Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №9 по курсу «Программирование графических процессоров»
(Лабораторная работа №3 по курсу «Параллельная обработка данных»)

Технология MPI и технология OpenMP.

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

Цель работы: Совместное использование технологии MPI и технологии OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант 2:
Распараллеливание в общем виде с разделением работы между нитями вручную (в стиле CUDA).
Входные данные: На первой строке заданы три числа: размер сетки процессов. Гарантируется, что при запуске программы количество процессов будет равно произведению этих трех чисел. На второй строке задается размер блока, который будет обрабатываться одним процессом: три числа. Далее задается путь к выходному файлу, в который необходимо записать конечный результат работы программы и точность ϵ. На последующих строках описывается задача: задаются размеры области, l_x, l_y, l_z, граничные условия: $u_{down}, u_{up}, u_{left}, u_{right}, u_{front}, u_{back}$ и начальное значение u_0.

Выходные данные: В файл, определенный во входных данных, необходимо напечатать построчно значения $(u_{1,1,1}, u_{2,1,1}, \ldots, u_{1,2,1}, u_{2,2,1}, .., , u_{n_x-1,n_y,n_z}, u_{n_x,n_y,n_z})$ в ячейках сетки в формате с плавающей запятой с семью знаками мантиссы.

Программное и аппаратное обеспечение:
Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024
multiProcessorCount : 3
maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения

В данной лабораторной работе нужно решить задачу, описанную в лабораторной работе 7. Одна итерация решения исходной задачи состоит из трех этапов.
На первом этапе происходит обмен граничными слоями между процессами.
На втором этапе выполняется обновление значений во всех ячейках.
И третий этап заключается в вычислении погрешности: сначала локально в рамках каждого процесса, а потом через обмены и во всей области.

В соответствие с вариантом задания по технологии OpenMP нужно выполнить распараллеливание в общем виде с разделением работы между нитями вручную (в стиле CUDA). Распараллеливание можно применить на втором этапе (при обновении значений в ячейках).

Описание программы

С помощью директивы #pragma omp parallel создатся параллельный регион для следующего за ней структурированного бока. Директива parallel указывает, что структурированный блок кода должен быть выполнен параллельно в несколько потоков. Таким образом, для каждого из процессов участок кода, отвечающий за обновление значений в ячейках будет выполняться в многопоточном режиме. Вместо трех циклов теперь реализован один цикл и специальная функция, которая отвечает за обновление итерационных координат i, j, k.

Для расчета значения, отвечающего за сходимость, нужно создать массив, в котором количество ячеек равно максимальному значению числа нитей в текущем параллельном участке. В каждой ячейке этого массива будет хранится максимальное значение, полученное каждым из потоков. Позже из этого массива находится максимум и дальше применяется функция MPI_Allreduce. Она имеет варианты каждой из операции редукции, где результат возвращается всем процессам группы (в нашем случае максимум).

2 Исходный код

```
#include <iostream>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include <chrono>
#include "mpi.h"
#include <omp.h>



const int axes = 3;
const int directions = 6;


void WaitAll(int* coords, int* gridProc, MPI_Request *arrOfRequests)
{
    MPI_Status tmp;

    for (int i = 0, j = 1; i < axes; ++i, j += 2)
    {
        if (coords[i] > 0)
        {
            MPI_Wait(&arrOfRequests[i * 2], &tmp);
        }
        if (coords[i] < gridProc[i] - 1)
        {
            MPI_Wait(&arrOfRequests[j], &tmp);
        }
    }
}


double FindMax(double a, double b, double max)
{
    if (fabs(a - b) > max)
    {
        max = fabs(a - b);
    }
    return max;
}


int GetPos(int i, int j, int k, int nY, int nX)
{
    return i + (j + k * nY) * nX;
}
```

```
48
49
50  void GetCoords(int* coords, int rank, int* gridProc)
51  {
52      coords[2] = rank / gridProc[0] / gridProc[1];
53      coords[1] = (rank - coords[2] * gridProc[0] * gridProc[1]) / gridProc[0];
54      coords[0] = rank - (coords[2] * gridProc[1] + coords[1]) * gridProc[0];
55  }
56
57
58  int GetRank(int* coords, int* gridProc)
59  {
60      return coords[0] + gridProc[0] * (coords[2] * gridProc[1] + coords[1]);
61  }
62
63
64  void Printer(FILE* out, double* arr, int size)
65  {
66      for (int i = 0; i < size; ++i)
67      {
68          fprintf(out, "%.6e ", arr[i]);
69      }
70      fprintf(out, "\n");
71  }
72
73
74  void WriteOut(std::string& output, int* gridProc, int* block, int* coords, double*
        grid, double* exchangeBuf, int rank)
75  {
76      MPI_Status status;
77      FILE* out;
78      int tmp[axes];
79
80      if (!rank)
81      {
82          out = fopen(output.c_str(), "w");
83      }
84
85      for(int procZ = 0; procZ < gridProc[2]; ++procZ)
86      {
87          tmp[2] = procZ;
88          for (int k = 1; k <= block[2]; ++k)
89          {
90              for (int procY = 0; procY < gridProc[1]; ++procY)
91              {
92                  tmp[1] = procY;
93                  for (int j = 1; j <= block[1]; ++j)
94                  {
95                      for (int procX = 0; procX < gridProc[0]; ++procX)
```

```
96                  {
97                      tmp[0] = procX;
98                      if (!rank)
99                      {
100                         if (coords[2] == procZ && coords[1] == procY && coords[0] ==
                                procX)
101                         {
102                             Printer(out, &grid[GetPos(1, j, k, block[1] + 2, block[0]
                                    + 2)], block[0]);
103                         }
104                         else
105                         {
106                             int rank = GetRank(tmp, gridProc);
107                             MPI_Recv(exchangeBuf, block[0], MPI_DOUBLE, rank,
108                                     0, MPI_COMM_WORLD, &status);
109                             Printer(out, exchangeBuf, block[0]);
110                         }
111                     }
112                     else
113                     {
114                         if (coords[0] == procX && coords[1] == procY && coords[2] ==
                                procZ)
115                         {
116                             MPI_Send(&grid[GetPos(1, j, k, block[1] + 2, block[0] +
                                2)],
117                                 block[0], MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); //
                                    Bsend may be better
118                         }
119                     }
120                 }
121                 MPI_Barrier(MPI_COMM_WORLD);
122             }
123         }
124     }
125 }
126 if (!rank)
127 {
128     fclose(out);
129 }
130 }
131
132
133 void FillBuffer(double* buf, int size, double val)
134 {
135     for (int i = 0; i < size; ++i)
136     {
137         buf[i] = val;
138     }
139 }
```

```
140
141
142 void InitBufsEdge(double** sendBuf, double** getBuf, int* sizeEdges, int* gridProc,
        int* coords, double* u, double u0)
143 {
144     for (int i = 0, j = 1; i < axes; ++i, j += 2)
145     {
146         sendBuf[i * 2] = new double[sizeEdges[i]];
147         getBuf[i * 2] = new double[sizeEdges[i]];
148         sendBuf[j] = new double[sizeEdges[i]];
149         getBuf[j] = new double[sizeEdges[i]];
150         FillBuffer(sendBuf[i * 2], sizeEdges[i], u0);
151         FillBuffer(sendBuf[j], sizeEdges[i], u0);
152
153         if (!coords[i])
154         {
155             FillBuffer(getBuf[i * 2], sizeEdges[i], u[i * 2]);
156         }
157         if (coords[i] == gridProc[i] - 1)
158         {
159             FillBuffer(getBuf[j], sizeEdges[i], u[j]);
160         }
161     }
162 }
163
164
165 void Clear(double* grid, double* newGrid, double** sendBuf, double** getBuf, double*
        maxValues)
166 {
167     delete[] grid;
168     delete[] newGrid;
169     delete[] maxValues;
170
171     for (int i = 0, j = 1; i < axes; ++i, j += 2)
172     {
173         delete[] sendBuf[i * 2];
174         delete[] getBuf[i * 2];
175         delete[] sendBuf[j];
176         delete[] getBuf[j];
177     }
178 }
179
180
181 void GetNeighbours(int* neighb, int* gridProc, int* coords)
182 {
183     int tmp[axes] = {coords[0], coords[1], coords[2]};
184     for (int i = 0, j = 1; i < axes; ++i, j += 2)
185     {
186         --tmp[i];
```

```
187         neighb[2 * i] = GetRank(tmp, gridProc);
188         tmp[i] += 2;
189         neighb[j] = GetRank(tmp, gridProc);
190         --tmp[i];
191     }
192 }
193 
194 
195 void Isend_Irecv(MPI_Request* in, MPI_Request* out, double** sendBuf, double** getBuf,
        int* sizeEdges, int* gridProc, int* coords, int* neighb)
196 {
197     for (int i = 0, j = 1; i < axes; ++i, j += 2)
198     {
199 
200         if (coords[i] > 0)
201         {
202             MPI_Isend(sendBuf[i * 2], sizeEdges[i], MPI_DOUBLE,
203                       neighb[i * 2], 0, MPI_COMM_WORLD, &out[i * 2]);
204 
205             MPI_Irecv(getBuf[i * 2], sizeEdges[i], MPI_DOUBLE,
206                       neighb[i * 2], 0, MPI_COMM_WORLD, &in[i * 2]);
207         }
208         if (coords[i] < gridProc[i] - 1)
209         {
210             MPI_Isend(sendBuf[j], sizeEdges[i], MPI_DOUBLE,
211                       neighb[j], 0, MPI_COMM_WORLD, &out[j]);
212             MPI_Irecv(getBuf[j], sizeEdges[i], MPI_DOUBLE,
213                       neighb[j], 0, MPI_COMM_WORLD, &in[j]);
214         }
215     }
216 }
217 
218 
219 void FillInEdges(double* grid, double** getBuf, int* block)
220 {
221     int tmp[axes] = {0};
222     ++tmp[0];
223     --tmp[0];
224     int nX = block[0] + 2, nY = block[1] + 2, nZ = block[2] + 2;
225 
226     for (int d = 0; d < directions; ++d)
227     {
228         int ax = d >> 1;
229         tmp[ax] = 1;
230         int lim = (d & 1);
231         auto idx = [nX, nY, nZ, tmp, lim](int i, int j)
232         {
233             return tmp[0] * (lim * (nX - 1) + (i + j * nY) * nX) +
234             tmp[1] * (i + (lim * (nY - 1) + j * nY) * nX) +
```

```
235             tmp[2] * (i + (j + (lim * (nZ - 1)) * nY) * nX);
236         };
237 
238         int first_n = (ax == 2) ? nY : nZ;
239         int second_n = (ax != 0) ? nX : nY;
240 
241         for (int j = 0; j < first_n; ++j)
242         {
243             for (int i = 0; i < second_n; ++i)
244             {
245                 grid[idx(i, j)] = getBuf[d][i + j * second_n];
246             }
247         }
248         tmp[ax] = 0;
249     }
250 }
251 
252 
253 
254 void FillOutEdges(double* newGrid, double** sendBuf, int* block)
255 {
256     int tmp[axes] = {0};
257     ++tmp[0];
258     --tmp[0];
259     int nX = block[0] + 2, nY = block[1] + 2, nZ = block[2] + 2;
260 
261     for (int d = 0; d < directions; ++d)
262     {
263         int ax = d >> 1;
264         tmp[ax] = 1;
265         int lim = (d & 1);
266         auto idx = [nX, nY, nZ, tmp, lim](int i, int j)
267         {
268             return tmp[0] * (lim * (nX - 3) + 1 + (i + j * nY) * nX) +
269             tmp[1] * (i + (lim * (nY - 3) + 1 + j * nY) * nX) +
270             tmp[2] * (i + (j + (lim * (nZ - 3) + 1) * nY) * nX);
271         };
272 
273         int first_n = (ax == 2) ? nY : nZ;
274         int second_n = (ax != 0) ? nX : nY;
275 
276         for (int j = 0; j < first_n; ++j)
277         {
278             for (int i = 0; i < second_n; ++i)
279             {
280                 sendBuf[d][i + j * second_n] = newGrid[idx(i, j)];
281             }
282         }
283         tmp[ax] = 0;
```

```
284     }
285 }
286
287
288 void UpdateCoords(int& i, int& j, int& k, int shift, int* block)
289 {
290     i += shift;
291     while (i > block[0])
292     {
293         i -= block[0];
294         ++j;
295     }
296     while (j > block[1])
297     {
298         j -= block[1];
299         ++k;
300     }
301 }
302
303
304
305 void Start(int* gridProc, int* block, std::string& output,
306           double eps, double* l, double* u,
307           double u0, int numProcs, int rank)
308 {
309     MPI_Bcast(gridProc, axes, MPI_INT, 0, MPI_COMM_WORLD);
310     MPI_Bcast(block, axes, MPI_INT, 0, MPI_COMM_WORLD);
311     MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
312     MPI_Bcast(l, axes, MPI_DOUBLE, 0, MPI_COMM_WORLD);
313     MPI_Bcast(u, directions, MPI_DOUBLE, 0, MPI_COMM_WORLD);
314     MPI_Bcast(&u0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
315
316
317     MPI_Request in[directions], out[directions];
318
319     int coords[axes];
320     GetCoords(coords, rank, gridProc);
321
322     int neighb[directions];
323     GetNeighbours(neighb, gridProc, coords);
324
325     double* grid = new double[(block[0] + 2) * (block[1] + 2) * (block[2] + 2)];
326     double* newGrid = new double[(block[0] + 2) * (block[1] + 2) * (block[2] + 2)];
327     FillBuffer(grid, (block[0] + 2) * (block[1] + 2) * (block[2] + 2), u0);
328     FillBuffer(newGrid, (block[0] + 2) * (block[1] + 2) * (block[2] + 2), u0);
329
330     double* sendBuf[directions];
331     double* getBuf[directions];
332
```

```
333     int sizeEdges[axes];
334     sizeEdges[0] = (block[1] + 2) * (block[2] + 2); // y z
335     sizeEdges[1] = (block[0] + 2) * (block[2] + 2); // x z
336     sizeEdges[2] = (block[0] + 2) * (block[1] + 2); // x y
337 
338     InitBufsEdge(sendBuf, getBuf, sizeEdges, gridProc, coords, u, u0);
339 
340 
341     double nX = gridProc[0] * block[0];
342     double nY = gridProc[1] * block[1];
343     double nZ = gridProc[2] * block[2];
344 
345 
346     double hX = (double)(l[0] / nX);
347     double hY = (double)(l[1] / nY);
348     double hZ = (double)(l[2] / nZ);
349 
350     int threadsMaxQuantity = omp_get_max_threads();
351     double* maxValues = new double[threadsMaxQuantity];
352     double maxConvergence = 0.0;
353 
354     double globalMax = 0.0;
355     FillBuffer(maxValues, threadsMaxQuantity, 0.0);
356     omp_set_dynamic(0);
357 
358     do {
359 
360         maxConvergence = 0.0;
361         FillBuffer(maxValues, threadsMaxQuantity, 0.0);
362 
363         Isend_Irecv(in, out, sendBuf, getBuf, sizeEdges, gridProc, coords, neighb);
364 
365         WaitAll(coords, gridProc, in);
366 
367         FillInEdges(grid, getBuf, block);
368 
369         #pragma omp parallel
370         {
371             double a;
372             double b = 2 * (1/(hX * hX) + 1/(hY * hY) + 1/(hZ * hZ));
373             int threadQuantity = omp_get_num_threads();
374             int threadId = omp_get_thread_num();
375 
376             int i = 1;
377             int j = 1;
378             int k = 1;
379             UpdateCoords(i, j, k, threadId, block);
380 
381             while(k <= block[2])
```

```
382             {
383                 a = (grid[GetPos(i - 1, j, k, block[1] + 2, block[0] + 2)] + grid[GetPos
                        (i + 1, j, k, block[1] + 2, block[0] + 2)]) / (hX * hX);
384                 a += (grid[GetPos(i, j - 1, k, block[1] + 2, block[0] + 2)] + grid[
                        GetPos(i, j + 1, k, block[1] + 2, block[0] + 2)]) / (hY * hY);
385                 a += (grid[GetPos(i, j, k - 1, block[1] + 2, block[0] + 2)] + grid[
                        GetPos(i, j, k + 1, block[1] + 2, block[0] + 2)]) / (hZ * hZ);
386
387                 newGrid[GetPos(i, j, k, block[1] + 2, block[0] + 2)] = a / b;
388                 maxValues[threadId] = FindMax(grid[GetPos(i, j, k, block[1] + 2, block
                        [0] + 2)], newGrid[GetPos(i, j, k, block[1] + 2, block[0] + 2)],
                        maxValues[threadId]);
389                 UpdateCoords(i, j, k, threadQuantity, block);
390             }
391
392         }
393
394         WaitAll(coords, gridProc, out);
395         FillOutEdges(newGrid, sendBuf, block);
396
397         for (int idx = 0; idx < threadsMaxQuantity; ++idx)
398         {
399             maxConvergence = maxValues[idx] > maxConvergence ? maxValues[idx] :
                    maxConvergence;
400         }
401         MPI_Allreduce(&maxConvergence, &globalMax, 1, MPI_DOUBLE, MPI_MAX,
                MPI_COMM_WORLD);
402
403         double* tmp = grid;
404         grid = newGrid;
405         newGrid = tmp;
406
407     } while (globalMax >= eps);
408
409     WriteOut(output, gridProc, block, coords, grid, newGrid, rank);
410     Clear(grid, newGrid, sendBuf, getBuf, maxValues);
411 }
412
413
414
415 int main(int argc, char *argv[])
416 {
417     int gridProc[axes], block[axes];
418     double l[axes];
419     double eps, u0;
420     double u[directions];
421     std::string output;
422
423     int numProcs, rank;
```

```
424
425     MPI_Init(&argc, &argv);
426     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
427     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
428
429     if (!rank)
430     {
431         std::cin >> gridProc[0] >> gridProc[1] >> gridProc[2];
432         std::cin >> block[0] >> block[1] >> block[2];
433         std::cin >> output;
434         std::cin >> eps;
435         std::cin >> l[0] >> l[1] >> l[2];
436         //front back doum up left right
437         std::cin >> u[4] >> u[5] >> u[0] >> u[1] >> u[2] >> u[3];
438         std::cin >> u0;
439     }
440
441     Start(gridProc, block, output, eps, l, u, u0, numProcs, rank);
442
443     MPI_Finalize();
444
445     return 0;
446 }
```

3 Результаты

proc	block	time CPU	time MPI	time OMP + MPI	winner
(1, 1, 1)	(10, 10, 10)	29.8595 ms	134.657 ms	10486.9ms	CPU
(2, 2, 2)	(10, 10, 10)	252.645 ms	4168.76 ms	115302 ms	CPU
(1, 1, 1)	(20, 20, 20)	299.33 ms	1020.22 ms	533.022 ms	CPU
(2, 2, 2)	(20, 20, 20)	5828.44 ms	5777.1 ms	177504 ms	MPI
(1, 1, 1)	(40, 40, 40)	5993.12 ms	23948.9 ms	9647.76 ms	CPU
(2, 2, 2)	(40, 40, 40)	163418 ms	156571 ms	994120 ms	MPI

4 Выводы

Для выполнения данной лабораторной работы нужно было решить задачу Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода. Для этого понадобилось познакомиться с новой технологией OpenMP. OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения. За счет идеи "инкрементального распараллеливания"OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы. В комбинации с технологией MPI эта технология должна была давать огромный прирост эффективности и сокращения времени выполнения, но это не подтвердилось на практике, когда дело дошло до измерения результатов работы. Только на нескольких тестах выиграла программа с MPI, на остальных CPU. Думаю, что полученные результаты сильно зависят от того, что время тратится на создание и инициализацию потоков, также кластеру могло не хватить ресурсов, к тому же допускаю, что сказывается моя неопытность в использовании этих технологий, уверена, что есть еще много разных фич, которые могут ускорить программу.