В.Ф. Алексахин, В.А. Бахтин,
О.Ф. Жукова, Д.А. Захаров, В.А. Крюков,
Н.В. Поддерюгина, О.А. Савицкая

Новые возможности DVM-системы

Рекомендуемая форма библиографической ссылки

Алексахин В.Ф., Бахтин В.А., Жукова О.Ф., Захаров Д.А., Крюков В.А., Поддерюгина Н.В., Савицкая О.А. Новые возможности DVM-системы // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции (23-28 сентября 2019 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2019. — С. 25-39. — URL: http://keldysh.ru/abrau/2019/theses/36.pdf doi:10.20948/abrau-2019-36

Размещена также презентация к докладу

Новые возможности 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] ИПМ им. М.В. Келдыша РАН
[2] МГУ им. М.В. Ломоносова

Аннотация. DVM-система предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH. Эти языки используют единую модель параллельного программирования (DVMH-модель) и являются расширением стандартных языков Си и Фортран спецификациями параллелизма, оформленными в виде директив компилятору. 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

New features of 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

Abstract. DVM-system is designed for the development of parallel programs of scientific and technical calculations in C-DVMH and Fortran-DVMH languages. These languages use a single parallel programming model (DVMH model) and are extensions of the standard C and Fortran languages with parallelism specifications, written in the form of directives to the compiler. The DVMH model makes it possible to create efficient parallel programs for heterogeneous computing clusters, 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

Для удовлетворения желания увеличения точности вычислений, исследователям–вычислителям приходится значительно измельчать расчетную сетку. Это приводит к пропорциональному росту потребления памяти ЭВМ и увеличению времени расчетов. Частично с этой проблемой позволяет справиться переход с использования структурированных сеток на неструктурированные. В этом случае появляется возможность варьировать подробность сетки по расчетной области, тем самым сократив и время на излишне точный обсчет некоторых областей, и оперативную память, освобожденную от хранения невостребованно подробных полей величин. Также привлекательной чертой является абстрагирование численных методов от геометрии расчетной области и практическое снятие требований к ней. Однако, такие программы значительно сложнее по своей структуре. При работе с регулярными сетками отношение соседства, равно как и пространственные координаты, не приходилось хранить явно, так как эти свойства и величины напрямую связывались с многомерными индексными пространствами массивов величин.

Модель DVMH [1, 2] построена на парадигме параллелизма по данным. В основе этой модели лежит понятие распределенного многомерного массива. При этом у каждого процессора имеется 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 производится посредством их отображения на распределенные массивы, и вследствие заранее известных смещений по индексам используемых массивов величин, 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 к форме, независимой 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 (A_1:B_1, A_2:B_2, A_3:B_3, ..., A_n:B_n).

В данной статье представлены новые возможности DVM-системы[3], нацеленные на борьбу с этим ограничением. В первой главе описано расширение DVM-языков с целью введения новых типов распределенных массивов, параллельных циклов и иных вспомогательных конструкций, которые позволяют существенно упростить распараллеливание на кластер приложений на нерегулярных сетках. Во второй главе представлены 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 центрального процессора или графического ускорителя.

1. Новые возможности для работы с нерегулярными сетками

Для работы с нерегулярными сетками в DVM-системе введен 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.

Добавлены 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 одного 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
выражениях 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 локальной 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 появляется 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, см. 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 не обязательно происходят с соседними процессорами, а с произвольным подмножеством процессоров) этот набор расширений позволяет распараллелить и запустить приложения на нерегулярных сетках на кластер с ускорителями.

Для иллюстрации возможностей DVM-системы для работы с неструктурированными сетками 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-ый элемент шаблона оказывается на том процессоре, индекс которого указан в карте на i-ой 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-ый 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 негранничных 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, а 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-ый 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 негранничных 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 – 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-ым 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-ый виток цикла выполняется на том процессоре, где находится элемент B(i), а 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 максимально значение eps по итогам всех витков цикла, а не только витков этого процессора. Клауза private(tmp) говорит о том, что переменная tmp

приватная, а 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.

2. Новые возможности для дополнительного распараллеливания существующих 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, а 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 и трюков и 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):

```
!DVM$ PARALLEL(I,J,K) REDUCTION (MAX(EPS))   ! Для языка Fortran-DVMH
```

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

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, то 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): с использованием OpenMP или без, возможность задания привязки нитей;
- 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 (распределенные массивы имеют набор ограничений при работе с ними, а при таком подходе их заводить необязательно) получить программу для многоядерного центрального процессора и графического ускорителя, а также оценить перспективы ускорения целевой программы на кластере с многоядерными процессорами и ускорителями.

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.

В статье 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[4], 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.

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проекты 19-07-00889-а и 19-07-01101-а.

Литература

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. 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 .

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. 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. vol. 8, no. 1. pp. 89-106. (in Russian) DOI: 10.14529/cmse190106 .