Распараллеливание алгоритма Якоби на Fortran-DVMH

Первое знакомство с DVM-системой лучше всего начать с небольшого примера.

Возьмем маленькую программу, которая совершает какие-то преобразования матрицы. В принципе для нас сейчас не имеет значения смысл этих преобразований, но мы хотим, чтобы они выполнялись быстрее, а результат вычислений остался таким же. Как этого добиться? – – распараллелить программу. И, естественно, для этого мы будем использовать технологию DVMH.

Рассмотрим исходную программу. Полные исходные коды можно посмотреть в конце этой статьи. Все вычисления производятся над массивом B, кроме него есть вспомогательный массив A:

```
1       program jac
2       parameter (L=16000, M=6250, ITMAX=200)
3       real A(L,M), eps, MAXEPS, B(L,M)

        ...
8       MAXEPS = 0.5e-7
```

Перед началом вычислений массив B инициализируется:

```
10      do j = 1, M
11        do i = 1, L
12          A(i,j) = 0.
13          if(i.eq.1 .or. j.eq.1 .or. i.eq.L .or. j.eq.M) then
14            B(i,j) = 0.
15          else
16            B(i,j) = (1. + i + j)
17          endif
18        enddo
19      enddo
```

После чего применяется итерационный алгоритм Якоби:

```
23      do it = 1, ITMAX
24        eps = 0.
25
26 !      variable eps is used for calculation of maximum value
27        do j =  2, M-1
28          do i = 2, L-1
29            eps = max(eps, abs(B(i,j) - A(i,j)))
30            A(i,j) = B(i,j)
31          enddo
32        enddo
33
34        do j = 2, M-1
35          do i = 2, L-1
36            B(i,j) = (A(i-1,j) + A(i,j-1) + A(i+1,j) + A(i,j+1)) / 4
37          enddo
38        enddo
39
40        print 200, it, eps
41 200    format(' it = ', i4, '   eps = ', e14.7)
42        if (eps .lt. MAXEPS) goto 3
43      enddo
44    3 continue
```

Чтобы получить работающую DVM-программу, необходимо сделать две вещи:

- распределить данные;
- распределить вычисления над этими данными.

Поскольку DVM-программа может выполняться на кластере, то можно себе представить, что наша программа будет состоять из отдельных mpi-процессов, запущенных на разных узлах этого кластера. И наша задача раздать каждому mpi-процессу свою порцию данных и свою часть работы.

Чтобы осуществить первое (распределить данные) достаточно добавить директивы:

```
3         real A(L,M), eps, MAXEPS, B(L,M)

          ...

7   !dvm$ distribute (block, block) :: A
8   !dvm$ align B(i,j) with A(i,j)
9   !     arrays A and B  with block distribution
```

Это означает, что массив A будет равномерно разделен на части по двум измерениям и полученные подматрицы будут розданы mpi-процессам. Т.е. каждый mpi-процесс получит свою подматрицу A(i1:i2, j1:j2). Кроме того, массив B будет распределен точно таким же образом и каждая подматрица B(i1:i2, j1:j2) попадет к тому же самому mpi-процессу, что и соответствующая подматрица A(i1:i2, j1:j2).

Теперь, чтобы распределить вычисления, нужно найти участки кода, в которых используются распределенные данные. В нашем случае есть три двумерных цикла, которые работают с массивами A и B.

Начнем с цикла инициализации. Этот цикл выполняется только один раз в начале программы и не занимает много времени, но после того, как мы распределили массивы, каждый mpi-процесс должен следить, чтобы у него не происходило обращений к элементам массива, отданным другим mpi-процессам. Поэтому теперь этот цикл будет выполняться гораздо медленнее и его неплохо бы распараллелить. Для этого достаточно добавить директиву **parallel**:

```
15  !dvm$ parallel (j,i) on A(i,j)
16  !     nest of two parallel loops, iteration (i,j) will be executed on
17  !     processor, which is owner of element A(i,j)
18        do j = 1, M
19          do i = 1, L
20            A(i,j) = 0.
21            if(i.eq.1 .or. j.eq.1 .or. i.eq.L .or. j.eq.M) then
22              B(i,j) = 0.
23            else
24              B(i,j) = 1. + i + j
25            endif
26          enddo
27        enddo
```

Это означает, что итерацию (j,i) должен выполнять только тот mpi-процесс, которому был отдан элемент A(i,j). Таким образом, mpi-процессы, работая параллельно, будут выполнять предназначенные только им итерации цикла.

Далее, нужно аналогичным образом распараллелить циклы внутри алгоритма Якоби:

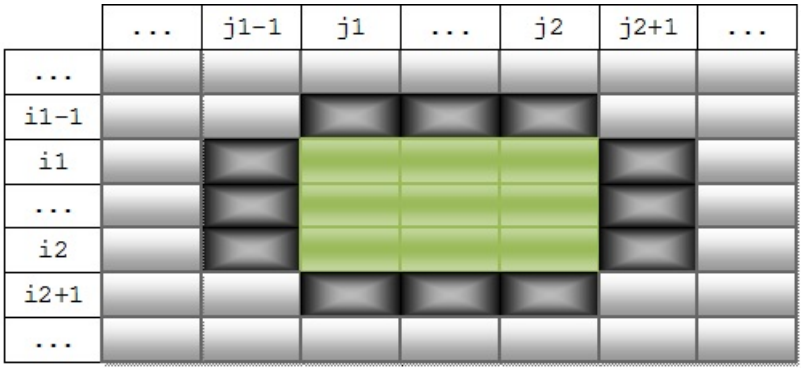
```
37  !dvm$    parallel (j,i) on A(i,j), reduction(max(eps))
38  !        variable eps is used for calculation of maximum value
39          do j =  2, M-1
40            do i = 2, L-1
41              eps = max(eps, abs(B(i,j) - A(i,j)))
```

```
42              A(i,j) = B(i,j)
43            enddo
44          enddo
45
46  !dvm$   parallel (j,i) on B(i,j), shadow_renew(A)
47  !       copying shadow elements of array A from
48  !       neighbouring processors before loop execution
49          do j = 2, M-1
50            do i = 2, L-1
51              B(i,j) = (A(i-1,j) + A(i,j-1) + A(i+1,j) + A(i,j+1)) / 4
52            enddo
53          enddo
```

В этих двух циклах пришлось использовать дополнительные указания **reduction** и **shadow_renew**.

В первом цикле встречается переменная *eps*, которая должна содержать максимальное из вычисленных значений на каждой итерации. Мы знаем, что mpi-процесс выполняет только предназначенные ему итерации параллельного цикла. Следовательно, после первого цикла его переменная *eps* будет иметь максимальное значение, вычисленное только среди его итераций. Получается, что для вычисления максимума на всех итерациях, нужно после цикла найти максимум среди *eps* всех mpi-процессов. Этим и занимается операция **reduction(max(*eps*))**, которая после цикла выполнит необходимые действия, и все mpi-процессы будут иметь одинаковые значения собственных переменных *eps*, равные максимальному среди вычисленных на всех итерациях цикла.

Во втором цикле происходит обращение из итерации (j,i) к элементам массива A(i-1,j), A(i,j-1) и так далее. Т.е. mpi-процесс, который владеет элементом A(i,j) и выполняет итерацию (j,i), пытается обратиться к элементу A(i-1,j), который может принадлежать соседнему mpi-процессу. Поскольку mpi-процесс не может напрямую обратиться в память другого mpi-процесса, ему приходится предварительно копировать данные к себе. Такое копирование можно обеспечить указанием **shadow_renew**. Если mpi-процесс владеет подматрицей A(i1:i2,j1:j2), то **shadow_renew(*A*)** до начала цикла скопирует ему от других mpi-процессов подматрицы A(i1-1,j1:j2), A(i2+1,j1:j2), A(i1:i2,j1-1), A(i1:i2,j2+1).



Область, образуемая этими подматрицами, называется теневыми гранями. Изменить размер теневых граней можно директивой **shadow**, но здесь этого не требуется, поскольку текущий размер нас устраивает.

На данном этапе у нас уже получилась DVM-программа, которую можно запустить, например, на кластере и увидеть, что программа ускорилась. Однако, она использует только MPI для параллельного выполнения, так что, если на этом кластере имеются графические ускорители (GPU), то они останутся не задействованы. Чтобы задействовать и их, нужно совсем немного,

а именно, указать, какие участки программы следует выполнять на GPU и, при необходимости, указать, какие данные нужно скопировать из оперативной памяти в память GPU или обратно.

Регионы (участки кода, которые могут быть выполнены на GPU) должны содержать параллельные циклы. Если взглянуть на нашу программу, то видно, что параллельные циклы просты и могут быть помещены в регионы. Что мы и делаем:

```
14 !dvm$ region out(A, B)
15 !dvm$ parallel (j,i) on A(i,j)
          ...
27        enddo
28 !dvm$ end region
          ...
36 !dvm$    region
37 !dvm$    parallel (j,i) on A(i,j), reduction(max(eps))
            ...
44          enddo
45
46 !dvm$    parallel (j,i) on B(i,j), shadow_renew(A)
            ...
53          enddo
54 !dvm$    end region
```

Расставив регионы, нужно обеспечить копирование данных между оперативной памятью и памятью GPU, чтобы обновленные данные были перемещены туда, где они нужны.

Для цикла инициализации ничего делать не нужно: массивы A и B будут созданы и заполнены на графическом ускорителе. Далее идет регион в алгоритме Якоби, который работает с этими массивами, которые уже расположены в памяти GPU, поэтому их копировать тоже не нужно. Однако, в регионе вычисляется значение *eps*, которое после региона печатается на экран, так что необходимо скопировать *eps* из региона в оперативную память. Делается это с помощью директивы **get_actual**:

```
53          enddo
54 !dvm$    end region
55
56 !dvm$    get_actual(eps)
57          print 200, it, eps
```

Теперь будет напечатано значение *eps*, посчитанное на GPU, а не 0, записанный до начала региона. На следующей итерации *it*, снова выполняется этот же регион. Переменная *eps* уже встречалась раньше и сохранила свое значение на GPU, так что при чтении будет использоваться оно, а не то, что мы присвоили перед регионом. Чтобы исправить ситуацию необходимо сообщить, что новое значение переменной *eps* нужно взять из оперативной памяти. Сделать это можно с помощью директивы **actual**:

```
33          eps = 0.
34 !dvm$    actual(eps)
35
36 !dvm$    region
```

У нас получилась работающая реализация алгоритма Якоби, которая может выполняться на кластере с GPU, но вычисленная матрица B по-прежнему располагается в памяти GPU. Если мы хотим ее, например, распечатать, то ее нужно вернуть в оперативную память с помощью директивы **get_actual**:

```
60        enddo
61      3 continue
62
63  !dvm$ get_actual(B)
```

Вот теперь все. Мы получили окончательную версию DVMH-программы.

Запуск DVMH-программы

Попробуем скомпилировать и запустить полученную DVMH-программу. Для этого нам понадобится вычислительная машина, на которой установлена DVM-система. В нашем случае этой машиной является ПК с процессором Intel Core i7-3770 3.40GHz и графической платой GeForce GTX Titan.

В директорию, где находится наша программа jac2d.fdv, следует скопировать dvm скрипт из директории с установленной DVM-системой.

```
$ ls
dvm   jac2d.fdv
```

Теперь можно скомпилировать нашу программу.

```
$ ./dvm f jac2d.fdv
$ ls
dvm   jac2d   jac2d.fdv
```

В текущей директории появился исполняемый файл jac2d, который можно запустить с помощью скрипта dvm, но предварительно надо настроить параметры запуска, отредактировав dvm скрипт.

```
#---------------- DVMH options:
export DVMH_PPN='1' # Number of processes per node
export DVMH_NUM_THREADS='0' # Number of CPU threads per process
export DVMH_NUM_CUDAS='1' # Number of GPUs per process
```

Задав такие параметры, мы сообщаем, что хотим выполнять регионы только на одном графическом ускорителе. Всё, можем запускать:

```
$ ./dvm run jac2d
...
 IT =   197    EPS =   0.4086035E+02
 IT =   198    EPS =   0.4062500E+02
 IT =   199    EPS =   0.4044824E+02
 IT =   200    EPS =   0.4021777E+02
 time =     3.176400
```

Программа успешно отработала за 3.17 секунды. Для сравнения, можно скомпилировать этот же исходный текст обычным компилятором, получить последовательную программу и запустить ее:

```
$ cp jac2d.fdv jac2d.f
$ ifort -O3 jac2d.f
$ ./a.out
...
 IT =   197    EPS =   0.4086035E+02
 IT =   198    EPS =   0.4062500E+02
 IT =   199    EPS =   0.4044824E+02
```

```
 IT =  200    EPS =  0.4021777E+02
 time =     27.48160
```

Получили те же самые результаты, но только вычисления выполнялись гораздо дольше.

Исходные тексты программ

Последовательная версия программы

```
      program jac
      parameter (L=16000, M=6250, ITMAX=200)
      real A(L,M), eps, MAXEPS, B(L,M)

      integer clock_rate, clock_start, clock_end

      print *, '**********  test_jacobi   **********'
      MAXEPS = 0.5e-7

      do j = 1, M
        do i = 1, L
          A(i,j) = 0.
          if(i.eq.1 .or. j.eq.1 .or. i.eq.L .or. j.eq.M) then
            B(i,j) = 0.
          else
            B(i,j) = (1. + i + j)
          endif
        enddo
      enddo

      call system_clock(count_rate = clock_rate)
      call system_clock(count = clock_start)
      do it = 1, ITMAX
        eps = 0.

!       variable eps is used for calculation of maximum value
        do j =  2, M-1
          do i = 2, L-1
            eps = max(eps, abs(B(i,j) - A(i,j)))
            A(i,j) = B(i,j)
          enddo
        enddo

        do j = 2, M-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
          enddo
        enddo

        print 200, it, eps
200     format(' it = ', i4, '   eps = ', e14.7)
        if (eps .lt. MAXEPS) goto 3
      enddo
    3 continue
```

```
45
46        call system_clock(count=clock_end)
47        print *, 'time = ', real(clock_end - clock_start) / clock_rate
48
49        end
```

DVMH-версия программы

```
1         program jac
2         parameter (L=16000, M=6250, ITMAX=200)
3         real A(L,M), eps, MAXEPS, B(L,M)
4
5         integer clock_rate, clock_start, clock_end
6
7   !dvm$ distribute (block, block) :: A
8   !dvm$ align B(i,j) with A(i,j)
9   !     arrays A and B  with block distribution
10
11        print *, '**********  test_jacobi   **********'
12        MAXEPS = 0.5e-7
13
14  !dvm$ region out(A, B)
15  !dvm$ parallel (j,i) on A(i,j)
16  !     nest of two parallel loops, iteration (i,j) will be executed on
17  !     processor, which is owner of element A(i,j)
18        do j = 1, M
19          do i = 1, L
20            A(i,j) = 0.
21            if(i.eq.1 .or. j.eq.1 .or. i.eq.L .or. j.eq.M) then
22              B(i,j) = 0.
23            else
24              B(i,j) = 1. + i + j
25            endif
26          enddo
27        enddo
28  !dvm$ end region
29
30        call system_clock(count_rate = clock_rate)
31        call system_clock(count = clock_start)
32        do it = 1, ITMAX
33          eps = 0.
34  !dvm$   actual(eps)
35
36  !dvm$   region
37  !dvm$   parallel (j,i) on A(i,j), reduction(max(eps))
38  !       variable eps is used for calculation of maximum value
39          do j =  2, M-1
40            do i = 2, L-1
41              eps = max(eps, abs(B(i,j) - A(i,j)))
42              A(i,j) = B(i,j)
43            enddo
44          enddo
45
```

```
!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, M-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
           enddo
         enddo
!dvm$    end region

!dvm$    get_actual(eps)
         print 200, it, eps
200      format(' it = ', i4, '   eps = ', e14.7)
         if (eps .lt. MAXEPS) goto 3
       enddo
     3 continue

!dvm$ get_actual(B)
       call system_clock(count=clock_end)
       print *, 'time = ', real(clock_end - clock_start) / clock_rate

       end
```