Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №7 по курсу «Программирование графических процессоров»
(Лабораторная работа №1 по курсу «Параллельная обработка данных»)

Message Passing Interface (MPI).

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

Москва, 2020

Условие

Цель работы: Знакомство с технологией MPI. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант 8:
обмен граничными слоями через isend/irecv, контроль сходимости allreduce. **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

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.

Описание программы

Используются виртуальные блоки (буферы), окружающие основной блок. В самом начале делаем обмен граничными слоями с помощью асинхронных Isend/Irecv. Далее ожидаем конец приема и заполняем полученные значения от соседей внутрь буфера, чтобы посчитать новую итерацию. Потом проходим по середине блока и вычисляем новые значения по заданной формуле из условия, не трогая границы. Здесь же изменяем значение модуля разности по блоку. Затем ожидаем конец обменов и заполняем границы по новым полученным значениям, чтобы отправить соседям. Функция MPI_Allreduce имеет варианты каждой из операции редукции, где результат возвращается всем процессам группы (в нашем случае максимум). Это необходимо для контроля сходимости.

$$u_{i,j,k}^{(k+1)} = \frac{\left(u_{i+1,j,k}^{(k)} + u_{i-1,j,k}^{(k)}\right)h_x^{-2} + \left(u_{i,j+1,k}^{(k)} + u_{i,j-1,k}^{(k)}\right)h_y^{-2} + \left(u_{i,j,k+1}^{(k)} + u_{i,j,k-1}^{(k)}\right)h_z^{-2}}{2\left(h_x^{-2} + h_y^{-2} + h_z^{-2}\right)},$$

2 Исходный код

```
1  #include <iostream>
2  #include <string>
3  #include <algorithm>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include "mpi.h"
7
8
9
10 const int axes = 3;
11 const int directions = 6;
12
13
14 void WaitAll(int* coords, int* gridProc, MPI_Request *arrOfRequests)
15 {
16     MPI_Status tmp;
17
18     for (int i = 0, j = 1; i < axes; ++i, j += 2)
19     {
20         if (coords[i] > 0)
21         {
22             MPI_Wait(&arrOfRequests[i * 2], &tmp);
23         }
24         if (coords[i] < gridProc[i] - 1)
25         {
26             MPI_Wait(&arrOfRequests[j], &tmp);
27         }
28     }
29 }
30
31
32 double FindMax(double a, double b, double max)
33 {
34     if (fabs(a - b) > max)
35     {
36         max = fabs(a - b);
37     }
38     return max;
39 }
40
41
42 int GetPos(int i, int j, int k, int nY, int nX)
43 {
44     return i + (j + k * nY) * nX;
45 }
46
47
```

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

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

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

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

```
        int second_n = (ax != 0) ? nX : nY;

        for (int j = 0; j < first_n; ++j)
        {
            for (int i = 0; i < second_n; ++i)
            {
                grid[idx(i, j)] = getBuf[d][i + j * second_n];
            }
        }
        tmp[ax] = 0;
    }
}



void FillOutEdges(double* newGrid, double** sendBuf, int* block)
{
    int tmp[axes] = {0};
    ++tmp[0];
    --tmp[0];
    int nX = block[0] + 2, nY = block[1] + 2, nZ = block[2] + 2;

    for (int d = 0; d < directions; ++d)
    {
        int ax = d >> 1;
        tmp[ax] = 1;
        int lim = (d & 1);
        // formula to get coords x + (y + z * ny) * nx
        auto idx = [nX, nY, nZ, tmp, lim](int i, int j)
        {
            return tmp[0] * (lim * (nX - 3) + 1 + (i + j * nY) * nX) +
            tmp[1] * (i + (lim * (nY - 3) + 1 + j * nY) * nX) +
            tmp[2] * (i + (j + (lim * (nZ - 3) + 1) * nY) * nX);
        };

        int first_n = (ax == 2) ? nY : nZ;
        int second_n = (ax != 0) ? nX : nY;

        for (int j = 0; j < first_n; ++j)
        {
            for (int i = 0; i < second_n; ++i)
            {
                sendBuf[d][i + j * second_n] = newGrid[idx(i, j)];
            }
        }
        tmp[ax] = 0;
    }
}
```

```
285
286 void Start(int* gridProc, int* block, std::string& output,
287           double eps, double* l, double* u,
288           double u0, int numProcs, int rank)
289 {
290     MPI_Bcast(gridProc, axes, MPI_INT, 0, MPI_COMM_WORLD);
291     MPI_Bcast(block, axes, MPI_INT, 0, MPI_COMM_WORLD);
292     MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
293     MPI_Bcast(l, axes, MPI_DOUBLE, 0, MPI_COMM_WORLD);
294     MPI_Bcast(u, directions, MPI_DOUBLE, 0, MPI_COMM_WORLD);
295     MPI_Bcast(&u0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
296
297
298     MPI_Request in[directions], out[directions];
299
300     int coords[axes];
301     GetCoords(coords, rank, gridProc);
302
303     int neighb[directions];
304     GetNeighbours(neighb, gridProc, coords);
305
306     double* grid = new double[(block[0] + 2) * (block[1] + 2) * (block[2] + 2)];
307     double* newGrid = new double[(block[0] + 2) * (block[1] + 2) * (block[2] + 2)];
308     FillBuffer(grid, (block[0] + 2) * (block[1] + 2) * (block[2] + 2), u0);
309     FillBuffer(newGrid, (block[0] + 2) * (block[1] + 2) * (block[2] + 2), u0);
310
311     double* sendBuf[directions];
312     double* getBuf[directions];
313
314     int sizeEdges[axes];
315     sizeEdges[0] = (block[1] + 2) * (block[2] + 2); // y z
316     sizeEdges[1] = (block[0] + 2) * (block[2] + 2); // x z
317     sizeEdges[2] = (block[0] + 2) * (block[1] + 2); // x y
318
319     InitBufsEdge(sendBuf, getBuf, sizeEdges, gridProc, coords, u, u0);
320
321
322     double nX = gridProc[0] * block[0];
323     double nY = gridProc[1] * block[1];
324     double nZ = gridProc[2] * block[2];
325
326
327     double hX = (double)(l[0] / nX);
328     double hY = (double)(l[1] / nY);
329     double hZ = (double)(l[2] / nZ);
330
331     double maxConvergence;
332     double a, b, globalMax = 0.0;
333
```

```
334     do {
335         maxConvergence = 0.0;
336         Isend_Irecv(in, out, sendBuf, getBuf, sizeEdges, gridProc, coords, neighb);
337 
338         WaitAll(coords, gridProc, in);
339 
340         FillInEdges(grid, getBuf, block);
341 
342         for (int k = 1; k <= block[2]; ++k)
343         {
344             for (int j = 1; j <= block[1]; ++j)
345             {
346                 for (int i = 1; i <= block[0]; ++i)
347                 {
348                     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);
349                     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);
350                     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);
351 
352                     b = 2 * (1/(hX * hX) + 1/(hY * hY) + 1/(hZ * hZ));
353                     newGrid[GetPos(i, j, k, block[1] + 2, block[0] + 2)] = a / b;
354                     maxConvergence = FindMax(grid[GetPos(i, j, k, block[1] + 2, block[0]
                             + 2)], newGrid[GetPos(i, j, k, block[1] + 2, block[0] + 2)],
                            maxConvergence);
355                 }
356             }
357         }
358 
359         WaitAll(coords, gridProc, out);
360 
361         FillOutEdges(newGrid, sendBuf, block);
362 
363         MPI_Allreduce(&maxConvergence, &globalMax, 1, MPI_DOUBLE, MPI_MAX,
                MPI_COMM_WORLD);
364 
365         double* tmp = grid;
366         grid = newGrid;
367         newGrid = tmp;
368 
369     } while (globalMax >= eps);
370 
371     WriteOut(output, gridProc, block, coords, grid, newGrid, rank);
372     Clear(grid, newGrid, sendBuf, getBuf);
373 }
374 
375 
376 
```

```
377 int main(int argc, char *argv[])
378 {
379     //axes=3
380     int gridProc[axes], block[axes];
381     double l[axes];
382     double eps, u0;
383     double u[directions];
384     std::string output;
385 
386     int numProcs, rank;
387 
388     MPI_Init(&argc, &argv);
389     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
390     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
391 
392     if (!rank) // main process
393     {
394         std::cin >> gridProc[0] >> gridProc[1] >> gridProc[2];
395         std::cin >> block[0] >> block[1] >> block[2];
396         std::cin >> output;
397         std::cin >> eps;
398         std::cin >> l[0] >> l[1] >> l[2];
399         //front back down up left right
400         std::cin >> u[4] >> u[5] >> u[0] >> u[1] >> u[2] >> u[3];
401         std::cin >> u0;
402     }
403 
404     Start(gridProc, block, output, eps, l, u, u0, numProcs, rank);
405 
406     MPI_Finalize();
407 
408     return 0;
409 }
```

3 Результаты

proc	time CPU	time MPI
(1, 1, 1)	26.2936 ms	24.282 ms
(2, 2, 2)	21.8879 ms	453.283 ms
(4, 4, 4)	25.7157 ms	10075.6 ms

Таким образом, использование нескольких процессов при выпонении данного задания не дает гарантии улучшения программы, программа написанная на CPU работает более эффективно.

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 технологией MPI. Это программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Ожидалось, что время работы с использованием данной технологии сократится, что не подтвердилось на практике, когда дело дошло до измерения результатов работы. Аналогичная программа, но написанная для CPU, превзошла по эффективности MPI. Можно предположить, что если применить распараллеливание внутри каждого процесса MPI, тогда программа будет работать быстрее, но все же сомневаюсь, что сможет догнать результаты на CPU.