Parallelization of Jacobi algorithm in C-DVMH

It is best of all to start first acquaintance with DVM system with a small example.

Let's consider small program performing some conversions of a matrix. In principle the sense of conversions doesn't matter for us now, but we want they will be executed quickly, and the result of computations will remain the same. How to achieve it? — to parallelize the program. And, of course, for this purpose we will use the DVMH technology.

Consider the source program. Full source codes are available at the end of this article. All computations are done over array B and there is an auxiliary array of A except it:

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

float MAXEPS = 0.5f;

float A[L][M];
float B[L][M];
```

Before start of computations the array B is initialized:

```
  for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
    {
      A[i][j] = 0;
      if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
        B[i][j] = 0;
      else
        B[i][j] = 3 + i + j;
    }
```

Then the iterative Jacobi algorithm is applied:

```
  for (it = 1; it <= ITMAX; it++)
  {
    eps = 0;

    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;
  }
```

To obtain ready-to-run DVM program, it is necessary to do two things:

- to distribute data

- to distribute computations over these data.

As the DVM program can be executed on a cluster, it is possible to imagine that our program will consist of the separate MPI processes launched on different nodes of the cluster. And our task distributes to each MPI process its own portion of data and its own part of job.

To realize the first (to distribute data) it is enough to add directives:

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

It means that the array A will be uniformly partitioned into parts along two dimensions and obtained submatrixes will be distributed over MPI processes. That is each MPI process will receive submatrix A(i1:i2, j1:j2). Moreover, the array B will be distributed in the same way and each submatrix B(i1:i2, j1:j2) will be received by the same MPI process, as the appropriate submatrix A(i1:i2, j1:j2).

Now, to distribute computations, it is necessary to find code fragments where the distributed data are used. In our case there are three two-dimensional loops which operate with arrays A and B.

Let's begin with an initialization loop. This loop is executed only once at the beginning of the program and doesn't take a lot of time but when we distributed arrays, each MPI process must check there will be no access to array elements, belonging to other MPI processes. Therefore this loop will be executed more slowly and it is quite good to parallelize it. For this purpose it is enough to add **parallel** directive:

```
  #pragma dvm parallel([i][j] on A[i][j])
  for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
    {
      A[i][j] = 0;
      if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
        B[i][j] = 0;
      else
        B[i][j] = 3 + i + j;
    }
```

It means that the iteration (i,j) should be executed only by that MPI process the element A(i,j) belongs to. Thus, MPI processes, working in parallel, will execute only intended to them iterations of the loop.

Further, it is necessary to parallelize loops inside Jacobi algorithm similarly:

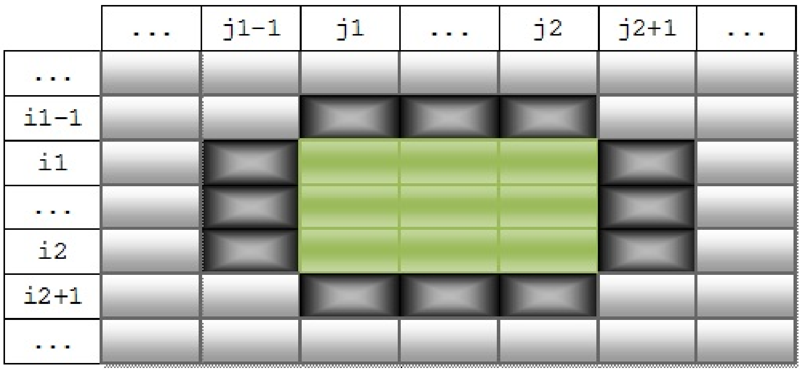
```
    #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
    for (i = 1; i < L - 1; i++)
      for (j = 1; j < M - 1; j++)
      {
        float tmp = fabs(B[i][j] - A[i][j]);
        eps = Max(tmp, eps);
        A[i][j] = B[i][j];
      }

    #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
    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;
```

There were necessary to use additional clauses **reduction** and **shadow_renew** in these two loops.

In the first loop variable *eps* meets which should contain maximum of the calculated values on each iteration. We know that MPI process executes only its own iterations of parallel loop. Therefore, after the first loop execution its *eps* variable will have the maximum value calculated only among its iterations. Thus to compute a maximum of *eps* among all iterations, it is necessary after the loop finishing to find the maximum among *eps* of all MPI processes. The clause **reduction(max(***eps***))** does it after the loop finishing and perform all necessary actions and all MPI processes will have identical values of own *eps* variables, equal maximum among calculated on all iterations of the loop.

In the second loop there is access from iteration (i,j) to the array elements A(i-1,j), A(i,j-1) and so on. So, MPI process which owns an element A(i,j) and executes iteration (i,j), tries to access to the element A(i-1,j) which can belong to adjacent MPI process. As MPI process can't access directly to memory of other MPI process, it should preliminarily copy data to itself. Such copying can be provided by specifying **shadow_renew**. If MPI process owns submatrixes A(i1:i2,j1:j2), **shadow_renew(***A***)** will copy to it submatrixes A(i1-1,j1:j2), A(i2+1,j1:j2), A(i1:i2,j1-1), A(i1:i2,j2+1) before loop beginning.



The area formed by these submatrixes is called shadow edges. It is possible to change the size of shadow edges by **shadow** directive, but here it isn't required as the current size suits us.

At this step we have DVM program which we can launched, for example, on a cluster and we can see that the program is speed up. However, the program uses only MPI for parallel execution, but if this cluster has graphic accelerators (GPU), they will remain unused. To use them, it is required very little, namely, to specify, what fragments of the program should be executed on GPU and, if necessary, to specify, what data should be copied from a random access memory in GPU memory or back.

Regions (code fragments which can be executed on GPU) should contain parallel loops. If to look at our program, it is visible that the parallel loops are simple and can be included into regions. We do it:

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

     ...

    }
  }

 ...

    #pragma dvm region
    {
    #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
     ...
    #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
     ...
    }
```

After specifying regions in the program, it is necessary to supply data copying between random access memory and memory of GPU to guarantee that updated data will be move there where they are necessary.

For initialization loop it's nothing to do: arrays A and B will be created and initiated on the graphic accelerator. Next region is the region in Jacobi algorithm which operates with these arrays already located in GPU memory, therefore they don't need to be copied too. However, *eps* value is calculated in the region and after the region *eps* is printed on the screen. So it is necessary to copy *eps* from the region to random access memory. Directive **get_actual** is used for this purpose:

```
    }

    #pragma dvm get_actual(eps)
    printf(" IT = %4i   EPS = %14.7E\n", it, eps);
```

Now *eps* value calculated on GPU will be printed, but not 0 written prior to the region beginning. On the following iteration *it*, the same region is executed again. The *eps* variable already met earlier and kept its value on GPU so this value will be used when reading, but not value, assigned before the region. To correct a situation it is necessary to report that new value of variable *eps* should be taken from a random access memory. Directive **actual** can be used for it:

```
    eps = 0;

    #pragma dvm actual (eps)

    #pragma dvm region
```

We obtain working implementation of Jacobi algorithm which can be executed on a cluster with GPU, but the calculated matrix B still locates in GPU memory. If we want to print it, for example, it needs to be returned to a random access memory by **get_actual** directive:

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

  et = get_time();

  #pragma dvm get_actual (B)
```

Now all. We obtained the final version of the DVMH program.

Startup of DVMH program

We will try to compile and launch the obtained DVMH program. To do that we need the computer on which the DVM system is installed. We have a PC with processor Intel Core i7-3770 3.40GHz and graphics card GeForce GTX Titan.

In a directory where our program jac2d.cdv is located, it is necessary to copy dvm script from the directory where DVM system is installed.

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

Now we can to compile our program.

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

The executable file jac2d appears in the current directory. It can be launched using dvm script, but previously it is necessary to set start parameters, having edited dvm script.

```
#--------------- 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
```

Setting such parameters, we report that we want to execute regions only on one graphic accelerator. Now we can launch the program:

```
$ ./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
```

The program successfully performed in 3.09 seconds. For comparing, we can to compile the same source text by usual compiler, to obtain the sequential program and to launch it:

```
$ 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
```

The results of execution are the same, but execution time was much more long.

Source code of programs

Serial version of the program

```c
#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;

float A[L][M];
float B[L][M];

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

  for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
    {
      A[i][j] = 0;
      if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
        B[i][j] = 0;
      else
        B[i][j] = 3 + i + j;
    }

  st = get_time();

  /* iteration loop */
  for (it = 1; it <= ITMAX; it++)
  {
    eps = 0;

    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 version of the program

```
#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
  {
  #pragma dvm parallel([i][j] on A[i][j])
  for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
    {
      A[i][j] = 0;
      if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
        B[i][j] = 0;
      else
        B[i][j] = 3 + i + j;
    }
  }

  st = get_time();

```

```c
  /* iteration loop */
  for (it = 1; it <= ITMAX; it++)
  {
    eps = 0;

    #pragma dvm actual (eps)

    #pragma dvm region
    {
    #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps))
    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];
      }

    #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A)
    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;
    }

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

  et = get_time();

  #pragma dvm get_actual (B)

  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;
}
```

File time.h

```c
#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
```