Язык Fortran DVMH. Fortran DVMH компилятор. Компиляция, выполнение и отладка DVMH-программ.

1 Введение

В последнее время все большее распространение получают вычислительные кластеры, в узлах которых установлены ускорители. В основном, это графические процессоры компании NVIDIA. В 2012 году появились кластеры с ускорителями другой архитектуры – Xeon Phi от компании Intel.

В 2011 году в Институте прикладной математики им. М.В. Келдыша РАН была разработана модель параллельного программирования для кластеров с ускорителями. Модель является расширением модели DVM и получила название DVMH (DVM for Heterogeneous systems).

Разработанный в рамках модели язык Fortran-DVMH (FDVMH) упрощает процесс написания параллельных программ, а также позволяет с небольшими изменениями перевести программу для кластера (DVM-программу) в программу для кластера с ускорителями (DVMH-программу).

Язык FDVMH представляет собой язык Фортран 95, расширенный спецификациями параллелизма. Эти спецификации оформлены в виде специальных комментариев, которые называются директивами. Директивы “невидимы” для стандартных компиляторов, что позволяет иметь один вариант программы для последовательного и параллельного выполнения.

Язык FDVMH позволяет написать программу (DVMH-программу), которая может выполняться и как последовательная, и как параллельная для кластеров с ускорителями и без ускорителей.

Для DVMH-программ разработаны средства функциональной отладки и отладки эффективности.

Основная работа по реализации модели выполнения параллельной программы осуществляется системой поддержки выполнения DVMH-программ (RTS – RunTime System).

Управление работой системы поддержки, работой отладчика параллельных программ, сбором статистики осуществляется при помощи параметров. В документе есть описание настройки DVMH-системы и способы коррекции параметров, необходимых для работы.

2 Словарь терминов

В данном разделе приведен список терминов и сокращений, используемых в документе.

2.1 Описание аппаратных вычислительных систем

Вычислительный кластер – совокупность связанных между собой вычислительных узлов (компьютеров). В состав **узла кластера,** помимо центрального процессорного устройства (ЦПУ) может входить несколько специализированных процессорных устройств – ускорителей.

ЦПУ, центральное процессорное устройство – несколько универсальных многоядерных процессоров, имеющих общую оперативную память.

Ускоритель – специализированное процессорное устройство, подключаемое к ЦПУ и ориентированное на высокопроизводительное выполнение некоторых программ. Если ускоритель имеет собственную память, то для запуска на нем программы центральное процессорное устройство переписывает ее и требуемые ей данные из оперативной памяти ЦПУ в оперативную память ускорителя.

В**ычислительное устройство, вычислитель** – ЦПУ или ускоритель.

ГПУ - графический процессор, один из видов ускорителей.

CUDA-устройство - графический процессор фирмы NVIDIA.

Сопроцессор Intel Xeon Phi – специализированный процессор, который может использоваться в качестве ЦПУ или в качестве ускорителя.

Ограничение.

Текущая версия DVM-системы позволяет использовать в качестве ускорителя только CUDA-устройство, а сопроцессор Intel Xeon Phi - только в качестве ЦПУ.

2.2 Описание виртуальных вычислительных систем, на которых может выполняться DVMH-программа

Виртуальная многопроцессорная система (или **массив виртуальных процессоров)** – та машина, которая предоставляется DVMH-программе пользователя аппаратурой и базовым системным программным обеспечением. Для распределённой ЭВМ примером такой машины может служить MPI-машина. В этом случае многопроцессорная система – это группа MPI-процессов, которые создаются при запуске параллельной программы на выполнение. На один узел аппаратного кластера может быть отображен один или несколько MPI-процессов. Число процессоров виртуальной многопроцессорной системы и её представление в виде многомерной решетки задаётся в командной строке при запуске программы.

В модели FDVMH **виртуальный процессор стал гетерогенным** – он состоит из виртуального хост-процессора (на нем начинает выполняться MPI-процесс и на нем выполняются фрагменты программы вне регионов), виртуального мультипроцессора (на нем выполняются регионы, у которых в качестве целевой архитектуры задана OpenMP-архитектура), и нескольких виртуальных ускорителей разной архитектуры. При этом виртуальный хост-процессор и виртуальный мультипроцессор отображаются на ЦПУ и работают на общей оперативной памяти, а каждый виртуальный ускоритель имеет свою память и отображается на отдельный аппаратный ускоритель соответствующей архитектуры.

3 Модель DVMH и разные архитектуры

Модель программирования DVMH и язык FDVMH позволяют разрабатывать параллельные программы для кластеров, в узлах которых помимо универсальных многоядерных процессоров установлены ускорители компании NVidia и сопроцессоры Intel Xeon Phi. Модель поддерживает использование всех перечисленных архитектур как по отдельности, так и одновременно в рамках одной программы.

При разработке модели DVMH за основу была взята модель DVM [1], в которую добавлены конструкции для организации вычислений на кластерах с ускорителями и спецификации потоков данных, для управления перемещением данных между оперативной памятью центрального процессора и памятями ускорителей.

Модель параллелизма базируется на специальной форме параллелизма по данным: одна программа – множество потоков данных . В этой модели одна и та же программа выполняется на каждом виртуальном процессоре, но каждый процессор выполняет свое подмножество операторов в соответствии с распределением данных.

Вначале программист определяет массивы (***распределенные данные***) и витки циклов, которые могут быть распределены между процессорами.

Распределенные массивы специфицируются директивами отображения данных (см. разделы 4.1.1 и 4.1.2), а параллельные циклы - директивами распределения вычислений (см. раздел 4.1.3).

Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый процессор (***размноженные данные***). Размноженная переменная должна иметь одно и то же значение на каждом процессоре за исключением редукционных переменных (см. раздел 4.3.5) и приватных переменных (см. раздел 4.2) в параллельном цикле.

Распределение данных определяет множество локальных или ***собственных переменных*** для каждого процессора. Множество собственных переменных определяет ***правило собственных вычислений***: процессор присваивает значения только собственным переменным.

При выполнении программы процессору могут потребоваться как значения собственных переменных, так и значения несобственных (***удаленных***) переменных. Все удаленные переменные должны быть указаны в директивах доступа к удаленным данным (см. раздел 4.3).

Далее программист определяет фрагменты кода, которые могут быть выполнены на ускорителях. Такие фрагменты называются ***вычислительными регионами*** или просто ***регионами***.

Фрагменты программы вне регионов всегда выполняются на центральном процессоре.

Для каждого региона указываются данные, необходимые для его выполнения (входные, выходные, локальные).

Перемещения данных между центральным процессором и ускорителями осуществляются в основном автоматически в соответствии с имеющейся в описании регионов информацией об используемых ими данных. Для управления перемещениями данных между ускорителями и центральным процессором предусмотрены специальные директивы (см. раздел 4.5).

Параллелизм в DVMH-программах проявляется на нескольких уровнях:

- Распределение данных и вычислений по MPI-процессам. Этот уровень задается директивами распределения и перераспределения данных и спецификациями параллельных подзадач и циклов.
- Распределение данных и вычислений по вычислительным устройствам при входе в вычислительный регион.
- Параллельная обработка в рамках конкретного вычислительного устройства. Этот уровень появляется при входе в параллельный цикл, находящийся внутри вычислительного региона и для параллельных циклов вне региона при выполнении в специальном режиме, который задается при помощи опции -Opl (см. **Приложение 3**).

4 Распараллеливание программ на языке Fortran-DVMH

Параллелизм программы описывается на языке FDVMH с помощью директив. Каждая директива оформляется в виде специального комментария:

!DVM$ <DVMH-директива>

Формальный синтаксис директив и правила записи директив в свободной и фиксированной формах приведены в **Приложении 1**. Все примеры написаны в фиксированной форме.

Распараллеливание программы в модели DVMH можно разделить на следующие этапы

1. Распределение данных (массивов) и вычислений (параллельных циклов) на массив виртуальных процессоров.
2. Определение и спецификация удаленных данных.
3. Определение регионов для выполнения на ускорителях.
4. Управление перемещением данных между памятью ЦПУ и памятями ускорителей.

4.1 Распределение массивов и параллельных циклов

На первом этапе используются директивы **DISTRIBUTE**, **ALIGN** и **PARALLEL**. Если рассматривать эти директивы на абстрактном уровне, то они устанавливают соответствие между точками индексных (дискретных) пространств двух объектов. При этом используются индексные пространства следующих объектов:

P – индексное пространство массива виртуальных процессоров. Массив виртуальных процессоров определяет пользователь и задает его при запуске программы на выполнение.

$\mathbf{A_i}$ - индексное пространство i–го массива данных.

$\mathbf{L_j}$ - индексное пространство j–го параллельного цикла. Параллельный цикл рассматривается как массив, элементами которого является витки цикла. Количество измерений этого массива определяется количеством заголовков цикла.

Директивы устанавливают соответствие между точками (элементами) следующих объектов:

DISTRIBUTE: $\mathbf{A_i \Rightarrow P}$. Каждой точке (виртуальному процессору) **P** ставится в соответствие подмножество точек (элементов массива) $\mathbf{A_i}$.

ALIGN: $\mathbf{A_{i1} \Rightarrow A_{i2}}$. Каждой точке (элементу массива) $\mathbf{A_{i2}}$ ставится в соответствие подмножество точек (элементов массива) $\mathbf{A_{i1}}$.

PARALLEL: $\mathbf{L_j \Rightarrow A_i}$. Каждой точке (элементу массива) $\mathbf{A_i}$ ставится в соответствие подмножество точек (витков цикла) $\mathbf{L_j}$.

Совокупность этих директив определяет для каждого виртуального процессора подмножество элементов массивов и витков параллельных циклов.

Данные и операторы, не специфицированные этими директивами, автоматически распределяются на каждый виртуальный процессор (размноженные данные и не распараллеливаемые вычисления).

4.1.1 Распределение массивов. Директивы DISTRIBUTE и REDISTRIBUTE.

Язык FDVMH поддерживает распределение блоками (равными и неравными) и распределение через выравнивание.

Распределение массива **A** описывается следующей директивой

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

где f_i - формат распределения для i-го измерения:

f_i	=	**BLOCK**	-	распределение равными блоками
	=	**WGT_BLOCK** (*WB, NBL*)	-	распределение блоками по их относительным «весам»
	=	*****	-	нераспределенное измерение

k – количество измерений

Если ***fi* = BLOCK**, то измерение массива распределяется преимущественно равными блоками. Если количество элементов массива (**N**) не превышает число процессоров (**P**), то на каждый процессор попадает либо один элемент массива, либо ни одного. Если количество элементов массива (**N**) больше числа процессоров, то количество элементов на процессоре определяется по формуле:

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

где: **k**-номер процессора, **N** – число элементов массива, **P** – число процессоров.

Если ***fi* = WGT_BLOCK**(*WB, NBL*), то измерение распределяется по их относительным “весам”. *WB* – одномерный массив вещественных чисел размера *NBL*.

Для $1 \le i \le NBL$, *WB(i)* определяет вес *i*-ого блока. Блоки распределяются на *P* процессоров с балансировкой сумм весов блоков на каждом процессоре. При этом должно выполняться условие

$P \le NBL$

Вес процессора определяется как сумма весов всех блоков, распределенных на него. Измерение массива распределяется пропорционально весам процессоров.

Формат **BLOCK** является частным случаем формата **WGT_BLOCK(***WB*,*P***)**, где $WB(i) = 1$ для $1 \le i \le P$ и $NBL = P$.

Если ***fi* = ***, то на каждый процессор распределяется целое измерение (локальное измерение).

Количество форматов **BLOCK** и **WGT_BLOCK**(*WB, NBL*) определяет количество измерений массива виртуальных процессоров. Если мы пронумеруем форматы **BLOCK** и **WGT_BLOCK(***WB*,*P***)**, как $fb_1, \ldots, fb_n$, то измерение с форматом

fb_i отображается на i–ое измерение массива виртуальных процессоров, а количество блоков определяется размером i–го измерения массива виртуальных процессоров. В этом случае при запуске программы на выполнение можно задавать любой n–мерный массив виртуальных процессоров.

Несколько одинаково распределяемых массивов (*A1, A2,…*) можно распределить одной директивой вида

!DVM$ DISTRIBUTE ($f_1 \ldots f_k$) :: A1, A2,…

где f_i - формат распределения для i-го измерения.

При этом массивы должны иметь одинаковое число измерений, но необязательно одинаковые размеры измерений.

Массив, который специфицирован директивой **DISTRIBUTE,** можно перераспределить следующей директивой:

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

Директива **REDISTRIBUTE** может применяться только к массивам со спецификацией **DYNAMIC:**

!DVM$ DYNAMIC A

Для перераспределяемого массива можно не указывать начальное распределение, это так называемое отложенное распределение. Директива **DISTRIBUTE** будет иметь вид:

!DVM$ DISTRIBUTE :: A

спецификацию **DYNAMIC** в этом случае можно не указывать.

Если в директиве **DISTRIBUTE** указан массив с атрибутом ALLOCATABLE, то распределение откладывается до выполнения оператора ALLOCATE**,** который размещает массив в памяти.

Директива **REDISTRIBUTE** для динамически размещаемого массива может выполняться только после выполнения оператора ALLOCATE.

Пример. Распределение блоками.

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

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

При запуске на четырех процессорах распределение будет следующим:

Процессор	**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**	

Пример. Распределение блоками по весам.

```
      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. /
```

Процессор **A**

R(1)

1
2
3
4

R(2)

5
6
7
8

R(3)

9
10

R(4)

11
12

Распределение по формату **WGT_BLOCK** можно выполнить для любого числа процессоров в диапазоне от *1* до *NBL*. Для данного примера размер массива процессоров *R* может изменяться от 1 до 12.

Пример. Распределение динамически размещаемых массивов.

```
      SUBROUTINE SP(K)
      REAL, ALLOCATABLE, DIMENSION (:) :: A
!DVM$ DISTRIBUTE (BLOCK) :: A
      READ(*,*) N
```

```
C       размещение и распределение массива A
        ALLOCATE (A(N))
        . . .
        END
```

4.1.2 Локализация данных. Директивы ALIGN и REALIGN

Выравнивание массива **A** на распределенный массив **B** ставит в соответствие каждому элементу массива **A** элемент или секцию массива **B**. При распределении массива **B** одновременно будет распределяться массив **A**. Если на данный процессор распределен элемент **B**, то на этот же процессор будет распределен элемент массива **A**, поставленный в соответствие выравниванием.

Метод распределения через выравнивание выполняет следующие две функции.

1) Одинаковое распределение массивов одной формы на один массив процессоров не всегда гарантирует, что соответствующие элементы будут размещены на одном процессоре. Это вынуждает специфицировать удаленный доступ (см. раздел 4.3) там, где его возможно нет. Гарантию размещения на одном процессоре дает только выравнивание соответствующих элементов массивов.
2) На один массив может быть выровнено несколько массивов. Изменение распределения одного массива директивой **REDISTRIBUTE** вызовет соответствующее изменение распределения группы массивов.

Основным способом локализации данных (т.е. уменьшения удаленных данных) является совместное распределение нескольких массивов. Совместное распределение двух массивов **A** и **B** описывается директивой выравнивания массивов.

!DVM$ ALIGN A ($a_1 \ldots a_n$) WITH B ($b_1 \ldots b_m$)

где:

a_i - параметр i–го измерения выравниваемого массива **A**,
b_j - параметр j–го измерения базового массива **B**,
n - количество измерений массива **A**,
m - количество измерений массива **B**.

Эта директива ставит в соответствие каждому элементу массива **B** некоторое подмножество элементов массива **A**. Установленное подмножество элементов массива **A** будет распределено на тот процессор, где будет размещен соответствующий элемент массива **B**. Параметры выравниваемого массива **A** и базового массива **B** могут иметь следующий вид:

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

где:

ID_i , ID_j - идентификаторы
c, d - целочисленные константы

Рассмотрим семантику этих обозначений.

Если a_i = * , то i–тое измерение массива **A** целиком распределяется на каждый процессор, где распределен хотя бы один элемент **B** (размножение, локальное измерение).

Если b_j = * , то выполнение директивы **ALIGN** не зависит от j-ого измерения массива **B** (коллапс, т.е. измерение как бы не существует при установлении соответствия).

Если $a_i = ID_i$, то обязательно существует одно и только одно $b_j = c*ID_j+d$, где $ID_i=ID_j$. Равенство $ID_i=ID_j$ означает, что i–тое измерение массива **A** ставится в соответствие j-ому измерению массива **B**. Соответствие элементов устанавливается функцией $c*ID_j+d$. При этом индексы базового массива не должны выходить за пределы индексного пространства выравниваемого массива, иначе нужно осуществлять выравнивание по шаблону (см. раздел 4.3.1).

Примеры директивы ALIGN и семантика.

!DVM$ ALIGN A(I) WITH B(2*I+1)
Распределить элемент **A(I)** и **B(2*I+1)** на один и тот же процессор.

!DVM$ ALIGN A(I,J) WITH B(J,I)
Распределить элемент **A(I,J)** и **B(J,I)** на один и тот же процессор.

!DVM$ ALIGN A(I) WITH B(*, I)
Распределить элемент **A(I)** на те процессоры, где размещен хотя бы один элемент **I**-ого столбца **B.**

!DVM$ ALIGN A(I, *) WITH B(I)
Распределить **I**-ую строку **A** и элемент **B(I)** на один процессор, т.е. размножить второе измерение массива **A.**

Несколько массивов (*A1, A2,…*) можно выровнять одинаковым образом на один и тот же массив *B* одной директивой вида

!DVM$ ALIGN (a_1… a_n) WITH B (b_1… b_m) :: A1, A2, …

При этом массивы *A1, A2…* должны иметь одинаковое число измерений *(n)*, но необязательно одинаковые размеры измерений.

Изменить параметры выравнивания и/или базовый массив можно с помощью директивы

!DVM$ REALIGN A (a_1… a_n) WITH C (c_1… c_k)

Выравниваемый массив должен быть описан как **DYNAMIC.**

Начальное выравнивание может быть не задано, это так называемое отложенное выравнивание. Директива **ALIGN** будет иметь вид:

!DVM$ ALIGN :: A,

а спецификацию **DYNAMIC** в этом случае можно не указывать.

Если в директиве **ALIGN** в качестве выравниваемого массива указана переменная с атрибутом ALLOCATABLE, то выполнение директивы откладывается

до выполнения оператора ALLOCATE. Директива **REALIGN** может выполняться только после выполнения оператора ALLOCATE.

Пример. Выравнивание динамически размещаемых массивов.

```
          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
```

Отметим, что операторы ALLOCATE не могут быть выполнены в обратном порядке.

4.1.3 Распределение витков параллельного цикла. Директива PARALLEL

Параллельный цикл в модели DVMH рассматривается как массив витков цикла. Количество измерений такого массива равно количеству заголовков параллельного цикла. Размер каждого измерения определяется параметрами соответствующего заголовка цикла. Чтобы такое представление было правильным, необходимо выполнение следующих условий:

- заголовки параллельного цикла не должны разделяться другими операторами (тесно-гнездовой цикл);
- параметры заголовков параллельного цикла не должны изменяться в процессе выполнения цикла (прямоугольное индексное пространство);
- виток цикла должен быть неделимым объектом и выполняться на одном процессоре. Поэтому левые части операторов присваивания одного витка цикла должны быть распределены на один процессор (согласование с правилом собственных вычислений).

Распределение витков параллельного цикла осуществляется следующей директивой:

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

где:

I_j - переменная (индекс) j–го заголовка параллельного цикла,

n - количество заголовков цикла,

A - идентификатор массива,

m - количество измерений массива,

$e_i = a * I_k + b$, a, b – целочисленные переменные

I_k – переменная (индекс) k-го заголовка цикла.

Это выражение означает следующее:

- k-ое измерение (заголовок цикла) массива витков цикла ставится в соответствие i–ому измерению массива данных,
- соответствие витка цикла и элемента массива устанавливается линейной функцией $a^* I_k+b$.

Директива **PARALLEL** каждый виток параллельного цикла ставит в соответствие некоторому элементу массива. Это означает, что виток цикла будет выполняться на том процессоре, на который распределен соответствующий элемент массива. По семантике директива **PARALLEL** аналогична директиве **ALIGN**. Отличием является то, что вместо выравниваемого массива данных используется массив витков параллельного цикла.

Пример.

```
!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
```

Для того, чтобы левые части операторов присваивания одного витка цикла были распределены на одном процессоре, необходимо к описанию массива B применить следующую директиву:

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

Если невозможно разместить левые части операторов на одном процессоре, то цикл необходимо разделить на несколько циклов, для которых выполняются условия массива витков цикла.

Пример.

```
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
```

Цикл разделен на 2 цикла, каждый из которых удовлетворяет условию параллельного цикла.

Параллельный цикл должен удовлетворять дополнительно следующим условиям:

- распределенные измерения массивов индексируются только регулярными выражениями типа $a*I + b$, где I - индекс цикла;
- левая часть оператора присваивания является ссылкой на распределенный массив, редукционную переменную (см. раздел 4.3.5) или переменную, описанную в теле цикла;
- в теле цикла нет DVMH-директив.

4.2 Приватные переменные

Переменная называется приватной, если ее использование локализовано в пределах одного витка цикла.

Приватные переменные не могут быть распределенными массивами. Значение приватной переменной не определено в начале витка цикла и не используется после витка цикла, поэтому в каждом витке цикла может использоваться свой экземпляр приватной переменной.

Если в параллельном цикле используются приватные переменные, то в директиве **PARALLEL** необходимо указать дополнительную спецификацию **PRIVATE**:

```
!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
```

Если в цикле несколько приватных переменных, то они должны быть перечислены через запятую в круглых скобках после ключевого слова **PRIVATE**.

4.3 Удаленные данные. Их виды и спецификация

Данные, которые вычисляются на одном процессоре, а используются на других, называются ***удаленными***.

Выявление удаленных данных производится с помощью анализа операторов присваивания. Оператор присваивания всегда выполняется на том процессоре, где размещены данные его левой части. Если данные левой и правой частей оператора присваивания размещены на одном процессоре, то удаленных данных для этого оператора не будет. В противном случае необходимо определить вид и размер удаленных данных и описать их соответствующими директивами (см. ниже). Такой анализ будем называть анализом локализации данных.

Цель распараллеливания - максимальный параллелизм при минимизации удаленных данных (максимум локализации).

В следующих разделах будем использовать фрагмент программы

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

где *expr* - выражение.

Изменяя состав выражения *expr*, рассмотрим основные способы локализации данных и спецификации удаленных данных для одномерных массивов (одного измерения многомерного массива).

4.3.1 Локализация данных. Выравнивание по шаблону.

Пусть

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

Если A(I), B(I) и C(I) для каждого I распределены на одном процессоре, то для этого оператора не существует удаленных данных. Локализацию данных можно выполнить директивами **ALIGN**:

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

Рассмотрим оператор

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

где d1,d2 – положительные константы.

Для этого оператора невозможно выполнить полную локализацию данных, используя массив A, т.к. смещение +d1 и -d2 выводят за пределы индексного пространства массива A. Поэтому необходимо применить выравнивание по шаблону следующим образом

```
!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 )
```

В этом случае A(I), B(I+d1) и C(I-d2) для каждого I будут распределены на один процессор. Шаблон TABC определяет некоторое индексное пространство, которое является посредником между массивом данных и массивом виртуальных процессоров. Элементы шаблона не имеют физического представления в памяти. Они указывают процессоры, на которые должны быть распределены соответствующие элементы массивов данных.

4.3.2 Удаленные данные вида SHADOW

Пусть

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

В этом случае невозможна полная локализация данных. Тем не менее необходимо выполнить частичную локализацию данных с помощью директивы

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

После выполнения этой директивы точно определяется местонахождение удаленных данных. Для вычисления всех A(I) на одном процессоре будут использоваться d1 элементов массива B с процессора, на котором располагаются

элементы массива с меньшими индексами и d2 – с большими индексами. Такие данные будем называть удаленными данными типа **SHADOW** (***теневыми гранями***).

Для спецификации размера теневых граней служит директива **SHADOW**:

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

Для несколько массивов (*A1, A2,…*) с одинаковыми размерами теневых граней можно использовать директиву вида

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

По умолчании размер теневых граней равен 1.

В каждом параллельном цикле, где используются удаленные данные типа **SHADOW** массива B, необходимо указать дополнительную спецификацию в директиве **PARALLEL**:

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

4.3.3 Удаленные данные вида ACROSS

Пусть

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

Как и в предыдущем разделе, необходимо описать размер удаленных данных директивой:

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

Но в директиве **PARALLEL** добавляется спецификация **ACROSS**:

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

Отличие данных типа **ACROSS** от данных типа **SHADOW** заключается в следующем: невозможно независимое выполнение витков цикла, т.к. прежде чем вычислить A(I), необходимо вычислить A(I-d1). В спецификации **ACROSS** перечисляются все распределенные массивы, по которым существует регулярная зависимость по данным.

4.3.4 Удаленные данные вида REMOTE

Пусть

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

где C - распределенный массив.

В этом случае в директиве **PARALLEL** необходимо указать следующую спецификацию **REMOTE_ACCESS**:

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

Если вне параллельного цикла встречается

A(I)=C(5)
или
A(I)=C(N),
то перед такими операторами нужно указать директиву

!DVM$ REMOTE_ACCESS (C(5))

или

!DVM$ REMOTE_ACCESS (C(N))

соответственно**.**

4.3.5 Удаленные данные вида REDUCTION

Пусть в цикле вычисляются

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

Для первого оператора необходимо локализовать данные как и в разделе 4.1.2. Для второго оператора в директиве **PARALLEL** необходимо указать спецификацию **REDUCTION**:

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

где:

SUM – имя редукционной операции суммирования,
S – редукционная переменная.

К редукционным операциям относятся: SUM, PRODUCT, AND, OR, MAX, MIN, EQV, NEQV, MAXLOC, MINLOC.

4.3.6 Многомерные массивы

При локализации данных с помощью директивы **ALIGN** для многомерных массивов необходимо указывать индексы выравниваемых элементов массивов по всем измерениям.

На предмет наличия удаленных данных достаточно анализировать только распределенные измерения массивов. Локальные измерения полностью распределены на каждом процессоре и по этим измерениям не существует удаленных данных. Каждое распределенное измерение, по которому существуют удаленные данные, должно быть учтено при спецификации удаленных данных (см. раздел 4.3).

4.4 Определение регионов для выполнения на ускорителях

Вычислительный регион выделяет часть программы (с одним входом и одним выходом) для возможного выполнения на одном или нескольких вычислительных устройствах.

Регион задается парой директив, отмечающих начало и конец региона, и имеет вид

!DVM$ REGION [*список-спецификаций*]

<содержимое региона>

!DVM$ END REGION

Содержимое региона – часть программы, содержащая произвольное количество параллельных циклов, возможно, разделенных последовательными группами операторов. Регион может быть пустым.

В *списке-спецификаций* в директиве **REGION** при необходимости задается информация о направления использования данных в регионе: входные, выходные, локальные.

Спецификации следуют за словом **REGION** и разделяются между собой запятыми. Данные – переменные, массивы, подмассивы задаются после имени спецификации в круглых скобках через запятую.

В качестве спецификаций данных может быть указано:

IN - входные данные: в регионе должны быть самые последние значения этих данных;

OUT - выходные данные: значения указанных переменных в регионе изменяются и могут быть использованы далее;

LOCAL - локальные данные: значения указанных переменных в регионе изменяются, но эти изменения не будут использованы далее;

INOUT - сокращенная запись одновременно двух спецификаций **IN** и **OUT;**

INLOCAL - сокращенная запись одновременно двух спецификаций **IN** и **LOCAL.**

Возможна запись **IN**(A,B) и **IN**(A),**IN**(B).

Секции подмассивов записываются через запятую, например**, IN**(s(1:5,2:6)).

Допускаются составные указания, например, **OUT**(s(1:5)), **OUT** (s(7:10)) или **IN**(s(1:5)), **OUT**(s(6:10)) и пересекающиеся указания, например, **OUT**(s(1:6)), **OUT**(s(3:10)) или даже **OUT** (s(1:6)), **OUT**(s(3:5)).

Не допускаются конфликтующие указания, такие как **OUT**(v), **LOCAL** (v).

Для используемых в регионе, но не указанных в спецификациях переменных действуют следующие правила по умолчанию:

- все используемые массивы считаются используемыми полностью (подмассивы не выделяются);

- всякая переменная, которая используется на чтение, получает атрибут **IN**;
- всякая переменная, которая используется на запись, получает атрибут **INOUT**;
- всякая переменная, направление использования, которой не поддаётся определению, получает атрибут **INOUT**;
- атрибуты **LOCAL** и **OUT** не проставляются.

Если для переменной указано только направление **IN** (не указано **OUT** или **LOCAL**), это означает, что в такую переменную в регионе вообще нет записей и она не меняется в процессе его выполнения.

Пример.

```
!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-массивы распределяются между вычислителями, нераспределенные данные размножаются. Витки параллельных DVMH-циклов внутри региона делятся между вычислителями в соответствии с правилом отображения параллельного цикла, заданного в директиве параллельного цикла.

Каждый оператор последовательной группы операторов выполняется на всех вычислителях, кроме случая модификации в нем распределенных данных - тогда действует правило собственных вычислений.

Ограничения:

- Вложенные регионы не допускаются.
- При выполнении на CUDA-устройстве в регионе не должно быть операций ввода-вывода.
- В регионе не должно быть операторов **ALLOCATE** и **DEALLOCATE**.
- В регионе не должно быть операторов перехода за границу региона.
- Среди последовательной группы операторов не должно быть операторов перехода за границу группы.
- В регионе запрещены операторы условного и безусловного перехода для обхода выполнения параллельных циклов, все параллельные циклы региона должны выполняться.

4.5 Управление перемещением данных между памятью ЦПУ и памятью ускорителей.

Вне вычислительных регионов управление перемещением данных между оперативной памятью ЦПУ и памятями ускорителей задается при помощи директив актуализации - **GET_ACTUAL** и **ACTUAL**.

Директива **GET_ACTUAL** делает все необходимые обновления для того, чтобы в памяти ЦПУ были актуальные (т.е. самые новые) значения данных в

указанных в списке подмассивах и скалярах. В случае отсутствия у директивы параметров все данные в памяти ЦПУ становятся актуальными.

Директива **ACTUAL** объявляет тот факт, что указанные в списке подмассивы и скаляры имеют самые новые значения в памяти ЦПУ. Значения указанных переменных и элементов массивов, находящиеся в памяти ускорителей, считаются устаревшими и перед использованием будут при необходимости обновлены. В случае отсутствия параметров все данные считаются актуальными только в памяти ЦПУ.

Параметры директив – список переменных, массивов и подмассивов - указываются в круглых скобках через запятую.

Использовать директивы **ACTUAL** и **GET_ACTUAL** без параметров не рекомендуется в силу повышения вероятности ошибок (**ACTUAL**), а также опасности излишних перемещений данных (**GET_ACTUAL**).

Пример.

```
!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 Распределенные массивы в COMMON блоках и операторах EQUIVALENCE

Массивы, распределяемые по умолчанию, могут без ограничений использоваться в COMMON блоках и операторах EQUIVALENCE.

Массивы, распределяемые директивами **DISTRIBUTE** и **ALIGN**, не могут использоваться в операторах EQUIVALENCE. Кроме того, эти массивы не могут ассоциироваться с другими объектами данных. Явно распределяемые массивы могут быть компонентами COMMON блока при следующих условиях:

- COMMON блок должен быть описан в главной программной единице.
- Каждое описание COMMON блока должно иметь одно и то же количество компонент, а соответствующие компоненты - последовательности памяти одинакового размера.
- Если компонентой COMMON блока является явно распределяемый массив, то объявления массива в разных программных единицах должны специфицировать

один и тот же тип данных и одинаковую конфигурацию. Директивы **DISTRIBUTE** и **ALIGN** для этого массива должны иметь идентичные параметры.

Пример. Явно распределяемый массив в COMMON блоке.

Описание в главной программе.

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

Описание в подпрограмме. Ошибка: другое количество компонент.

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

Описание в подпрограмме. Ошибка: другое распределение массива.

```
      SUBROUTINE  SUB2
!DVM$ DISTRIBUTE  B2 ( BLOCK, BLOCK )
      COMMON /COM1/  X, Y(12), B2(12,30)
```

Описание в подпрограмме. Ошибка: другая конфигурация массива.

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

Описание в подпрограмме. Нет ошибок.

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

4.7 Процедуры в параллельной программе

Вызов процедуры из параллельного цикла

Процедура, вызываемая из параллельного цикла, не должна иметь побочных эффектов и содержать обменов между процессорами (*прозрачная процедура*). Как следствие этого, прозрачная процедура не содержит:

- операторов ввода-вывода;
- FDVMH-директив;
- присваивание значений переменным COMMON блоков;
- переменных из программной единицы-модуля.

Вызов процедуры вне параллельного цикла

Если фактическим аргументом является явно распределенный массив (**DISTRIBUTE** или **ALIGN**), то он должен передаваться без изменения формы. Это означает, что фактический аргумент является ссылкой на начало массива, а соответствующий формальный аргумент имеет конфигурацию, полностью совпадающую с конфигурацией фактического аргумента.

Формальные аргументы

Если фактический аргумент является распределенным массивом, то соответствующий формальный аргумент должен иметь *явное* или *наследуемое распределение*.

Явное распределение описывается директивами **DISTRIBUTE** и **ALIGN** со следующим ограничением: формальный аргумент может быть выровнен только на другой формальный аргумент. Явное распределение формального аргумента означает, что пользователь должен перед вызовом процедуры обеспечить распределение фактического аргумента в точном соответствии с распределением формального аргумента.

Наследуемое распределение массива C (формального аргумента) описывается директивой:

!DVM$ INHERIT C

Наследуемое распределение означает, что формальный аргумент наследует распределение фактического аргумента при каждом вызове процедуры. Наследуемое распределение не требует от пользователя распределять фактический аргумент в соответствии с формальным аргументом.

Локальные массивы

Локальные массивы могут распределяться в процедуре директивами **DISTRIBUTE** и **ALIGN**. Локальный массив может быть выровнен на формальный аргумент. Директива **DISTRIBUTE** распределяет локальный массив на ту подсистему процессоров, на которой была вызвана процедура (*текущая подсистема)*.

Для распределенного локального массива с атрибутом SAVE директивы **DISTRIBUTE** или **ALIGN** имеют идентичные параметры при каждом вызове процедуры.

Пример. Распределение локальных массивов и формальных аргументов.

```
      SUBROUTINE  DIST( A, B, C, N )
      DIMENSION  A(N,N), B(N,N), C(N,N), X(N,N), Y(N,N)
C     явное распределение формального аргумента
!DVM$ DISTRIBUTE  A ( *, BLOCK )
C     выравниваемый формальный аргумент
!DVM$ ALIGN  B( I, J )  WITH  A( I, J )
C     наследуемое распределение формального аргумента
!DVM$ INHERIT  C
C     выравнивание локального массива на формальный аргумент
!DVM$ ALIGN  X( I, J )  WITH  C( I, J )
C      распределение локального массива
!DVM$ DISTRIBUTE  Y ( *, BLOCK )
      . . .
      END
```

4.8 Ввод-вывод

Для организации ввода-вывода данных в программе на языке FDVMH используются операторы стандарта Фортран 77.

FDVMH допускает только ограниченную форму операторов ввода-вывода для распределенных массивов:

- Список ввода-вывода должен состоять только из одного имени распределенного массива и не может содержать других объектов ввода-вывода.
- В операторах ввода-вывода по формату допускается только формат, задаваемый *.
- Список управляющей информации не должен содержать параметры ERR, END и IOSTAT.
- В списке управляющей информации допускается использование только размноженных переменных.

Не разрешается использовать операторы ввода-вывода распределенных массивов в параллельном цикле.

На операторы ввода-вывода размноженных данных распространяются следующие ограничения:

- Список управляющей информации не должен содержать параметры ERR и END.
- Допускается лишь следующая упрощенная форма списка с неявным циклом: (A(i1,i2,...,I), I = n1,n2)
  при вводе размноженного массива неопределенного размера.

Оператор ввода, оператор INQUIRE, а также любой другой оператор ввода-вывода с управляющим параметром IOSTAT не должны использоваться в параллельном цикле.

Программа на языке FDVMH, выполняющая бесформатный ввод-вывод распределенных массивов, в общем случае не совместима с последовательной программой на Фортране 77. Данные, записанные одной программой, не могут быть прочитаны другой, вследствие разницы длин записей.

5 Схема выполнения DVMH-программы

Выполнение DVMH-программы можно представить, как выполнение последовательности вычислительных регионов и участков между ними, которые будем называть внерегионным пространством. Код во внерегионном пространстве выполняется на центральном процессоре, тогда как вычислительные регионы могут выполняться на разнородных вычислительных устройствах. Как внутри, так и вне регионов могут быть как параллельные циклы, так и последовательные участки программы.

Выполнение DVMH-программы начинается синхронно всеми запущенными процессами. На каждый процесс выделяется одна основная последовательная нить выполнения для выполнения последовательных участков программы.

При входе в вычислительный регион каждый процесс независимо выполняет дополнительное распределение данных, используемых этим вычислительным регионом, по вычислительным устройствам. На этом этапе производится динамическое планирование с целью балансировки нагрузки и минимизации временных затрат на перемещения данных, связанных с перераспределением данных.

6 Компиляция, выполнение и отладка DVMH-программ

6.1 Что такое DVMH-программа?

Параллельная программа представляет собой обычную последовательную программу, в которую вставлены DVMH-директивы, определяющие ее параллельное выполнение.

DVMH-директивы записываются в виде специального комментария

!DVM$ < DVMH-директива > ,

который в последовательной программе воспринимается как комментарий.

DVMH-программа – это один или несколько файлов с исходными текстами на языке FDVMH, имеющих расширение **fdv, f, for**, **f90, f95, f03.** Если файлы имеют расширение **f90, f95, f03,** то считается, что они в свободной форме, иначе при помощи опции компиляции **(–FI**) необходимо указать, что форма записи является фиксированной. При компиляции нужно указать опцию **-f90**, если файлы имеют расширение **fdv, f, for,** но записаны в свободной форме.

Если среди файлов есть программная единица-модуль, то файлы необходимо объединить в единую программу при помощи INCLUDE.

Если файлов много, но модулей нет, то для объединения файлов можно использовать **makefile** или перечислить все файлы в строке компиляции.

6.2 Настройка DVM-системы

Для компиляции и запуска на выполнение DVMH-программы в рабочую директорию, в которой она находится, необходимо скопировать файл запуска dvm-команд (**dvm**) из директории **dvm_sys/user** DVM-системы.

В этом файле определены переменные окружения, которые могут быть изменены пользователем. В **Приложении 3** приведено описание переменных окружения для DVMH-программ.

Инструкции по настройке переменных окружения, компиляции, запуску программ на разных платформах доступны на сайте DVM.

6.3 Методика отладки DVMH-программ

Под отладкой DVM-программ подразумеваются два различных вида деятельности:

- функциональная отладка, целью которой является достижение правильности функционального выполнения программы;
- отладка эффективности, целью которой является достижение требуемого уровня эффективности параллельного выполнения программы.

Отладку DVMH-программ рекомендуется проводить на тестовых данных в следующей последовательности шагов:

1. Последовательное выполнение и отладка при помощи стандартного компилятора с языка Fortran и стандартных средств отладки.
2. Функциональная отладка параллельной программы.
3. Отладка эффективности параллельной программы.

6.3.1 Последовательное выполнение и отладка при помощи стандартного компилятора с языка Fortran и стандартных средств отладки

DVMH-директивы являются комментариями языка Fortran для стандартных компиляторов, поэтому DVMH-программа обрабатывается ими как обычная последовательная программа Это позволяет отлаживать ее как обычную последовательную программу (в режиме *игнорирования* DVMH-директив) с использованием привычных средств отладки.

6.3.2 Функциональная отладка параллельной программы

Функциональная отладка DVMH-программы осуществляется в следующей последовательности шагов:

1. Компиляция. Получение готовой программы.
2. Динамический контроль DVMH-директив.
3. Сравнение результатов выполнения на кластере без ускорителей.
4. Сравнение результатов выполнения в регионе на ЦПУ и ускорителях.

6.3.2.1 Компиляция. Получение готовой программы.

Команда конвертации и компиляции DVMH-программы имеет вид:

dvm f <имя DVMH-программы>

где:

dvm – префикс (имя файла запуска DVMH-команд);

<имя DVMH-программы> – имя файла с исходным текстом программы с расширеним. Поиск файла производится только в текущей директории;

Результат работы: выполняемый файл **<имя DVMH-программы>** в текущей директории. Если конвертор обнаружил ошибки, выполняемый файл не создается.

6.3.2.2 Динамический контроль DVMH-директив

Динамический контроль позволяет выявлять ошибки следующих типов:

1. Необъявленная зависимость по данным в параллельном цикле.
2. Использование в параллельном цикле или после выхода из него приватных переменных без их предварительной инициализации.
3. Запись в переменные, доступные только на чтение.
4. Использование редукционных переменных после запуска асинхронной редукции, но до ее завершения.
5. Необъявленный доступ к нелокальным элементам распределенного массива.
6. Запись в теневые грани массива.
7. Чтение теневых элементов массива до завершения операции их обновления.
8. Модификация нелокального элемента распределенного массива в последовательной части программы.
9. Выход за пределы распределенного массива.
10. Запись в буфер удаленного доступа.

Для динамического контроля программы ее следует сначала скомпилировать в режиме получения *отладочного варианта параллельной программы*

Команда получения отладочного параллельного варианта имеет вид:

dvm fpdeb <имя DVMH-программы>

Результат работы: выполняемый файл **<имя DVMH-программы>_p.**

Команда запуска отладочного параллельного варианта DVMH-программ, осуществляющего динамический контроль DVMH-указаний имеет вид:

dvm err <имя DVMH-программы>

Результат работы: ошибки, обнаруженные в DVMH-указаниях (при их наличии).

В случае обнаружения ошибок в текущей директории появляется файл **error.dbg**, в котором перечислены все найденные ошибки. Краткое сообщение о наличии ошибок появляется после выдачи результатов выполнения задачи.

Структуру и перечень сообщений об ошибках динамического контроля см. в **Приложении 4.**

Отсутствие ошибок при динамическом контроле не гарантирует правильной работы параллельной программы по следующим причинам:

- динамический контроль не проверяет правильность описания редукционных операций;
- источником ошибок могут быть процедуры, вызываемые из DVMH-программ, но написанные на других языках и не подлежащие динамическому контролю;
- динамический контроль не проверяет корректность выполнения региона на ускорителях, а также отсутствия спецификаций **OUT** или **LOCAL** для переменных, которые изменяются в регионе;

- отлаженная последовательная программа может содержать ошибки, которые не проявились при ее последовательном выполнении, но могут проявиться при параллельном выполнении.

Поэтому отладку программы следует продолжить.

6.3.2.3 Сравнение результатов выполнения на кластере без ускорителей

Для поиска таких ошибок используется метод накопления и сравнения трассировок последовательного и параллельного выполнения программы, который позволяет определить место в программе и момент, когда появляются расхождения в результатах вычислений.

Программа выполняется на ЦПУ, ускорители не используются.

При трассировке выполняется сбор информации обо всех чтениях и модификациях переменных, о начале выполнения каждого витка цикла, о начале и конце выполнения параллельного цикла.

Для выполнения сравнения необходимо выполнить следующую последовательность команд.

1. Команда получения отладочного параллельного варианта:

dvm fpdeb <имя DVMH-программы>

Результат работы: выполняемый файл **<имя DVMH-программы>_p**.

2. Команда получения отладочного последовательного варианта:

dvm fsdeb <имя DVMH-программы>

Результат работы: выполняемый файл **<имя DVMH-программы>_s**.

3. Команда запуска отладочного последовательного варианта DVM-программ, осуществляющая накопление эталонной трассировки на одном процессоре:

dvm trc <имя DVMH-программы>

Результат работы: файл с накопленной трассировкой **0.trd**. При наличии ошибок сбора трассировки – сообщение об ошибках после выдачи результатов выполнения задачи и появление файла **error.trd** в текущей директории.

4. Команда запуска отладочного параллельного варианта DVMH-программ, осуществляющая сравнение результатов вычислений, полученных при запуске программы на одном процессоре в специальном режиме проверки редукционных операций, с накопленной ранее эталонной трассировкой.

dvm red <имя DVMH-программы>

Результат работы: При наличии ошибок – сообщение об ошибках после выдачи результатов выполнения задачи и появление файла **error.trd** в текущей директории.

5. Команда запуска отладочного параллельного варианта DVM-программы, осуществляющая сравнение результатов вычислений, полученных при запуске

программы на нескольких процессорах, с накопленной ранее эталонной трассировкой.

dvm dif [**N1** [**N2** [**N3 [N4]**]]] **<имя DVMH-программы>**

где **N1, N2, N3, N4** – размеры матрицы процессоров (по умолчанию – 1 1 1 1).
Результат работы: При наличии ошибок – сообщение об ошибках после выдачи результатов выполнения задачи и появление файла **error.trd** в текущей директории.

Если различий в трассировке не обнаружено, можно переходить к параллельному выполнению программы с реальными данными.

6. Если обнаружены различия, но ошибку в программе не удается определить по эталонной трассировке и диагностике сравнения трассировок, пользователь может накопить трассировку на каждом процессоре при запуске отладочного параллельного варианта программы на требуемой матрице процессоров. Для этого используется следующая команда.

dvm ptrc [**N1** [**N2** [**N3 [N4]**]]] **<имя DVMH-программы>**

где **N1, N2, N3, N4** – размеры матрицы процессоров (по умолчанию – 1 1 1 1).

Результат работы: Для каждого процессора накапливается трассировка в отдельном файле с именами: **0.trd, 1.trd, 2.trd** и т.д. При наличии ошибок выдается соответствующее сообщение после выдачи результатов выполнения задачи.

Структуру накапливаемых файлов трассировки и перечень сообщений об ошибках сравнения результатов см. в **Приложении 4.**

Замечание. Все шаги отладки, описанные в разделах 6.3.2.2 и 6.3.2.3, можно запустить одной командой:

dvm ftest [**N1** [**N2** [**N3 [N4]**]]] **<имя DVMH-программы>**

6.3.2.4 Сравнение результатов выполнения в регионах на ЦПУ и ускорителях

Сравнение результатов выполнения в регионах на ЦПУ и ускорителях - это специальный режим работы DVMH-программы, при котором все вычисления в регионах одновременно выполняются на ЦПУ и ГПУ.

В данном режиме с заданной степенью точности сравниваются выходные данные, полученные в регионе при выполнении на ГПУ, с данными, полученными в регионе при выполнении на ЦПУ.

Такой механизм позволяет выявлять и локализовать ошибки, проявляющиеся при работе на ускорителях.

При включенном режиме сравнительной отладки региона одни и те же витки одного и того же параллельного цикла будут выполнены дважды – один раз на ЦПУ, другой – на ГПУ.

В сравнение включаются все выходные данные вычислительного региона. При этом целочисленные данные сравниваются на совпадение, а вещественные числа

сравниваются с заданной точностью по абсолютной и относительной погрешности. В случае нахождения расхождений выдается информация о найденных расхождениях. Далее в программе используется та версия данных, которая была получена при счете на ЦПУ.

Такое сравнение используется для контроля корректности выполнения региона на ускорителях, а также отсутствия спецификаций **OUT** или **LOCAL** для переменных, которые изменяются в регионе.

Ошибки при выполнении региона на ускорителе могут возникать по нескольким причинам:

1. Программистом произведено некорректное распараллеливание, не подходящее для массивно-параллельного выполнения в общей памяти.
2. Программист некорректно указал приватные или редукционные переменные в параллельном цикле.
3. Арифметические операции или математические функции на ускорителе отработали с иным по сравнению с работой на ЦПУ результатом. Это может происходить из-за различий в системе команд, приводящих к различным результатам (в пределах точности округлений).
4. Программист указал неверные директивы актуализации данных **GET_ACTUAL** и **ACTUAL**, вследствие чего обрабатываемые данные на ЦПУ и ускорителе оказались разными.

Включение и использование режима сравнительной отладки не требует от программиста вносить какие-либо изменения в программу, а также заново ее компилировать.

Программа будет выполняться в режиме сравнительной отладки если для запуска на выполнение использовать команду:

dvm cmph [N1 [N2 [N3 [N4]]]] < имя выполняемого файла DVMH-программы >

где **N1, N2, N3, N4** – размеры матрицы процессоров (по умолчанию – 1 1 1 1).

В случае обнаружения ошибок информация о них выдается в стандартный поток вывода ошибок или в файл. Имя этого файла можно указать в переменнной окружения **DVMH_LOGFILE.**

Точность сравнения переменных можно изменить, указав значения переменных окружения **DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS, DVMH_COMPARE_LONGDOUBLES_EPS**.

Сравнительную отладку целесообразно проводить на тестовых данных.

6.3.3 Компиляция и выполнение DVMH-программ на кластере с ускорителями

Компиляция и выполнение DVMH-программ осуществляется при помощи следующих команд:

dvm f <опции компиляции> <имя DVMH-программы>

dvm run [N1 [N2 [N3[N4]]]] < имя выполняемого файла DVMH-программы >

где **N1, N2, N3, N4** – размеры матрицы процессоров (по умолчанию – 1 1 1 1).

При запуске DVMH-программ размерность и размеры матрицы виртуальных процессоров определяют конфигурацию и число процессов (**N1*N2*N3*N4**), на которых DVMH-программа будет выполняться параллельно.

Опции компиляции описаны в **Приложении 3**.

Инструкции по настройке переменных окружения, компиляции, запуску программ на разных платформах доступна на сайте DVM.

6.3.4 Отладка эффективности параллельной программы

Для отладки эффективности используются анализатор производительности**,** который позволяет получить информацию об основных характеристиках эффективности выполнения программы (или ее частей) на параллельной системе. Эффективность выполнения параллельных программ на многопроцессорных ЭВМ с распределенной памятью определяется следующими основными факторами:

- степенью распараллеливания программы - долей параллельных вычислений в общем объеме вычислений;
- равномерностью загрузки процессоров во время выполнения параллельных вычислений;
- временем, необходимым для выполнения межпроцессорных обменов;
- степенью совмещения межпроцессорных обменов с вычислениями.

6.3.4.1 Основные характеристики производительности

Основные характеристики и их компоненты

- **Коэффициент эффективности** (**Parallelization efficiency**) равен отношению полезного времени к общему времени использования процессоров.
- **Время выполнения** (**Execution time**) - максимальное значение среди времен выполнения программы на всех используемых ею процессорах.
- **Число используемых процессоров** (**Processors**).
- **Общее время использования процессоров** (**Total time**) - произведение времени выполнения (**Execution time**) на число используемых процессоров (**Processors**).
- **Полезное время** (**Productive time**) - сумма трех составляющих – полезного процессорного времени (**CPU**), времени ввода-вывода (**I/O**) и полезного системного времени (**Sys**).
- **Потерянное время** (**Lost_time**) - разница между общим временем использования процессоров и полезным временем. Недостаточный параллелизм, коммуникации и простои - составляющие потерянного времени.
- **Недостаточный параллелизм** (**Insufficient_par**) и его компоненты.
- **Коммуникации** (**Communication**) и все их компоненты.
- **Простои** (**Idle time**) процессора из-за его недостаточной загрузки.

- **Потенциальные потери из-за разбалансировки** (**Load_Imbalance**).
- **Потенциальные потери из-за синхронизации** (**Synchronization**) при выполнении коллективных операций и все их компоненты.
- **Потенциальные потери из-за разброса времен** (**Time_variation**) выполнения коллективных операций и все их компоненты.
- **Время перекрытия** (**Overlap**) и его компоненты. Эта характеристика отражает потенциальное сокращение коммуникационных расходов за счет совмещения межпроцессорных обменов с вычислениями.

Характеристики выполнения программы на каждом процессоре

- Потерянное время (**Lost time**) - сумма его составляющих – потерь из-за недостаточного параллелизма (**User insufficient_par**), системных потерь из-за недостаточного параллелизма (**Sys insufficient_par**), коммуникаций (**Communication**) и простоев (**Idle time**).
- Потери из-за недостаточного параллелизма (**User insufficient_par**).
- Системные потери из-за недостаточного параллелизма (**Sys insufficient_par**).
- Простои на данном процессоре (**Idle time**) - разность между максимальным временем выполнения интервала (на каком-то процессоре) и временем его выполнения на данном процессоре.
- Общее время коммуникаций (**Communication**).
- Реальные потери из-за рассинхронизации (**Real synchronization**).
- Потенциальные потери из-за рассинхронизации (**Synchronization**).
- Потенциальные потери из-за разброса времен (**Variation**).
- Время перекрытия асинхронных операций (**Overlap**).
- Разбалансировка (**Load_imbalance**) вычисляется как разность между максимальным процессорным временем (**CPU+Sys**) и соответствующим временем на данном процессоре.
- Время выполнения интервала (**Execution_time**).
- Полезное процессорное время (**User CPU_time**).
- Полезное системное время (**Sys CPU_time**).
- Время ввода-вывода (**I/O_time**).
- Число используемых процессоров для данного интервала (**Processors**).
- Времена коммуникаций для всех типов коллективных операций (**Reduction, Shadow, Remote_access, Redistribution** и **I/O**).
- Реальные потери из-за рассинхронизации для всех типов коллективных операций.
- Потенциальные потери из-за рассинхронизации для всех типов коллективных операций.
- Потенциальные потери из-за разброса времен для всех типов коллективных операций.
- Время перекрытия для всех типов коллективных операций.

Замечание 1. Последние три характеристики выдаются только в том случае, если в параметрах запуска задан параметр **IsTimeVariation=1;**

Для получения величины реальных потери из-за рассинхронизации необходимо задать параметр **IsSynchrTime=1**.

Замечание 2. Анализатор производительности выдает пользователю характеристики выполнения как для всей программы, так и на каждом процессоре.

6.3.4.2 Представление программы в виде иерархии интервалов. Выполнение со сбором статистики.

Выполнение всей программы целиком рассматривается как интервал самого высокого (нулевого) уровня. Этот интервал может включать в себя несколько интервалов следующего (первого) уровня. Такими интервалами могут быть параллельные циклы, последовательные циклы, а также любые отмеченные программистом последовательности операторов, выполнение которых всегда начинается с выполнения первого оператора, а заканчивается выполнением последнего. Интервалы первого уровня могут в свою очередь включать в себя интервалы второго уровня, и т.д.

Все описанные выше характеристики вычисляются не только для всей программы, но и для каждого ее интервала. В языке Fortran DVMH интервал задается следующим образом:

!DVM$ INTERVAL [<целочисленное-выражение>]

<последовательность-операторов>

!DVM$ END INTERVAL

Выделив, например, тело цикла как интервал и задав в качестве целочисленного выражения индексную переменную цикла, можно оформить каждый виток цикла отдельным интервалом. Подобным образом можно получить характеристики четных и нечетных витков цикла, либо характеристики выполнения процедуры при заданных значениях ее параметров.

Для сбора информации о производительности DVMH-программы при ее запуске на многопроцессорной ЭВМ или сети рабочих станций параметр **Is_DVM_STAT** (признак сбора статистики в файле **usr.par**) должен быть равен 1.

После окончания выполнения со сбором статистики в текущей директории должен создаться файл статистики с именем **sts.gz+** (или просто **sts,** или **<имя задачи>.sts.gz+**). Если при сборе данных были обнаружены ошибки, то файл все равно может создаться, а сообщение об ошибке будет выведено на экран или в файл. Список сообщений приведен в **Приложении 5.**

Ограничение:

- Интервал не может располагаться внутри цикла, он должен включать цикл целиком.

6.3.4.3 Запуск анализатора производительности. Представление характеристик задачи по интервалам.

Для получения временных характеристик по интервалам следует выполнить команду:

dvm pa <имя файла статистики> < имя файла с характеристиками>

Все характеристики записываются в текстовом виде в указанный файл. Для каждого интервала выдается следующая информация:

- имя файла с исходным текстом DVMH-программы и номер первого оператора интервала в нем (SOURCE, LINE);
- тип интервала – вся программа, параллельный цикл (PAR), последовательный цикл (SEQ) или выделенная пользователем последовательность операторов (USER);
- номер уровня вложенности (LEVEL);
- количество входов (и выходов) в интервал (EXE_COUNT);
- значение выражения, заданного при описании интервала средствами языка (EXPR);
- основные характеристики выполнения и их компоненты (Main characteristics);
- минимальные, максимальные и средние значения характеристик выполнения программы на каждом процессоре (Comparative characteristics);
- характеристики выполнения программы на каждом процессоре (Execution characteristics on processors).

При выдаче характеристик их компоненты располагаются в той же строке (справа в скобках), либо в следующей строке (справа от символов “*” или “-”).

Компоненты некоторых характеристик, связанных с выполнением коллективных операций, выдаются в виде столбцов таблицы, где строки соответствуют типу коллективной операции, а столбцы - характеристикам. Один из столбцов (Nop) этой таблицы содержит количества операций каждого типа, которые являются характеристиками, не зависящими от числа процессоров, используемых для выполнения программы.

Информация о минимальных, максимальных и средних значениях таких характеристик оформлена в таблицу аналогичным образом. Некоторые характеристики вообще не выдаются в том случае, если их значения равны нулю.

6.3.4.4 Рекомендации по анализу характеристик

Главным критерием является **коэффициент эффективности** распараллеливания. Если коэффициент эффективности невысокий, то необходимо анализировать потерянное время и его компоненты.

Сначала следует оценить три компоненты потерянного времени для основного интервала (как правило, в качестве такого интервала выделяется итерационный цикл в программе). Наиболее вероятно, что основная доля потерянного времени приходится на одну из первых двух компонент (недостаточный параллелизм или коммуникации).

В случае если причиной оказался **недостаточный параллелизм**, необходимо уточнить, на каких участках он обнаружен – последовательных или параллельных. В последнем случае причина может быть очень простой – неверное задание матрицы процессоров при запуске программы или неверное распределение данных и вычислений. Если же недостаточный параллелизм обнаружен на последовательных участках, то причиной этого, скорее всего, является наличие последовательного

цикла, выполняющего большой объем вычислений. Устранение этой причины может потребовать больших усилий.

В том же случае, если основной причиной потерь являются **коммуникации,** необходимо, прежде всего, обратить внимание на **реальные потери из-за рассинхронизации (Real synchronization)**. Если ее значение близко к размерам потерь из-за коммуникаций, то необходимо рассмотреть **потенциальные потери из-за разбалансировки (Load_Imbalance)**, поскольку именно разбалансировка вычислений в параллельном цикле является наиболее вероятной причиной рассинхронизации и больших потерь на коммуникациях. Если величина разбалансировки намного меньше величины **потенциальных потерь из-за синхронизации (Synchronization)** , то необходимо обратить внимание на величину **потенциальных потерь из-за разброса времен (Time_variation)** коллективных операций. Если рассинхронизация не является следствием разброса времен завершения коллективных операций, то ее возможной причиной могут быть разбалансировки некоторых параллельных циклов, которые на рассматриваемом интервале выполнения программы могли взаимно компенсироваться. Поэтому имеет смысл перейти к рассмотрению характеристик разбалансировки на интервалах более низкого уровня.

Второй вероятной причиной больших потерь из-за рассинхронизации может быть рассинхронизация процессоров, которая возникает при выдаче операций ввода-вывода. Это происходит из-за того, что основная работа (обращение к функциям ввода-вывода операционной системы) производится на процессоре ввода-вывода, а остальные процессоры в это время ждут получения от него данных или информации о завершении коллективной операции. Эту причину потерь легко обнаружить, обратив внимание на соответствующую компоненту характеристики **коммуникации** – потери из-за коммуникаций при вводе-выводе.

Основной причиной потерь из-за коммуникаций может быть и просто большое количество операций редукции или загрузки требуемых данных с других процессоров (обновление теневых граней или удаленный доступ). В этом случае необходимо проверить спецификации удаленных данных. Наличие лишних спецификаций – одна из причин потерь из-за коммуникаций.

Возможен и другой подход к анализу характеристик, когда сначала анализируются коэффициенты эффективности и потерянное время на различных интервалах первого уровня, затем второго уровня, и т.д. В результате определяется критический участок программы. При этом необходимо иметь в виду, что причиной потерь на данном интервале из-за рассинхронизации и простоев могут быть разбалансировки и разбросы времен не только на этом интервале, но и на других, выполнявшихся до него интервалах.

Замечание. Поскольку при переходе от последовательного выполнения программы к ее параллельному выполнению на одном процессоре возможны потери эффективности, то рекомендуется скомпилировать последовательную программу с вызовами функций сбора информации для оценки производительности и пропустить полученную последовательную программу на одном процессоре. Для этого необходимо выполнить команду компиляции:

dvm f -s <имя DVMH-программы>

и команду запуска программы:

dvm run 1 <имя DVMH-программы>

Далее необходимо сравнить полученную статистику со статистикой параллельного выполнения на одном процессоре.

7 Литература

1. DVM-система [Электронный ресурс] - : [web-сайт] – DVM
   http://dvm-system.org/

8 Пример программы Якоби на языке Fortran-DVMH

Проиллюстрируем возможности языка Fortan-DVMH на примере программы для алгоритма Якоби.

```
            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
```

В результате выполнения директивы

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

массив A будет распределен между вычислителями. Количество и тип используемых вычислителей задается при запуске программы с помощью переменных окружения и параметров командной строки.

Директива

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

задает совместное распределение двух массивов A и B. Элементы массива B будут распределены на тот же вычислитель, где будут размещены соответствующие элементы массива A.

Директива

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

задает распределение вычислений. Витки цикла будут выполняться на том вычислителе, где распределены соответствующие элементы массива A.

Спецификация **REDUCTION (MAX(EPS))** организует эффективное выполнение редукционной операции - глобальной операции с расположенными на различных вычислителях данных (нахождение максимального значения).

Спецификация **SHADOW_RENEW (A)** указывает на необходимость подкачки удаленных данных (теневых граней) с другихвычислителей перед выполнением цикла. Поскольку никакие дополнительные спецификации в директивах **REGION** не заданы, компилятор определяет направления использования переменных автоматически - **INOUT**(A,B,EPS).

При выполнении первого вычислительного региона (цикла инициализации) для распределенных частей массивов A и B на ускорителях будет выделена необходимая пямять.

При входе во второй вычислительный регион (в итерационном цикле) осуществляется проверка, присутствуют ли актуальные представители для массивов A и B на вычислителе. Поскольку такие представители уже присутствуют, то никакие дополнительные операции копирования актуальных данных на вычислители не выполняются.

При выходе из вычислительного региона обновление данных в памяти хоста не производится. Перед выводом массива в файл, требуется скопировать последние изменения массива из памяти вычислителя при помощи директивы **GET_ACTUAL**(B).

Приложение 1. Синтаксис директив FDVMH

Синтаксис директив FDVMH описывается следующей БНФ формой:

is	по определению
or	альтернатива
[ ]	необязательная конструкция
[ ]…	повторение конструкции 0 или более раз
x-list	x [, x]…

Синтаксис директивы.

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*

Ограничения:

- Директивы спецификации должны находиться в разделе спецификаций.
- Исполняемые директивы должны находиться среди исполняемых операторов.
- Любое выражение, входящее в директиву спецификации, должно быть выражением спецификации.
  Выражение спецификации – это выражение, в котором каждое первичное должно быть одной из следующих форм:
  1) константа,
  2) переменная, которая является формальным аргументом,
  3) переменная из COMMON блока,
  4) ссылка на встроенную функцию, где каждый аргумент является выражением спецификации,
  5) выражение спецификации, заключенное в скобки.

Никакой оператор не может находиться среди строк продолжения директивы. Строка *directive-line* не может находиться внутри продолженного оператора. Ниже приводится пример директивы с продолжением в фиксированной форме. Отметим, что позиция 6 должна быть пробелом за исключением случая, когда она используется для обозначения продолжения.

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

Пример директивы с продолжением в свободной форме:

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

Следующий пример демонстрирует универсальную директиву с продолжением, т.е. удовлетворяющую правилам и свободной, и фиксированной формы:

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

Заметим, что знак **&** в первой строке стоит в 73 позиции.

Директивы DISTRIBUTE и REDISTRIBUTE

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*

distributee **is** *array-name*
or *template-name*

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

Ограничения:

- Длина списка *dist-format-list* должна быть равна количеству измерений массива. Т.е. для каждого измерения должен быть задан формат распределения.

Директивы ALIGN и REALIGN

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 -

Ограничение:

- Длина списка *align-source-list* должна быть равна количеству измерений выравниваемого массива.

Директива TEMPLATE

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

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

Распределение витков цикла. Директива PARALLEL

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*]

Приватные переменные. Спецификация PRIVATE

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

private_variable **is** *array-name*

or *scalar*

Редукционные операции и переменные. Спецификация REDUCTION

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**

Ограничения:

- Редукционными переменными не могут быть распределенные массивы.
- Редукционные переменные вычисляются и используются только в операторах определенного вида - редукционных операторах.

Спецификация массива с теневыми гранями

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*

Ограничения:

- Размер левой теневой грани (*low-width*) и размер правой теневой грани (*high-width*) должны быть целыми константными выражениями, значения которых больше или равны 0.
- Задание размера теневых граней как *width* эквивалентно заданию *width : width.*
- По умолчанию, распределенный массив имеет теневые грани шириной 1 с обеих сторон каждого распределенного измерения.

Спецификация SHADOW_RENEW

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

Ограничения:

- Размер теневых граней, заполняемых значениями, не должен превышать максимального размера, описанного в директиве **SHADOW**.
- Если размеры теневых граней не указаны, то используются максимальные размеры.

Спецификация ACROSS

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*

Ограничение:

- В каждой ссылке на массив может существовать зависимость по данным только по одному распределенному измерению.

Директива REMOTE_ACCESS

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*

Директива REGION

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**

Директивы GET_ACTUAL и ACTUAL

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*

Директива INHERIT

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

Приложение 2. Переменные окружения для DVMH-программ.

В файле запуска dvm-команд определены переменные окружения, которые могут быть изменены пользователем.

DVMH_NUM_CUDAS - количество CUDA-устройств для использования одним процессом. Если переменная не задана, то ее оптимальное значение определяет система поддержки. Система поддержки обеспечивает эффективное использование всех ресурсов узла. Оптимальное значение зависит от количества устройств на узле и количества запущенных процессов на узле. Значение переменной **DVMH_NUM_CUDAS** не может превышать физически доступное количество ускорителей.

DVMH_NUM_THREADS - количество нитей, работающих на ЦПУ. Если переменная не задана, то ее оптимальное значение определяет система поддержки. Система подержки обеспечивает эффективное использование всех ресурсов узла. Оптимальное значение зависит от количества устройств на узле, количества запущенных процессов на узле и количества CUDA-устройств для использования одним процессом. (**DVMH_NUM_CUDAS)**. Значение переменной **DVMH_NUM_THREADS** может быть любым положительным числом.

DVMH_LOGFILE - имя лог-файла. Имя файла задается в кавычках, можно использовать конструкцию %d (например,’dvmh_%d.log’), в этом случае лог каждого MPI-процесса будет в отдельном файле. Если переменная не задана, то используется стандартный поток вывода ошибок.

DVMH_LOGLEVEL - уровень детализации лог-файла. Задается в виде целого десятичного числа. Имеются следующие уровни:

0 – выдаются ошибки уровня fatal err,
1 - err,
2 - warning,
3 - info,
4 - debug,
5 - trace.

Указание меньше нуля приравнивается к нулю. Указание больше 5 приравнивается к 5. Если переменная не задана, то устанавливается уровень 1 - err.

DVMH_PPN - количество процессов на узел. Определяет распределение вычислительных ресурсов по процессам — каждому даётся одинаковая доля от вычислительных ресурсов узла. Если переменная не задана, то приравнивается к единице (каждый процесс использует все ресурсы узла).

DVMH_COMPARE_FLOATS_EPS - точность сравнения переменных с плавающей точкой одинарной точности по относительной и абсолютной погрешности при сравнительной отладке. По умолчанию равна FLT_EPSILON*1000, где FLT_EPSILON – минимальное положительное x такое, что 1.0+x=1.0.

DVMH_COMPARE_DOUBLES_EPS - точность сравнения переменных с плавающей точкой двойной точности по относительной и абсолютной погрешности при сравнительной отладке. По умолчанию равна DBL_EPSILON*10000, где DBL_EPSILON – минимальное положительное x такое, что 1.0+x=1.0..

DVMH_COMPARE_LONGDOUBLES_EPS - точность сравнения переменных с плавающей точкой длинной двойной точности по относительной и абсолютной погрешности при сравнительной отладке. По умолчанию LDBL_EPSILON*100000, где LDBL_EPSILON – минимальное положительное x такое, что 1.0+x=1.0.

DVMH_CPU_PERF - относительная производительность ЦПУ (суммарная всех ядер ЦПУ), используется для разделения работы в режиме планирования 1. По умолчанию равна 1.

DVMH_CUDAS_PERF - относительная производительность ГПУ, задается закольцованным списком вещественных чисел через пробел или запятую. По умолчанию равна 1.

Приложение 3. Опции компиляции для DVMH-программ.

-noH	-	режим игнорирования DVMH-директив
-f90,-FR	-	указание на свободную форму записи исходных текстов на Фортране и директив FDVMH. Если файлы имеют расширение **f90, f95, f03,** то считается, что они в свободной форме.
-FI	-	указание на фиксированную форму записи исходных текстов на Фортране и директив FDVMH
-autoTfm	-	режим динамического переупорядочивания массива (режим перестановки измерений массива для оптимизации доступа к памяти ГПУ)
-Opl	-	Параллельные циклы вне регионов выполняются на хост-процессоре также, как и циклы внутри регионов. В этом режиме нужны описания приватностей для циклов вне регионов.

-gpuO1	-	оптимизация использования приватных переменных для часто используемых элементов массива, которые отображаются компилятором на регистры.
-noCuda	-	управление процессом компиляции – компилятор не готовит выполнение регионов на CUDA-устройствах .
-collapse<N>	-	режим компиляции, при котором для каждого параллельного цикла при выполнении на ЦПУ в OpenMP директиву добавляется спецификация COLLAPSE(N)
-mmic	-	компиляция программы для выполнения на сопроцессоре Intel Xeon Phi

При компиляции DVMH-программы с опцией –noH DVMH-директивы игнорируются, и DVMH-программа превращается в DVM-программу.

Использование опций оптимизации (–autoTfm, -Opl, –gpuO1) может способствовать повышению производительности программы.

Опция оптимизации -collapse<N> может оптимизимировать выполнение программы на Xeon Phi.

Приложение 4. Диагностические сообщения DVM- отладчика.

Общая структура сообщения об ошибке DVM- отладчика:

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

где:

<process number>	–	номер процессора, на котором произошла ошибка. Выводится, только при запуске программы на нескольких процессорах.
<context>	–	контекст, в котором произошла ошибка. Может иметь одну из следующих форм:
	sequential branch –	ошибка произошла в последовательной части программы;
	Loop(No(N_1), Iter(I_1,I_2,…)), …, Loop(No(N_m), Iter(I_1,I_2,…)) –	ошибка произошла в ходе выполнения цикла **m**-степени вложенности.
<file>	–	имя файла, где произошла ошибка.
<line>	–	номер строки.
<count>	–	число повторений данной ошибки в данном контексте. Выводится при суммарной выдаче всех найденных ошибок.
<error message>	–	Сообщение о произошедшей ошибке.

1. Динамический контроль

Сообщение	Описание
Writing to read-only variable<var>	Обнаружена запись в переменную, которая доступна только для чтения.
Using non-initialized private variable <var>	Обнаружено обращение к неинициализированной переменной.
Using non-initialized element <elem>	Обнаружено обращение к неинициализированному элементу распределенного массива.
Using variable <var> before asynchronous reduction competed	Обращение к редукционной переменной до завершения операции редукции.
Access to non-local element <elem>	Обращение к нелокальному элементу массива.
Writing to shadow element <elem> of array	Запись в теневой элемент массива.
Shadow element <elem> was not updated	Обращение к теневым элементам до завершения операции пересылки границ.
Data dependence in loop due to access to element <elem>	Обнаружена зависимость параллельного цикла по данным.
Using shadow element <elem> before asynchronous shadow renew competed	Использование теневого элемента <elem> распределенного массива во время выполнения операции обновления теневых граней.
Writing to remote data buffer <var>	Запись в буфер удаленного доступа <var>
Write to remote element <elem> in sequential branch	Запись в элемент <elem> распределенного массива в последовательной части программы без спецификации собственных вычислений.
Reading remote element <elem> in sequential branch	Использование нелокального элемента <elem> распределенного массива в последовательной части программы.
WAIT for reduction without START	Ожидание завершения редукции без ее фактического запуска.
Using an element outside of array limits: <elem>	Обращение к элементу массива за пределами его границ.
START for reduction without WAIT	Отсутствие операции ожидания завершения асинхронной редукции для соответствующей операции старта асинхронной редукции.
Reduction operation was not started	Специфицирована редукционная переменная, но соответствующее вычисление редукционной операции никогда не было стартовано.

2. Накопление и сравнение трассировки

Сообщение	Описание
Bad file structure	Нарушена структура файла трассировки.
Undefined keyword	Файл трассировки содержит неизвестное ключевое слово.
Bad command syntax	Неправильная структура записи трассировки.
Can't open a file <file name>	Невозможно открыть указанный файл.
Trace file <file name> is empty	Указанный файл трассировки не содержит информации.
Bad trace structure (missing current program construct)	Нарушена структура файла трассировки. Отсутствует запись начала исполняемой конструкции.
No current program construct	Отсутствует запись начала исполняемой конструкции.
Unexpected task or iteration of loop	Эталонная трассировка не содержит записи о выполнении данной итерации или задачи.
Double execution of task or iteration, No = <iter no>	Повторное исполнение той же итерации или задачи.
Unexpected execution of program construct	Эталонная трассировка не содержит записи о начале выполнения данного цикла или задачи.
Abnormal loop exit	Завершение цикла не соответствует записи в эталонной трассировке.
Unexpected use of variable	Эталонная трассировка не содержит записи об использовании данной переменной.
Unexpected trace record	Эталонная трассировка не содержит записи о исполнении данного события.
Different <type> values: <standard value> != <current value>	Значение переменной не соответствует эталонной трассировке.
Different <type> values of reduction variable: <standard value> != <current value>	Значение результата вычисления редукции не соответствует эталонной трассировке.

3. Структура конфигурационного файла трассировки

 # Trace size = <размер всего файла трассировки в байтах>
 # String count = <число строк всего файла трассировки>

SL, **PL** или **TR** <номер конструкции> (<номер объемлющей конструкции>) [<размерность конструкции>] {<имя файла>, <номер строки>} = <уровень накопления трассировки>, (<измерение> : <первая итерация> , <последняя итерация> , <шаг трассируемых итераций>)

 # Trace size = <размер трассировки в байтах данной конструкции для указанного уровня накопления трассировки>
 # String count = <число строк трассировки данной конструкции для указанного уровня накопления трассировки >

 # Count of traced iterations = <число трассируемых итераций цикла или задач>

EL: <номер конструкции>

. . . . . .

SL, **PL** или **TR** <номер конструкции> (<номер объемлющей конструкции>) [<размерность конструкции>] {<имя файла>, <номер строки>} = <уровень накопления трассировки>, (<измерение> : <первая итерация> , <последняя итерация> , <шаг трассируемых итераций>)

 # Trace size = <размер трассировки в байтах данной конструкции для указанного уровня накопления трассировки>
 # String count = <число строк трассировки данной конструкции для указанного уровня накопления трассировки >
 # Count of traced iterations = <число трассируемых итераций цикла или задач>

EL: <номер конструкции>

4. Структура трассировки вычислений

При трассировке вычислений накапливаемая трассировочная информация состоит из двух частей:

- заголовка трассировки;
- тела трассировки (может отсутствовать).

Заголовок присутствует в трассировке даже тогда, когда накопление трассировки отключено для всей программы. Его структура напоминает структуру конфигурационного файла трассировки, но без вычисляемых значений объемов трассировки для всей программы и для отдельных циклов:

MODE = <уровень накопления трассировки для всей программы>,

SL, **PL** или **TR** <номер конструкции> (<номер объемлющей конструкции>) [<размерность конструкции>] {<имя файла>, <номер строки>} = <уровень накопления трассировки>, (<измерение> : <первая итерация> , <последняя итерация> , <шаг трассируемых итераций>)

EL: <номер конструкции>

. . . . . .

SL, **PL** или **TR** <номер конструкции> (<номер объемлющей конструкции>) [<размерность конструкции>] {<имя файла>, <номер строки>} = <уровень накопления трассировки>, (<измерение> : <первая итерация> , <последняя итерация> , <шаг трассируемых итераций>)

EL: <номер конструкции>

Тело трассировки отсутствует, когда накопление трассировки отключено для всей программы. Иначе тело трассировки состоит из множества записей следующих типов:

- **Обращение к переменной на чтение.**
  RD: [<тип переменной>] <имя переменной> = <значение>; {<имя файла>, <номер строки>}
- **Обращение к переменной на запись (перед вычислением выражения).**
  BW: [<тип переменной>] <имя переменной>; {<имя файла>, <номер строки>}
- **Результат записи в переменную.**
  AW: [<тип переменной>] <имя переменной> = <значение>; {<имя файла>, <номер строки>}
- **Обращение к редукционной переменной на чтение.**
  RV_RD: [<тип переменной>] <имя переменной> = <значение>; {<имя файла>, <номер строки>}
- **Обращение к редукционной переменной на запись (перед вычислением выражения).**
  RV_BW: [<тип переменной>] <имя переменной>; {<имя файла>, <номер строки>}
- **Результат записи в редукционную переменную.**
  RV_AW: [<тип переменной>] <имя переменной> = <значение>; {<имя файла>, <номер строки>}
- **Результат вычисления редукции.**
  RV: [<тип переменной>] <значение>; {<имя файла>, <номер строки>}
- **Пропуск операторов при обращении к элементу распределенного массива в последовательной части программы.**
  SKP: {<имя файла>, <номер строки>}
- **Начало выполнения параллельного цикла.**
  PL: <номер цикла> (<номер родительской конструкции или пусто>) [<размерность цикла>] = <уровень трассировки: FULL, MODIFY, MINIMAL, NONE>, (<диапазон трассируемых итераций (может отсутствовать)>); {<имя файла>, <номер строки>}
- **Начало выполнения последовательного цикла.**
  SL: <номер цикла> (<номер родительской конструкции или пусто>) [<размерность цикла>] = <уровень трассировки: FULL, MODIFY, MINIMAL, NONE>, (<диапазон трассируемых итераций (может отсутствовать)>); {<имя файла>, <номер строки>}
- **Начало выполнения области задач.**
  TR: <номер области> (<номер родительской конструкции или пусто>) [<размерность области>] = <уровень трассировки: FULL, MODIFY, MINIMAL, NONE>, (<диапазон трассируемых задач>); {<имя файла>, <номер строки>}
- **Начало итерации (помещается в трассировку только при изменении самой вложенной итерации цикла) или параллельной задачи.**
  IT: <абсолютный индекс итерации (вычисляется из значений всех итерационных переменных) или номер задачи>, (<значение итерационной переменной>,< значение итерационной переменной >,…).
- **Конец выполнения параллельного цикла или области задач.**
  EL: <номер конструкции>; {<имя файла>, <номер строки>}

Приложение 5. Сообщения при сборе статистики

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