# Распараллеливание алгоритма Якоби на C-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:

```
7  #define L 16000
8  #define M 6250
9  #define ITMAX 200
10
11 float MAXEPS = 0.5f;
12
13 float A[L][M];
14 float B[L][M];
```

Перед началом вычислений массив B инициализируется:

```
22   for (i = 0; i < L; i++)
23     for (j = 0; j < M; j++)
24     {
25       A[i][j] = 0;
26       if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
27         B[i][j] = 0;
28       else
29         B[i][j] = 3 + i + j;
30     }
```

После чего применяется итерационный алгоритм Якоби:

```
35   for (it = 1; it <= ITMAX; it++)
36   {
37     eps = 0;
38
39     for (i = 1; i < L - 1; i++)
40       for (j = 1; j < M - 1; j++)
41       {
42         float tmp = fabs(B[i][j] - A[i][j]);
43         eps = Max(tmp, eps);
44         A[i][j] = B[i][j];
45       }
46
47     for (i = 1; i < L - 1; i++)
48       for (j = 1; j < M - 1; j++)
49         B[i][j] = (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]) / 4.0f;
50
51     printf(" IT = %4i   EPS = %14.7E\n", it, eps);
52     if (eps < MAXEPS)
53       break;
54   }
```

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:

```
13 #pragma dvm array distribute [block][block]
14 float A[L][M];
15 #pragma dvm array align([i][j] with A[i][j])
16 float B[L][M];
```

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**:

```
26   #pragma dvm parallel([i][j] on A[i][j])
27   for (i = 0; i < L; i++)
28     for (j = 0; j < M; j++)
29     {
30       A[i][j] = 0;
31       if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
32         B[i][j] = 0;
33       else
34         B[i][j] = 3 + i + j;
35     }
```

Это означает, что итерацию (i,j) 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:

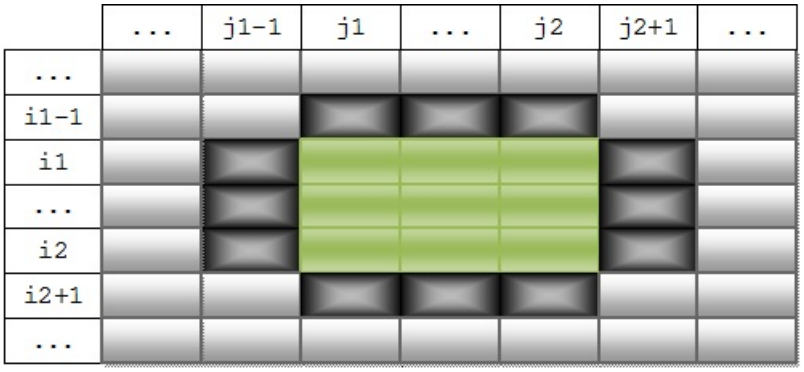
```
49     #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
50     for (i = 1; i < L - 1; i++)
51       for (j = 1; j < M - 1; j++)
52       {
53         float tmp = fabs(B[i][j] - A[i][j]);
54         eps = Max(tmp, eps);
55         A[i][j] = B[i][j];
56       }
57
58     #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
```

```
59      for (i = 1; i < L - 1; i++)
60        for (j = 1; j < M - 1; j++)
61          B[i][j] = (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]) / 4.0f;
```

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 (i,j) 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 (i,j), 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:

```
24    #pragma dvm region
25    {
26    #pragma dvm parallel([i][j] on A[i][j])
27    for (i = 0; i < L; i++)
```

```
28      for (j = 0; j < M; j++)
29      {
          ...
35      }
36    }
    ...
47      #pragma dvm region
48      {
49      #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
          ...
58      #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
          ...
62      }
```

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**:

```
62      }
63
64      #pragma dvm get_actual(eps)
65      printf(" IT = %4i   EPS = %14.7E\n", 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**:

```
43      eps = 0;
44
45      #pragma dvm actual (eps)
46
47      #pragma 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**:

```
64      #pragma dvm get_actual(eps)
65      printf(" IT = %4i   EPS = %14.7E\n", it, eps);
66      if (eps < MAXEPS)
67        break;
68    }
69
```

```
70    et = get_time();
71
72    #pragma 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.cdv, следует скопировать dvm скрипт из директории с установленной DVM-системой.

```
$ ls
dvm   jac2d.cdv   time.h
```

Теперь можно скомпилировать нашу программу.

```
$ ./dvm c jac2d.cdv
$ ls
dvm   jac2d   jac2d.cdv   time.h
```

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 =   4.0860352E+01
 IT =   198    EPS =   4.0625000E+01
 IT =   199    EPS =   4.0448242E+01
 IT =   200    EPS =   4.0217773E+01
 Size            =     16000 x   6250
 Iterations      =                200
 Operation type  =     floating point
time = 3.093755
```

Программа успешно отработала за 3.09 секунды. Для сравнения, можно скомпилировать этот же исходный текст обычным компилятором, получить последовательную программу и запустить ее:

```
$ cp jac2d.cdv jac2d.c
$ icc -O3 jac2d.c
$ ./a.out
...
 IT =   197    EPS =   4.0860352E+01
```

```
 IT =   198    EPS =   4.0625000E+01
 IT =   199    EPS =   4.0448242E+01
 IT =   200    EPS =   4.0217773E+01
 Size            =     16000 x   6250
 Iterations      =                200
 Operation type  =     floating point
time = 27.277828
```

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

# Исходные тексты программ

## Последовательная версия программы

```
1  #include <math.h>
2  #include <stdio.h>
3  #include "time.h"
4
5  #define Max(a, b) ((a) > (b) ? (a) : (b))
6
7  #define L 16000
8  #define M 6250
9  #define ITMAX 200
10
11 float MAXEPS = 0.5f;
12
13 float A[L][M];
14 float B[L][M];
15
16 int main(int an, char **as)
17 {
18   double st, et;
19   int i, j, it;
20   float eps;
21
22   for (i = 0; i < L; i++)
23     for (j = 0; j < M; j++)
24     {
25       A[i][j] = 0;
26       if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
27         B[i][j] = 0;
28       else
29         B[i][j] = 3 + i + j;
30     }
31
32   st = get_time();
33
34   /* iteration loop */
35   for (it = 1; it <= ITMAX; it++)
36   {
37     eps = 0;
38
39     for (i = 1; i < L - 1; i++)
```

```
      for (j = 1; j < M - 1; j++)
      {
        float tmp = fabs(B[i][j] - A[i][j]);
        eps = Max(tmp, eps);
        A[i][j] = B[i][j];
      }

    for (i = 1; i < L - 1; i++)
      for (j = 1; j < M - 1; j++)
        B[i][j] = (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]) / 4.0f;

    printf(" IT = %4i   EPS = %14.7E\n", it, eps);
    if (eps < MAXEPS)
      break;
  }

  et = get_time();

  printf(" Size            =    %6d x %6d\n", L, M);
  printf(" Iterations      =       %12d\n", ITMAX);
  printf(" Operation type  =     floating point\n");

  printf("time = %f\n", et - st);
  return 0;
}
```

## DVMH-версия программы

```
#include <math.h>
#include <stdio.h>
#include "time.h"

#define Max(a, b) ((a) > (b) ? (a) : (b))

#define L 16000
#define M 6250
#define ITMAX 200

float MAXEPS = 0.5f;

#pragma dvm array distribute [block][block]
float A[L][M];
#pragma dvm array align([i][j] with A[i][j])
float B[L][M];

int main(int an, char **as)
{
  double st, et;
  int i, j, it;
  float eps;

  #pragma dvm region
  {
```

```
26    #pragma dvm parallel([i][j] on A[i][j])
27    for (i = 0; i < L; i++)
28      for (j = 0; j < M; j++)
29      {
30        A[i][j] = 0;
31        if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
32          B[i][j] = 0;
33        else
34          B[i][j] = 3 + i + j;
35      }
36    }
37
38    st = get_time();
39
40    /* iteration loop */
41    for (it = 1; it <= ITMAX; it++)
42    {
43      eps = 0;
44
45      #pragma dvm actual (eps)
46
47      #pragma dvm region
48      {
49      #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
50      for (i = 1; i < L - 1; i++)
51        for (j = 1; j < M - 1; j++)
52        {
53          float tmp = fabs(B[i][j] - A[i][j]);
54          eps = Max(tmp, eps);
55          A[i][j] = B[i][j];
56        }
57
58      #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
59      for (i = 1; i < L - 1; i++)
60        for (j = 1; j < M - 1; j++)
61          B[i][j] = (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]) / 4.0f;
62      }
63
64      #pragma dvm get_actual(eps)
65      printf(" IT = %4i   EPS = %14.7E\n", it, eps);
66      if (eps < MAXEPS)
67        break;
68    }
69
70    et = get_time();
71
72    #pragma dvm get_actual (B)
73
74    printf(" Size            =    %6d x %6d\n", L, M);
75    printf(" Iterations      =       %12d\n", ITMAX);
76    printf(" Operation type  =     floating point\n");
77
78    printf("time = %f\n", et - st);
```

```
  return 0;
}
```

### Файл time.h

```
#ifndef _TIME_H_
#define _TIME_H_

//##### get_time() implementation #####

#ifdef _WIN32
#include <windows.h>

double get_time()
{
  FILETIME systime;
  ULARGE_INTEGER utime;
  GetSystemTimeAsFileTime(&systime); // resolution 100 nanoseconds
  utime.LowPart = systime.dwLowDateTime;
  utime.HighPart = systime.dwHighDateTime;

  return utime.QuadPart / 10000000.0;
}
#else
#include <sys/time.h>

double get_time()
{
  struct timeval tv;
  gettimeofday(&tv, NULL);
  return tv.tv_sec + (tv.tv_usec)/1000000.0;
}
#endif

#endif
```