УДК 004.432

РАЗВИТИЕ DVM-СИСТЕМЫ

В. Ф. Алексахин [1], В. А. Бахтин [1,2], О. Ф. Жукова [1], Д. А. Захаров [1], В. А. Крюков [1,2], Н. В. Поддерюгина [1], О. А. Савицкая [1]

[1] Институт прикладной математики им. М.В. Келдыша Российской академии наук, г. Москва;
[2] Московский государственный университет им. М.В. Ломоносова, г. Москва

valex@keldysh.ru, bakhtin@keldysh.ru, socol@keldysh.ru, s123-93@mail.ru, krukov@keldysh.ru, npodderyugina@gmail.com, savol@keldysh.ru

Аннотация

DVM-система предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH. Эти языки используют единую DVMH-модель параллельного программирования и являются расширением стандартных языков Си и Фортран спецификациями параллелизма, оформленными в виде директив для компилятора. DVMH-модель позволяет создавать эффективные параллельные программы для гетерогенных вычислительных кластеров, в узлах которых в качестве вычислительных устройств наряду с универсальными многоядерными процессорами могут использоваться ускорители, графические процессоры или сопроцессоры Intel Xeon Phi. В статье представлены новые возможности DVM-системы, которые были разработаны в последнее время.

***Ключевые слова**: автоматизация разработки параллельных программ, DVM-система, ускоритель, ГПУ, Фортран, Си, нерегулярная сетка, неструктурированная сетка.*

ВВЕДЕНИЕ

Модель программирования DVMH [1, 2] построена на парадигме параллелизма по данным. В основе этой модели лежит понятие распределенного многомерного массива. При этом у каждого процессора имеются не только локальная часть распределенного массива, но и так называемые теневые грани – копии элементов из локальных частей соседних процессоров, через которые осуществляется основное взаимодействие процессоров. Распределение вычислений про-

изводится посредством их отображения на распределенные массивы, при этом обращения происходят либо в свою локальную часть, либо в теневые грани, определяемые как продолжение локальной части по конкретному измерению распределенного массива на заранее известную ширину. Например, для шаблона типа «крест» с 4 соседями, элемент с индексами (i,j) рассчитывается по элементам с индексами (i-1,j), (i,j-1), (i+1,j), (i,j+1), что приводит к необходимости иметь теневые грани ширины 1 по обоим измерениям.

DVMH-компиляторы преобразуют обращения к распределенным многомерным массивам в форму, независимую от размеров и положения локальной части на каждом процессоре, при этом исходные индексные выражения остаются нетронутыми. В результате каждое обращение к распределенным данным ведется в глобальных (исходных) индексах, а при доступе к памяти применяются вычисляемые во время выполнения коэффициенты и смещения для каждого измерения. Такой подход (в отличие от изменения индексных выражений) позволяет абстрагироваться от содержания распараллеливаемых циклов, но и вводит серьезное ограничение на форму адресуемой каждым процессором части распределенного массива, называемой расширенной локальной частью, которая является объединением локальной части и теневых граней. В модели DVMH используются блочные распределения массивов с теневыми гранями. Таким образом, расширенная локальная часть представляет собой подмассив исходного массива вида (A1:B1, A2:B2, A3:B3, ..., An:Bn). Такое ограничение затрудняет процесс распараллеливания программ на неструктурированных сетках с помощью DVM-системы [3].

В статье описаны новые возможности DVM-системы, нацеленные на борьбу с этим ограничением. В первой главе представлены новые виды распределений массивов, новые конструкции для работы с теневыми гранями, новые средства для перехода от глобальных индексов к локальным, а также приведен пример программы на языке Fortran-DVMH, использующей названные возможности.

Во второй главе описаны средства, которые позволяют программисту вручную распределять данные, используя MPI или другие технологии параллельного программирования, оставляя при этом возможность использования DVM-языков внутри узла кластера для отображения вычислений по ядрам цен-

трального процессора или графического ускорителя. Новые спецификации параллелизма существенно упрощают процесс разработки программ, использующих неструктурированные сетки.

РАСШИРЕНИЕ DVM-СИСТЕМЫ ДЛЯ РАБОТЫ С НЕСТРУКТУРИРОВАННЫМИ СЕТКАМИ

Для работы с неструктурированными сетками в DVM-системе реализован новый вид распределения массивов и шаблонов – поэлементное распределение. Этот вид распределения не накладывает никаких ограничений на то, какие элементы массива должны располагаться на одном и том же процессоре или какие элементы массива должны располагаться на соседних процессорах. Напротив, он позволяет задать произвольную принадлежность каждого элемента массива независимо.

Введены два новых правила поэлементного распределения: косвенное (INDIRECT) и производное (DERIVED). Косвенное распределение задается массивом целых чисел, размер которого равен размеру косвенно распределяемого измерения, а значения задают номер домена. При этом доменов может быть как больше числа процессоров, так и меньше. DVM-система гарантирует принадлежность всех элементов домена одному и тому же процессору.

Производное распределение задается правилом, по форме похожим на правило выравнивания (ALIGN) модели DVMH. Однако у него появляется значительно большая гибкость. Синтаксис можно описать так, как показано на рис. 1.

```
      indirect-rule ::= INDIRECT ( var-name )
      derived-rule ::= DERIVED ( derived-elem-list WITH derived-templ )
      derived-elem ::= int-range-expr
      int-range-expr ::= прозвольное целочисленное выражение + в индексных выра-
жениях допустимы диапазоны, использование align-dummy переменных.
      derived-templ ::= var-name [ derived-templ-axis-spec ]...
      derived-templ-axis-spec ::= [ ] | [ @ align-dummy [ + shadow-name ]... ] |
[ int-expr ]
```

Рис. 1. Формула БНФ для новых правил распределения

Все ссылки на распределенные массивы в int-range-expr обязаны быть доступны (элемент входит в расширенную локальную часть) для соответствующего элемента шаблона (перебор элементов шаблона осуществляется по его локаль-

ной части и указанным теневым граням). Если производным правилом один и тот же элемент подлежит распределению сразу на несколько процессоров, то DVM-система решает, на какой из них фактически будет распределен такой элемент, а на остальных процессорах добавляет его в теневую грань с названием «overlay». Элементов, не распределенных ни на один процессор, быть не должно. Такие случаи являются ошибкой времени выполнения и приводят к останову. Вычисленные несуществующие индексы распределяемого массива игнорируются, не приводя к ошибке.

Наложение (overlay) вводится для возможности согласованного распределения сеточных элементов, например, ячейки, ребра, вершины. В таком случае появляется возможность построить одно распределение на основе другого, причем в любой последовательности.

В результате такого распределения у массива появляются два вида нумерации элементов: глобальная (она же исходная в последовательной программе) и локальная. Локальная нумерация непрерывна в рамках одного процессора, т. е. существует такой порядок локальных элементов, что их локальные индексы полностью заполнят некоторый целочисленный отрезок [Li, Hi].

Также вводятся поэлементые теневые грани. Теневая грань – это набор элементов, не принадлежащих текущему процессу (требование принадлежности соседнему процессу снимается), для которых, во-первых, возможен доступ без специальных указаний из любой точки программы, и, во-вторых, введены специальные средства работы с ними: обновление указанием SHADOW_RENEW, расширение параллельного цикла указанием SHADOW_COMPUTE и т. п.

В отличие от традиционных, поэлементные теневые грани добавляются к шаблонам во время работы программы и имеют имя для ссылки на них. Задаются они практически так же, как и производное распределение (рис. 2).

```
      shadow-add ::= SHADOW_ADD ( templ-name [ shadow-axis ]... = shadow-name ) [
INCLUDE_TO ( var-name-list ) ]
      shadow-axis ::= [ ] | [ derived-elem-list WITH derived-templ ]
```

Рис. 2. Формула БНФ для задания поэлементных теневых граней

Ровно один из shadow-axis должен быть непустыми скобочками. Все массивы из списка, указанного в INCLUDE_TO, должны быть выравнены на шаблон, к

измерению которого добавляется теневая грань. В результате выполнения такой директивы к шаблону добавляется теневая грань и включается в указанные распределенные массивы. После этой операции теневые элементы массивов доступны на чтение из программы, а также могут обновляться с помощью директивы SHADOW_RENEW.

Для реализации поэлементных теневых граней и производного распределения компилятор на основе указанных выражений генерирует специального вида функцию, в которую передаются системой поддержки параметры для обхода локальной части шаблона. Эта функция, обходя шаблон, заполняет буфер индексов элементов согласно выражениям в левой части правила отображения, а затем возвращает обратно в систему поддержки. Затем буфер анализируется средствами системы поддержки.

Для экспериментальной эксплуатации этих возможностей была введена вспомогательная директива локализации значений индексного массива, которая изменяет значения целочисленного массива, заменяя глобальные индексы указанного целевого массива на локальные (рис. 3).

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

Рис. 3. Формула БНФ для директивы локализации значений индексного массива

После проведения такой операции становится возможным использовать имеющийся способ компиляции параллельных циклов: они будут выполняться полностью в локальных индексах.

Вместе с модификацией директивы теневых обменов и реализацией обменов для поэлементных теневых граней (которые теперь происходят не обязательно с соседними процессорами, а с произвольным подмножеством процессоров), этот набор расширений позволяет распараллелить и запустить приложения, использующие нерегулярные сетки, на кластере с ускорителями.

Для иллюстрации новых возможностей рассмотрим небольшой пример программы на языке Fortran, реализующий трехмерный алгоритм Якоби (рис. 4). В данной программе вместо трехмерных массивов используются одномерные

массивы. Из-за этого появляется косвенная адресация, инструментов для работы с которой ранее в DVM не было.

```
            program JAC_INDIRECT
            parameter (L=100, itmax=5000)
            real*8:: tmp,eps, maxeps=0.005
            integer x_t,y_t,z_t,cur
            real*8, allocatable :: A(:),B(:)
            integer, allocatable :: ibstart(:), ibend(:), ib(:)
            integer, allocatable :: indir_x(:), indir_y(:),indir_z(:)
            allocate(A(L*L*L),B(L*L*L), ibstart(L*L*L), ibend(L*L*L))
            allocate(indir_x(L*L*L), indir_y(L*L*L), indir_z(L*L*L))
! Здесь происходит создание одномерного массива, который "эмулирует"
! трехмерный массив в обычном трехмерном алгоритме Якоби
            cur = 1
            do i = 1,L*L*L
              x_t = (i-1) / (L*L)
              y_t = mod((i-1) / L, L)
              z_t = mod(i-1, L)
              indir_x(i) = x_t
              indir_y(i) = y_t
              indir_z(i) = z_t
              ibstart(i) = cur
              if (x_t.gt.0) cur = cur + 1
              if (x_t.lt.L-1  cur = cur + 1
              if (y_t.gt.0)   cur = cur + 1
              if (y_t.lt.L-1) cur = cur + 1
              if (z_t.gt.0)   cur = cur + 1
              if (z_t.lt.L-1) cur = cur + 1
              ibend(i) = cur - 1
            enddo
            allocate(ib(cur-1))
            cur = 1
            do i = 1,L*L*L
              x_t = (i-1) / (L*L)
              y_t = mod((i-1) / L, L)
              z_t = mod(i-1, L)
              if (x_t.gt.0) then
                  ib(cur) = i - (L*L)
                  cur = cur + 1
              endif
              if (x_t.lt.L-1) then
                  ib(cur) = i+(L*L)
                  cur = cur + 1
              endif
              if (y_t.gt.0) then
                  ib(cur) = i-L
                  cur = cur + 1
              endif
              if (y_t.lt.L-1) then
                  ib(cur) = i+L
                  cur = cur + 1
              endif
              if (z_t.gt.0) then
                  ib(cur) = i-1
                  cur = cur + 1
```

```
              endif
              if (z_t.lt.L-1) then
                  ib(cur) = i+1
                  cur = cur + 1
              endif
            enddo
! Для упаковки массива используется аналогичный CSR (Compressed Sparse Row)
! формат. У каждого элемента может быть до 6 соседних элементов - слева и
! справа по каждому из трех измерений. Для i-ого элемента массива A список
! его соседей содержится в массиве ib начиная с индекса ibstart(i) и заканчивая
! индексом ibend(i). Выше происходит создание этой похожей на CSR
! структуры. Также заполняются массивы indir_x/y/z, которые содержат
! индексы, которые были у элемента в трехмерном массиве.

! Перед итерационным циклом массивы заполняются. Так как все элементы
! теперь сложены в одномерный массив - требуется проверять трехмерные
! индексы, чтобы исключить обработку граничных элементов.
            do i = 1, L*L*L
              A(i) = 0
              if (indir_x(i) == 0 .or. indir_x(i) == L-1 .or.
           &      indir_y(i) == 0 .or. indir_y(i) == L-1 .or.
           &      indir_z(i) == 0 .or. indir_z(i) == L-1) then
                  B(i) = 0
              else
                  B(i) = 4 + indir_x(i) + indir_y(i) + indir_z(i)
              endif
            enddo
! После заполнения применяется видоизмененный алгоритм Якоби
            do it = 1, itmax
              eps = 0
              do i = 1,L*L*L
                 if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
           &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
           &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
                     tmp  = ABS(B(i) - A(i))
                     eps = MAX(tmp, eps)
                     A(i) = B(i)
                 endif
              enddo
              do i = 1, L*L*L
                 if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
           &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
           &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
! Косвенная адресация
                     B(i) = (A(ib(ibstart(i)))   + A(ib(ibstart(i)+1))
           &               + A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3))
           &               + A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5)))
           &               / 6.0
                 endif
              enddo
              print 200,  it, eps
      200     format(' it = ', i4, '   eps = ', e14.7)
              if ( eps .lt. maxeps )    exit
            enddo
            deallocate(ibstart,ibend)
            deallocate(ib)
            deallocate(A,B,indir_x,indir_y,indir_z)
            end program
```

Рис. 4. Последовательная версия программы, реализующая алгоритм Якоби

Начнем рассматривать параллельный вариант программы:

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

Первое изменение – добавление массива MAP. Этот массив будет служить «картой распределения», на основе которой мы будем распределять данные. Также объявляются два шаблона – статический шаблон E, который будет распределяться поэлементно, и динамический шаблон E2, о котором будет сказано чуть позднее. Для этих шаблонов указана директива DISTRIBUTE без параметров, которая означает, что эти шаблоны будут распределены позже. Также указана директива ALIGN без параметров для всех массивов, которая говорит о том, что эти массивы будут в дальнейшем выровнены на какой-либо шаблон или уже

распределенный массив. После этого добавляется функция заполнения карты – fillMap. Одна из возможных реализаций данной функции выглядит так:

```
      subroutine fillMap(map,L,axis)
      integer  numproc
      integer i,L,axis
      integer map(L*L*L)
! Эта строка нужна для совместимости программы с обычными
! компиляторами
      PROCESSORS_SIZE(axis) = 1
      numproc = PROCESSORS_SIZE(axis)
        do i = 1,L*L*L
           map(i) = ((i-1) * numproc) / (L*L*L)
        enddo
      end subroutine
```

PROCESSORS_SIZE(axis) – служебная функция, которая возвращает количество процессоров в оси axis решетки процессоров, на которой была запущена программа. Так как данная программа одномерная – axis равно 1, и в дальнейшем все будет описываться с учетом того, что решетка запуска одномерная. Конкретная реализация имитирует блочное распределение – карта делится на равные блоки, и все элементы из первого блока идут на процессор с индексом 0, все элементы из второго блока идут на процессор с индексом 1 и так далее.

После заполнения карты распределения она тут же используется в директиве REDISTRIBUTE. Здесь в качестве типа распределения указан INDIRECT – поэлементное распределение. При поэлементном распределении i-й элемент шаблона оказывается на том процессоре, индекс которого указан в карте на i-й позиции. Это позволяет распределять данные в любом формате – можно использовать блочное распределение, как здесь, можно распределять элементы поочередно, когда каждый следующий элемент распределяется на другой процессор, а можно и вовсе распределить их случайным образом. У программиста есть возможность задать любое отображение.

После этого все нужные массивы выравниваются на новосозданный шаблон через директиву REALIGN. После выполнения этой директивы элементы с

индексом i для всех указанных в ней массивов будут распределены на тот же процессор, на который был распределен i-й элемент шаблона E. Использование шаблонов для задания изначального поэлементного распределения в данном случае является необходимым, распределять поэлементно массив напрямую нельзя.

Следующее изменение в программе появляется после выделения памяти под массив ib:

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

Здесь появляется еще один новый тип распределения данных – DERIVED (производное распределение). Производное распределение – это вариант поэлементного распределения, идея которого состоит в том, что оно как раз является «производным» из какого-либо другого распределения. Стоит вспомнить, как выглядела косвенная адресация в последовательной программе:

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

Отсюда мы можем заметить, что на одном процессоре вместе с B(i), который у нас уже распределен поэлементно, мы должны иметь элементы массива ib с индексами от ibstart(i) до ibstart(i)+5, то есть все элементы-соседи, учитывая формат хранения данных – конечный индекс на самом деле будет ibend(i), который для всех неграничных элементов как раз равен ibstart(i)+5. Обеспечить наличие всех нужных элементов мы сможем через производное распределение. Для распределения массива ib будет использоваться шаблон E2, который создается динамически, поскольку на старте программы мы не знаем размера массива ib, значит, и размер шаблона. Сразу после этого к шаблону применяется директива redistribute с производным типом распределения. Данная директива означает, что в новом шаблоне E2 элементы с индексами, начиная с ibstart(i) и

заканчивая ibend(i), должны находиться на том же процессоре, где находится i-й элемент шаблона E. Вместо указания диапазона ibstart(i):ibend(i) в директиве может указываться просто список индексов через запятую (или вовсе один индекс). После этого массив ib выравнивается на новосозданный шаблон, и тем самым гарантируется, что на одном процессоре вместе с элементом B(i) будут находиться все его соседи. Для всех неграничных элементов массива B это значит, что на один процессор вместе с элементом B(i) попадут все элементы от ib(ibstart(i)) до ib(ibstart(i)+5). Стоит отметить, что если при создании производного шаблона сразу несколько процессоров захотят получить один и тот же элемент, то этот элемент дается какому-то одному из процессоров, а для других он помещается в автоматически создаваемую теневую грань.

Следующее изменение появляется после заполнения массива ib:

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

Директива LOCALIZE – служебная директива, которая преобразует глобальные индексы в локальные, что необходимо для корректной адресации массивов. Данная директива должна быть применена ко всем массивам, которые используются для индексации поэлементно распределенных массивов. В директиве слева указывается массив, который нужно локализовать, а справа – массив, который будет индексироваться локализуемым массивом. Для массивов с 2 и более измерениями необходимо также указывать измерение, на которое производится локализация. Директива должна использоваться после того, как локализуемый массив был полностью заполнен и больше не будет изменяться, но до

его использования для индексации распределенного массива в параллельном цикле или в директиве SHADOW_ADD. В данном случае – массивы ibstart и ibend уже были заполнены и будут использованы для индексации тут же в директиве SHADOW_ADD.

Еще раз вспомним, как выглядела косвенная индексация в основном цикле:

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

Мы позаботились о массиве ib, но у нас остался массив A, который индексируется посредством массива ib. Для того чтобы гарантировать наличие нужных элементов массива A на процессоре, где находится элемент B(i), нам необходимо добавить теневую грань к массиву A, что и делает директива SHADOW_ADD. Данный экземпляр директивы говорит о том, что на один процессор вместе с i-м элементом шаблона E (часть WITH E(@i)) мы должны добавить в теневую грань все элементы шаблона E (первое вхождение E в директиве), индексы которых находятся в массиве ib, начиная с индекса ibstart(i) и заканчивая индексом ibend(i). Далее эта теневая грань получает название «nei1», и указывается, что эту теневую грань нужно добавить для массива A. Тем самым мы создали теневую грань, которая для каждого элемента A(i) содержит всех его соседей. При этом директива SHADOW_ADD следит за тем, чтобы в теневой грани не было дублирующих элементов. Если элемент уже присутствует на процессоре, он не будет добавлен в теневую грань. Необходимо отметить, что массив ib локализуется после директивы SHADOW_ADD. Так как он используется для индексации массива A, локализуется он именно на него.

После этого остается лишь указать директивы PARALLEL и регионы:

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

```
     &      indir_z(i) == 0 .or. indir_z(i) == L-1) then

            B(i) = 0
        else
            B(i) = 4 + indir_x(i) + indir_y(i) + indir_z(i)
        endif
      enddo
!DVM$ END REGION
      do it = 1, itmax
!DVM$ REGION
        eps = 0
!DVM$   PARALLEL (i) ON B(i), REDUCTION(MAX(eps)), PRIVATE(tmp)
        do i = 1,L*L*L
           if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
     &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
     &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
               tmp  = ABS(B(i) - A(i))
               eps = MAX(tmp, eps)
               A(i) = B(i)
           endif
        enddo
!DVM$   PARALLEL (i) ON B(i), SHADOW_RENEW(A)
        do i = 1, L*L*L
           if (indir_x(i) /= 0 .and. indir_x(i) /= L-1 .and.
     &         indir_y(i) /= 0 .and. indir_y(i) /= L-1 .and.
     &         indir_z(i) /= 0 .and. indir_z(i) /= L-1) then
               B(i) = (A(ib(ibstart(i)))   + A(ib(ibstart(i)+1))
     &               + A(ib(ibstart(i)+2)) + A(ib(ibstart(i)+3))
     &               + A(ib(ibstart(i)+4)) + A(ib(ibstart(i)+5)))
     &               / 6.0
           endif
        enddo
!DVM$ END REGION
```

```
!DVM$ GET_ACTUAL(eps)
        print 200,  it, eps
200     format(' it = ', i4, '   eps = ', e14.7)
        if ( eps .lt. maxeps )    exit
      enddo
```

Директива PARALLEL в данном случае распределяет витки цикла поэлементно на основе распределения массива B. i-й виток цикла выполняется на том процессоре, где находится элемент B(i), значит, на том процессоре, индекс которого был записан в map(i) на момент выполнения директивы распределения шаблона E.

Клауза SHADOW_RENEW для A в данном случае будет обновлять все теневые грани, привязанные к массиву A. В этом примере таковая одна – та самая nei1, которая была объявлена через SHADOW_ADD. Остальные директивы/клаузы ничем не отличаются от стандартного DVM без расширения. Клауза REDUCTION(max(eps)) обеспечивает то, что на каждом процессоре у нас будет максимальным значение eps по итогам всех витков цикла, а не только витков этого процессора. Клауза PRIVATE(tmp) говорит о том, что переменная tmp приватная, значит, ее значение на одном витке не влияет на другие витки. Директивы REGION и END REGION показывают участки кода, которые следует выполнять на графическом ускорителе, если таковой выделен программе, и директива GET_ACTUAL(eps) указывает на то, что конкретное значение переменной eps находится на графическом ускорителе, и его нужно скопировать в оперативную память.

Полученная программа может выполняться на гетерогенном вычислительном кластере с ускорителями.

ДОПОЛНИТЕЛЬНОЕ РАСПАРАЛЛЕЛИВАНИЕ СУЩЕСТВУЮЩИХ MPI-ПРОГРАММ

В настоящее время, когда параллельные машины уже не одно десятилетие эксплуатируются для проведения расчетов, имеется множество программ, которые уже распараллелены на кластер, однако не имеют распараллеливания по

ядрам центрального процессора, а также не используют графические ускорители.

Традиционно в DVM-подходе весь процесс программирования (или распараллеливания имеющихся последовательных программ) начинается с распределения массивов, а затем отображения на них параллельных вычислений. Это означает, что для использования средств DVM-системы распараллеленные, например, на MPI, программы приходится превращать обратно в последовательные и заменять распределенные вручную данные и вычисления на описанные на DVM-языке распределенные массивы и параллельные циклы.

Однако, во-первых, автору не всегда хочется отказываться от своей параллельной программы, во-вторых, не всегда удастся перевести исходную схему распределения данных и вычислений на DVM-язык. В частности, перевод задач на нерегулярных сетках в модель DVMH может потребовать применения нетривиальных решений, что не всегда возможно.

Одним из способов избавиться от обеих проблем является новый режим работы DVM-системы, в котором она не вовлечена в межпроцессорное взаимодействие, а работает локально в каждом процессе.

Данный режим включается заданием специально созданной MPI-библиотеки при сборке DVM-системы. Эта библиотека не производит никаких коммуникаций и не конфликтует с реальными MPI-реализациями. В результате для системы поддержки выполнения DVMH-программ создается иллюзия запуска программы на 1 процессоре.

Кроме такого режима, в языках Fortran-DVMH и C-DVMH введено понятие нераспределенного параллельного цикла, для которого нет необходимости задавать отображение на распределенный массив. Например, трехмерный параллельный цикл может выглядеть так (рис. 5):

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

```
DO I = L1,H1
   DO J = L2, H2
      DO K = L3, H3
...
```

Рис. 5. Нераспределенный параллельный цикл

По определению такой цикл выполняется всеми процессорами текущей многопроцессорной системы, но т. к. DVM-система в описанном новом режиме считает многопроцессорной системой ровно один процесс, такая конструкция не приводит к размножению вычислений, а только лишь позволяет использовать параллелизм внутри одного процесса – использовать ядра центрального процессора или графического ускорителя. Как следствие, появляется возможность не задавать ни одного распределенного в терминах модели DVMH массива и в то же время пользоваться возможностями DVM-системы:

– использовать параллелизм на общей памяти (задействовать ядра центрального процессора);

– задействовать графические ускорители: не только «наивное» портирование параллельного цикла на ускоритель, но и выполнение автоматической реорганизации данных, упрощенное управление перемещениями данных;

– подбирать оптимизационные параметры;

– использовать удобные средства отладки производительности.

Такой режим может быть использован, в том числе, для получения промежуточных результатов в процессе проведения полноценного распараллеливания программы в модели DVMH. Он позволяет быстро и заметно проще получить программу для многоядерного центрального процессора и графического ускорителя, а также оценить перспективы ускорения целевой программы на кластере с многоядерными процессорами и ускорителями.

ИСПОЛЬЗОВАНИЕ НОВЫХ ВОЗМОЖНОСТЕЙ

Для демонстрации использования новых возможностей при работе с неструктурированными сетками рассмотрим программу для решения задачи газовой динамики методом Галеркина, который широко применяется на практике и характеризуется высоким порядком точности получаемого решения. В качестве

тестовой задачи рассмотрим простую волну, в которой энтропия и инвариант Римана являются постоянными [4, 5], используется классический лимитер Кокбурна, который легко реализуется в многомерном случае на сетках произвольной структуры. Основными вычислительными элементами в программе являются массивы ячеек сетки, содержащие различные величины и характеристики. Сетка является неструктурной, а ячейки в этой сетке – треугольные, поэтому обращения к соседним элементам по всей программе происходят с использованием косвенной адресации.

Для данной программы были разработаны 2 параллельных версии – с использованием «старых» и «новых» возможностей DVM-системы. «Старая» версия программы использует блочное распределение данных, «новая» версия программы использует расширение для работы с неструктурированными сетками, описанное в данной статье. На рис. 6 показано сравнение времен выполнения 2-х версий программы для сетки из 200000 ячеек на различном числе ядер вычислительного кластера К-100 (ИПМ им. М.В. Келдыша РАН).

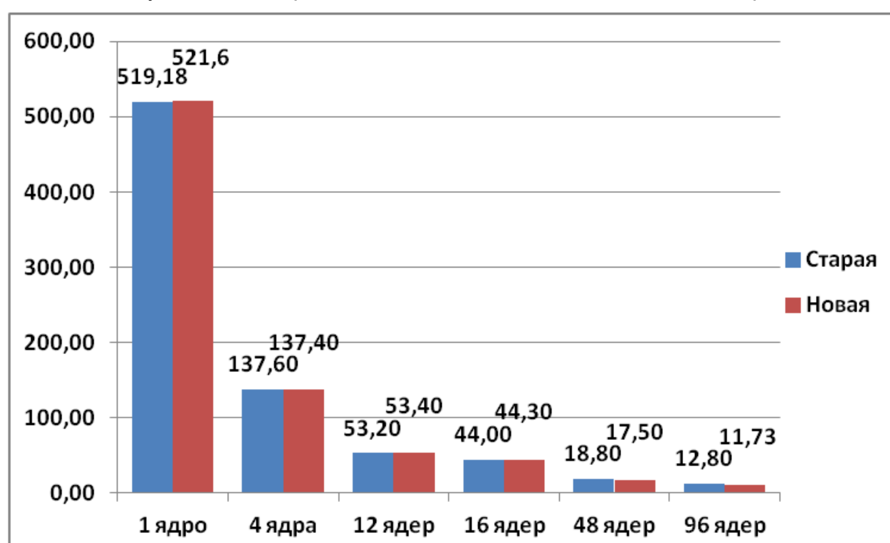

Рис. 6. Времена выполнения 2-х версий программы на кластере К-100 (в секундах)

Времена выполнения «старой» и «новой» версий программы практически не отличаются. При использовании большого числа ядер «новая» версия программы начинает выполняться быстрее, чем «старая». При этом следует отметить, что процесс распараллеливания программы без использования расширения был значительно сложнее. Потребовалось существенное изменение кода

программы, а самое главное – такое распараллеливание стало возможным лишь после переупорядочивания элементов сетки по принципу локальности. В результате выполнения данного переупорядочивания сетки удалось найти небольшое значение M, при котором все соседи ячейки с номером N имели номера, отличающиеся от N не более, чем на M, что позволило использовать блочное распределение для массивов и старый механизм теневых граней для обновления значений соседних ячеек (максимальный размер теневых граней был задан равным M).

Для демонстрации возможности дополнительного распараллеливания существующих MPI-программ рассмотрим программу, являющуюся частью большого развитого комплекса вычислительных программ (В.А. Гасилов, А.С. Болдарев, ИПМ им. М.В. Келдыша РАН). Будучи ориентированным на решение по явной схеме систем гиперболических уравнений (в основном, газовой динамики) в двумерных областях сложной формы с использованием неструктурированных сеток, этот код был написан на C++ с очень широким использованием объектно-ориентированного подхода для обеспечения максимальной универсальности и простоты дальнейшего развития.

Так как эта программа является частью целого комплекса, код ее основан на богатой платформе базовых понятий и структур данных. Это приводит к значительным размерам (39 тыс. строк) и сложности всей программы, если ее рассматривать целиком.

Полноценное распараллеливание программы в модели DVMH вряд ли возможно без рассмотрения и модификации всей программы. Новые возможности позволили выполнить «локальное» распараллеливание вычислительно-емких частей программы. Модификации подверглись лишь 3 из 39 тыс. строк программы.

В результате такого распараллеливания на 12 ядрах ЦПУ с использованием OpenMP-нитей было получено ускорение в 9,83 раза относительно последовательной версии, а на ГПУ NVIDIA GTX Titan — в 18 раз относительно последовательной версии. Данные результаты подтверждают эффективность отображения рассматриваемой программы DVM-системой на ускорители и многоядерные процессоры и дают основания продолжить распараллеливание программы уже с использованием распределенных массивов в модели DVMH.

ЗАКЛЮЧЕНИЕ

DVM-система автоматизирует процесс разработки параллельных программ.

Получаемые DVMH-программы без каких-либо изменений могут эффективно выполняться на кластерах различной архитектуры, использующих многоядерные универсальные процессоры, графические ускорители и сопроцессоры Intel Xeon Phi. Это достигается за счет различных оптимизаций, которые выполняются как статически, при компиляции DVMH-программ, так и динамически.

Выше были представлены новые возможности DVM-системы, которые позволяют расширить область ее применимости и позволяют рапараллеливать не только задачи на структурированных сетках, для которых DVM-система была предназначена изначально [6], но и задачи на неструктурированных сетках.

В последнее время для численного решения задач математической физики стали активно использоваться адаптивные сетки – метод, который позволяет локально перестраивать сетку. Адаптация требуется, чтобы сгустить сеточные элементы в областях, где они наиболее необходимы, оставив сетку грубой в остальных местах. Такие сетки позволяют максимально точно передать ударные волны, фазовые переходы и другие области больших градиентов функций. Авторы проекта работают над расширением возможностей DVM-системы для поддержки адаптивных сеток.

СПИСОК ЛИТЕРАТУРЫ

1. Язык C-DVMH. C-DVMH компилятор. Компиляция, выполнение и отладка CDVMH-программ. URL: http://dvm-system.org/static_data/docs/CDVMH-reference-ru.pdf

2. Язык Fortran-DVMH. Fortran-DVMH компилятор. Компиляция, выполнение и отладка DVMH-программ. URL: http://dvm-system.org/static_data/docs/FDVMH-user-guide-ru.pdf

3. Система автоматизации разработки параллельных программ (DVM-система). URL: http://dvm-system.org

4. *Ладонкина М.Е., Неклюдова О.А., Тишкин В.Ф.* Лимитер повышенного порядка точности для разрывного метода Галеркина на треугольных сетках // Препринты ИПМ им. М.В.Келдыша. 2013. № 53. 26 с.

5. *Ладонкина М.Е., Неклюдова О.А., Тишкин В.Ф.* Исследование влияния лимитера на порядок точности решения разрывным методом Галеркина // Препринты ИПМ им. М.В. Келдыша. 2012. № 34. 31 с.

6. *Бахтин В.А., Захаров Д.А., Колганов А.С., Крюков В.А., Поддерюгина Н.В., Притула М.Н.* Решение прикладных задач с использованием DVM-системы // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8. № 1. С. 89–106. DOI: 10.14529/cmse190106

PROGRESS IN DVM-SYSTEM

V.F. Aleksahin*[1]*, V.A. Bakhtin*[1,2]*, O.F. Zhukova*[1]*, D.A. Zakharov*[1]*, V.A. Krukov*[1,2]*,
***N.V. Podderyugina*[1]*, O.A. Savitskaya*[1]**

[1] *Keldysh Institute of Applied Mathematics,*
[2] *Lomonosov Moscow State University*

valex@keldysh.ru, bakhtin@keldysh.ru, socol@keldysh.ru, s123-93@mail.ru, krukov@keldysh.ru, npodderyugina@gmail.com, savol@keldysh.ru

Abstract

DVM-system is designed for the development of parallel programs of scientific and technical calculations in the C-DVMH and Fortran-DVMH languages. These languages use a single DVMH-model of parallel programming model and are an extension of the standard C and Fortran languages with parallelism specifications in the form of compiler directives. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters, in the nodes of which accelerators, graphic processors or Intel Xeon Phi coprocessors can be used as computing devices along with universal multi-core processors. The article presents new features of DVM-system that have been developed recently.

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

REFERENCES

1. C-DVMH language, C-DVMH compiler, compilation, execution and debugging of DVMH programs. URL: http://dvm-system.org/static_data/docs/CDVMH-reference-en.pdf
2. Fortran DVMH language, Fortran DVMH compiler, compilation, execution and debugging of DVMH programs. URL: http://dvm-system.org/static_data/docs/FDVMH-user-guide-en.pdf
3. System for automating the development of parallel programs (DVM-system). URL: http://dvm-system.org
4. *Ladonkina M.E., Neklyudova O.A., Tishkin V.F.* Limiter povyshennogo poryadka tochnosti dlya razryvnogo metoda Galerkina na treugol'nyh setkah // Preprinty IPM im. M.V. Keldysha. 2013. No 53. 26 s.
5. *Ladonkina M.E., Neklyudova O.A., Tishkin V.F.* Issledovanie vliyaniya limitera na poryadok tochnosti resheniya razryvnym metodom Galerkina // Preprinty IPM im. M.V. Keldysha. 2012. No 34. 31 s.
6. *Bakhtin V.A., Zaharov D.A., Kolganov A.S., Krukov V.A., Podderyugina N.V., Pritula M.N.* Development of Parallel Applications Using DVM-system. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering. 2019. V. 8. No 1. P. 89-106. (in Russian) DOI: 10.14529/cmse190106

СВЕДЕНИЯ ОБ АВТОРАХ

АЛЕКСАХИН Валерий Федорович – ведущий инженер ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы.

Valery Fedorovich ALEKSAHIN – leading engineer of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms.

email: valex@keldysh.ru

БАХТИН Владимир Александрович – ведущий научный сотрудник ИПМ им. М.В. Келдыша РАН, доцент кафедры системного программирования факультета ВМК МГУ им. М.В. Ломоносова. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы; методы, средства и системы обработки данных большого объема.

Vladimir Aleksandrovich BAKHTIN – leading researcher of Keldysh Institute of Applied Mathematics, docent of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; methods, tools and systems of large data processing.

email: bakhtin@keldysh.ru

ЖУКОВА Ольга Федоровна – научный сотрудник ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы.

Olga Fedorovna ZHUKOVA – researcher of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms.

email: socol@keldysh.ru

ЗАХАРОВ Дмитрий Александрович – программист ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – программные средства и системы для распределенных вычислений; параллельные алгоритмы; автоматизация параллельного программирования; распараллеливание программ, использующих неструктурные сетки.

Dmitry Aleksandrovich ZAKHAROV – programmer of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; automatization of parallel programming; parallelization of unstructured grid applications.

email: s123-93@mail.ru

КРЮКОВ Виктор Алексеевич – главный научный сотрудник ИПМ им. М.В. Келдыша РАН, профессор кафедры системного программирования факультета ВМК МГУ им. М.В. Ломоносова. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы; методы, средства и системы обработки данных большого объема.

Victor Alekseevich KRUKOV – chief researcher of Keldysh Institute of Applied Mathematics, professor of the faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms; methods, tools and systems of large data processing.

email: krukov@keldysh.ru

ПОДДЕРЮГИНА Наталия Викторовна – старший научный сотрудник ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы.

Nataliya Victorovna PODDERYUGINA – senior researcher of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms.

email: npodderyugina@gmail.com

САВИЦКАЯ Ольга Антониевна – научный сотрудник ИПМ им. М.В. Келдыша РАН. Сфера научных интересов – математическое обеспечение, программные средства и системы для распределенных вычислений; параллельные алгоритмы.

Olga Antonievna SAVITSKAYA – researcher of Keldysh Institute of Applied Mathematics. Research interests include mathematical software, software and systems for distributed computing; parallel algorithms.

email: savol@keldysh.ru

Материал поступил в редакцию 13 ноября 2019 года