operations and variables. REDUCTION clause.

reduction-clause **is** **REDUCTION** (*reduction-op-list*)

reduction-op **is** *reduction-op-name* (*reduction-variable*)
or *reduction-loc-name* (*reduction-variable* **,**
location-variable, int-expr)

reduction-variable **is** *array-name*
or *scalar-variable-name*

location-variable **is** *array-name*

reduction-op-name **is** **SUM**
or **PRODUCT**
or **MAX**
or **MIN**
or **AND**
or **OR**
or **EQV**

	or	**NEQV**
reduction-loc-name	**is**	**MAXLOC**
	or	**MINLOC**

Constraints:

- Distributed arrays can't be used as reduction variables.
- Reduction variables are calculated and used only in statements of a certain type: the reduction statements.

Specification of array with shadow edges

shadow-directive	**is**	**SHADOW** *dist-array* (*shadow-edge-list*)
	or	**SHADOW** (*shadow-edge-list*) **::** *dist-array-list*
dist-array	**is**	*array-name*
shadow-edge	**is**	*width*
	or	*low-width* **:** *high-width*
Width	**is**	*int-expr*
low-width	**is**	*int-expr*
high-width	**is**	*int-expr*

Constraints:

- The size of left shadow edge (*low-width*), and size of right shadow edge (*high-width*) must be integer constant specification expression with value greater than or equal to 0.
- A shadow edge specification of *width* is equivalent to shadow edge of *width : width*.
- By default distributed array has a shadow edge width of 1 on both sides of each distributed dimension.

SHADOW_RENEW clause

shadow-renew-clause	**is**	**SHADOW_RENEW** **(** *renewee-list* **)**
renewee	**is**	*dist-array-name* [(*shadow-edge-list*)

Constraints:

- Width of the shadow edges filled by values must not exceed the maximal width specified in **SHADOW** directive.
- If shadow edge widths are not specified, then the maximal widths are used.

ACROSS clause

across-clause	**is**	**ACROSS** (*dependent-array-list*)
dependent-array	**is**	*dist-array-name* (*dependence-list*)

dependence **is** *flow-dep-length : anti-dep-length*

flow-dep-length **is** *int-constant*

anti-dep-length is *int-constant*

Constraint:

- In each array reference, data dependence may appear only in one distributed dimension.

REMOTE_ACCESS directive

remote-access-directive **is** **REMOTE_ACCESS**
(*regular-reference-list*)

regular-reference **is** *dist-array-name* [(*regular-subscript-list*)]

regular-subscript **is** *int-expr*
or *do-variable-use*
or **:**

remote-access-clause **is** *remote-access-directive*

stride **is** *int-expr*

REGION directive

region-begin-directive **is** **REGION** [*region-clause-list*]

region-clause **is** *in-out-local-clause*

in-out-local-clause **is** **IN (***in-out-local-variable-list***)**
or **OUT (***in-out-local-variable-list***)**
or **LOCAL (***in-out-local-variable-list***)**
or **INOUT (***in-out-local-variable-list***)**
or **INLOCAL (***in-out-local-variable-list***)**

in-out-local-variable **is** *array-name*
or *array-name***(***subarray-subscript-list***)**
or *scalar-variable-name*

subarray-subscript **is** *int-expr*
or [*int-expr*] **:** [*int-expr*]

region-end-directive **is END REGION**

GET_ACTUAL and ACTUAL directives

get-actual-directive **is GET_ACTUAL**([*actual-variable-list*])

actual-directive **is ACTUAL**([*actual-variable-list*])

actual-variable **is** *array-name*
or *array-name*(*subarray-subscript -list*)
or *scalar-variable-name*

INHERIT directive

inherit-directive **is INHERIT** *dummy-array-name-list*

Annex 2. Environment variables for DVMH-programs

A user can to modify environment variables which are defined in **dvm** startup file.

DVMH_NUM_CUDAS - a number of CUDA devices for use by one process. If the variable isn't set, its optimal value is defined by RunTime System. RunTime System provides effective usage of all resources of a node. Optimal value depends on a number of devices and a number of launched processes on the node. The value of **DVMH_NUM_CUDAS** variable can't exceed physically available quantity of accelerators.

DVMH_NUM_THREADS – a number of the threads operating at CPU. If the variable isn't set, its optimal value is defined by RunTime System. RunTime System provides effective usage of all resources of a node. Optimal value depends on a number of devices on the node, a number of launched processes on the node and a number of CUDA devices (**DVMH_NUM_CUDAS**) to use by one process.. The value of **DVMH_NUM_THREADS** variable can be any positive number.

DVMH_LOGFILE - a log file name. The file name is set in quotes, it is possible to use construction %d (for example, 'dvmh_%d.log'), in this case the log of each MPI process will be in separate file. If the variable isn't set, the standard error output stream is used.

DVMH_LOGLEVEL - detail level of log file. It is set as integer decimal number. There are the following levels:

0 – errors of fatal err level are output,
1 - err,
2 - warning,
3 - info,
4 - debug,
5 - trace.

Negative value is equated to zero. The value more than 5 is equated to 5. If the variable isn't set, level 1 - err is set.

DVMH_PPN – a number of processes per a node. It determines distribution of computational resources by processes — everyone is given an identical portion from computational resources of the node. If the variable isn't set, it is equated to one (each process uses all resources of a node).

DVMH_COMPARE_FLOATS_EPS - the accuracy of comparing of variables with floating point of single precision on the relative and absolute inaccuracy in case of comparative debugging. By default it is equal FLT_EPSILON*1000, where FLT_EPSILON is minimum positive x such that 1.0+x=1.0.

DVMH_COMPARE_DOUBLES_EPS - the accuracy of comparing of variables with a floating point of double precision on the relative and absolute inaccuracy in case of comparative debugging. By default it is equal to DBL_EPSILON*10000, where DBL_EPSILON is minimum positive x such that 1.0+x=1.0.

DVMH_COMPARE_LONGDOUBLES_EPS - the comparing accuracy of variables with floating point of long double precision on the relative and absolute inaccuracy in case of comparative debugging. By default LDBL_EPSILON*100000, where LDBL_EPSILON – minimum positive x such that 1.0+x=1.0.

DVMH_CPU_PERF - the relative productivity of CPU (summary of all cores of the CPU), it is used for job distribution in the planning mode 1. It is equal 1 by default.

DVMH_CUDAS_PERF - the relative productivity of GPU, it is set by the looped-back list of real numbers through a space or a comma. It is equal 1 by default.

Annex 3. Compilation options for DVMH programs

-noH	-	a mode of DVMH directive ignoring.
-f90,-FR	-	specifying the free form of record of source FORTRAN codes and FDVMH directives. If files have extension **f90, f95, f03**, it is considered that they are in the free form.
-FI	-	specifying the fixed form of record of source FORTRAN codes and FDVMH directives.
-autoTfm	-	the mode of an array dynamic reordering (the mode of dimension reodering of the array for optimization of GPU memory access).
-Opl	-	Parallel loops out of regions are executed on a host processor, as well as the loops inside regions. In this mode descriptions of privacy for loops out of regions are necessary.
-gpuO1	-	optimization of private variable usage for often used array elements which are mapped on registers by the compiler.
-noCuda	-	compilation process control – the compiler doesn't prepare execution of regions on CUDA devices.

-collapse<N> - compilation mode when for each parallel loop if it is executed on CPU the COLLAPSE (N) specification is added to OpenMP the directive.

-mmic - program compilation for execution on the Intel Xeon Phi coprocessor.

If DVMH program is compiled with option –noH DVMH directives are ignored, and the DVMH program is transformed into the DVM program.

Usage of optimization options (– autoTfm, - Opl, – gpuO1) can promote the increase of the program performance.

The optimization option -collapse<N> can optimize the program execution on Xeon Phi.

Annex 4. Diagnostics messages of DVMH debugger

The common format of error messages of dynamic debugger:

(<process number>**)**<context> **File:** <file**>, Line:** <line> **(**<count> **times)**<error message>

where:

<process number> – number of processor, where error occurred. It is reported only if a program is executed on several processors.

<context> – context, where the error occurred. It can be one of the following forms:

sequential branch	–	the error occurred in sequential part of the program;
Loop(No(N_1), Iter(I_1,I_2,…)), …, Loop(No(N_m), Iter(I_1,I_2,…))	–	the error occurred when m-dimensional loop was executed.

<file> – name of the file, where the error occurred.

<line> – line number.

<count> – a number of given error repetitions in the given context. It is output when all detected errors are reported.

<error message> – error description message.

1. Dynamical control

Error message	Description
Writing to read-only variable<var>	Writing to read-only variable was detected.
Using non-initialized private variable <var>	Access to non-initialized variable was detected.
Using non-initialized element <elem>	Access to non-initialized distributed array element was detected.
Using variable <var> before asynchronous reduction competed	Access to reduction variable before reduction operation completion.

Access to non-local element <elem>	Access to non-local element of distributed array.
Writing to shadow element <elem> of array	Writing to shadow element of array.
Shadow element <elem> was not updated	Access to shadow elements before completion of shadow edge renewing operation.
Data dependence in loop due to access to element <elem>	Data dependence in parallel loop was detected.
Using shadow element <elem> before asynchronous shadow renew competed	Usage of shadow element <elem> of distributed array during shadow edge renewing operation execution.
Writing to remote data buffer <var>	Writing to remote data buffer <var>.
Write to remote element <elem> in sequential branch	Writing to distributed array element <elem> in sequential branch of the program without own computation specification.
Reading remote element <elem> in sequential branch	Using not local element <elem> of distributed array in sequensial branch of the program.
WAIT for reduction without START	Waiting for reduction completion is issued without the reduction start.
Using an element outside of array limits: <elem>	Access to array element beyond its limits.
START for reduction without WAIT	Absence of waiting for asynchronous reduction completion operation for corresponding operation of asynchronous reduction start.
Reduction operation was not started	Reduction variable is specified, but corresponding computation of reduction operation was never started.

2. Trace accumulation and comparison

Error message	Description
Bad file structure	Trace file structure is incorrect.
Undefined keyword	Unknown keyword appears in trace file.
Bad command syntax	Wrong structure of trace record.
Can't open a file <file name>	Specified file can’t be opened.
Trace file <file name> is empty	Specified trace file is empty.
Bad trace structure (missing current program construct)	Trace file structure is incorrect. Record of executable construct beginning is missing.
No current program construct	Record of executable construct beginning is missing.
Unexpected task or iteration of loop	There is no record about given iteration or task execution in reference trace.
Double execution of task or iteration, No = <iter no>	Repeated execution of the same iteration or task.
Unexpected execution of program construct	There is no record about given loop or task execution beginning in reference trace.

Abnormal loop exit	Loop end doesn't correspond to the record in reference trace.
Unexpected use of variable	There is no given variable usage record in reference trace.
Unexpected trace record	There is no given event performance record in reference trace.
Different <type> values: <standard value> != <current value>	Variable value differs from the variable value in reference trace.
Different <type> values of reduction variable: <standard value> != <current value>	Result value of reduction computation differs from the value in reference trace.

3. Structure of trace configuration file

 # Trace size = <size of full trace file in bytes>
 # String count = <a number of lines in full trace file>

SL, PL or TR <construct number> (<number of surrounding construct>) [<construct rank>] {<file name>, <line number>} = <trace accumulation level>, (<dimension>:<first iteration>, < last iteration >, <iteration step>)

 # Trace size = <construct trace size in bytes for given construction for specified trace level>
 # String count = <number of construct trace lines for specified trace level>
 # Count of traced iterations = <number of traced loop iterations or tasks>

EL: <construct number>

. . . . . .

SL, PL or TR <construct number> (<number of surrounding construct>) [<construct rank>] {<file name>, <line number>} = <trace accumulation level>, (<dimension>:<first iteration>, < last iteration >, <iteration step>)

 # Trace size = <construct trace size in bytes for given construction for specified trace level>
 # String count = <number of construct trace lines for specified trace level>
 # Count of traced iterations = <number of traced loop iterations or tasks>

EL: <construct number>

4. Execution trace structure

When executions are traced, accumulated trace information consists of two parts:

- trace header;
- trace body (may be omitted).

The header exists in trace even if the trace accumulation is disabled for whole program. Its structure looks like the structure of trace configuration loop, but without calculated values of trace size for whole program and for loops:

MODE = <accumulation trace level for whole program>,

SL, **PL** or **TR** <construct number> (<number of surrounding construct>) [<construct rank>] {<file name>, <line number>} = <trace accumulation level>, (<dimension>:<first iteration>, < last iteration >, <iteration step>)

EL: <construct number>

. . . . . .

SL, **PL** or **TR** <construct number> (<number of surrounding construct>) [<construct rank>] {<file name>, <line number>} = <trace accumulation level>, (<dimension>:<first iteration>, < last iteration >, <iteration step>)

EL: <construct number>

Trace body is absent, when trace accumulation is disabled for whole program. Otherwise trace body consists of a lot of records of the following types:

- **Reading variable.**
  RD: [<variable type>] <variable name> = <value>; {<file name>, <line number>}
- **Referring to variable (before expression computation).**
  BW: [<variable type>] <variable name>; {<file name>, <line number>}
- **Result of assigning value to variable.**
  AW: [<variable type>] <variable name> = <value>; {<file name>, <line number>}
- **Reading reduction variable.**
  RV_RD: [<variable type>] <variable name> = <value>; {<file name>, <line number>}
- **Referring to reduction variable (before expression computation).**
  RV_BW: [<variable type>] <variable name>; {<file name>, <line number>}
- **Result of assigning value to reduction variable.**
  RV_AW: [<variable type>] <variable name> = <value>; {<file name>, <line number>}
- **Result of reduction computation.**
  RV: [<variable type>] <value>; {<file name>, <line number>}
- **Skipping a group of statements when referring an element of distributed array in a sequential branch of the program.**
  SKP: {<file name>, <line number>}
- **Parallel loop beginning.**
  PL: <loop number> (<parent construct number or empty>) [<loop rank>] = <trace level: FULL, MODIFY, MINIMAL, NONE>, (<traced iteration range (can be absent)>); {<file name>, <line number>}
- **Sequential loop beginning.**
  SL: <loop number> (<parent construct number or empty>) [<loop rank>] = <trace level: FULL, MODIFY, MINIMAL, NONE>, (<traced iteration range (can be absent)>); {<file name>, <line number>}
- **Task region beginning.**
  TR: <region number> (<parent construct number or empty>) [region rank>] = <trace level: FULL, MODIFY, MINIMAL, NONE>, (<traced task range>); {<file name>, <line number>}
- **Beginning of iteration (put in trace file only when the most nested loop iteration is executed) or parallel task.**

IT: <absolute iteration index (calculated from all values of all iteration variables) or task number>, (<iteration variable value>,<iteration variable value>,…).

- **End of parallel loop or task region execution.**
  EL: <construct number>; {<file name>, <line number>}

Annex 5. Error messages of statistics accumulation

Statistics: not enough memory for interval, data were not wrote to the file,

Statistics: number of ends of interval > number of begins of interval, data were not wrote to the file,

Statistics: end of interval nline = <N>, name = <name>, no end nline = <N> name =<name>, data were not wrote to the file,

Statistics: StatBufLength=<length>, increase buffer's size by <N> bytes, data were not wrote to the file,

Statistics: StatBufLength=<length>, not enough memory for times of collective operations, increase buffer's size by <N> bytes, only part of times of collective operations and all intervals were wrote to the file.

Statistics warning :used return or goto, times may be incorrect