3МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №9 (2)
по курсу «Параллельная Обработка Данных»

Технология MPI и технология OpenMP

Выполнил: Иларионов Д.А.
Группа: М8О-408Б-17
Преподаватели: Крашенинников К.Г.,
Морозов А.Ю.

Москва, 2020

Условие

Совместное использование технологии MPI и технологии
OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.
Требуется решить задачу описанную в лабораторной работе No7, с использованием стандарта распараллеливания openmp в рамках одного процесса.

Вариант 2. Распараллеливание в общем виде с разделением работы между нитями вручную (“в стиле CUDA”).

Программное и аппаратное обеспечение

GPU:

- Name: GeForce GTX 1060
- Compute capability: 6.1
- Частота видеопроцессора: 1404 – 1670 (Boost) МГц
- Частота памяти: 8000 МГц
- Графическая память: 6144 МБ
- Разделяемая память: отсутствует
- Количество регистров на блок: 65536
- Максимальное количество блоков: (2147483647, 65535, 65535)
- Максимальное количество нитей: (1024, 1024, 64)
- Количество мультипроцессоров: 10

Сведения о системе:

- Процессор: Intel Core i7-8750H 2.20GHz x 6
- Оперативная память: 16 ГБ
- SSD: 128 ГБ
- HDD: 1000 ГБ

Программное обеспечение:

- OS: Windows 10
- IDE: Visual Studio 2019
- Компилятор: nvcc

Метод решения

С этой лабораторной у меня не возникло таких проблем, как с седьмой. А просто, она основана на седьмой лабе. Нужно лишь внести некоторые изменения. Самое главное – распараллелить вычисления. У нас есть потоки, и каждый поток выполняет свою работу. Распараллеливание идет как в CUDA – то есть есть индекс (номер потока), а смещение равно количеству всех потоков. У меня их 12, как я понял – макс. число потоков = логическому числу ядер CPU. (У меня 6 физических – 12 логических всего). Одногруппник посоветовал мне сделать отдельные функции для удобства, чтобы переходить на следующие индексы. В OpenMP все блоки программы в #pragma omp parallel {}, а функции в #define function(…) {}. Чтобы функции работали корректно в конце каждой строки нужно ставить слеш “\”. Мы прибавляем к I наше смешение, вычитаем g1, если I > g1, и прибавляем 1 к j. Если j > g2, то вычитаем из j g2 и прибавляем 1 к k. Можно было это сделать и с помощью функции остатка, но мне лень

было, да и это не особо улучшит программу, ведь смещение = числу потоков не сильно велико. Да и думаю, операция взятия остатка менее эффективная, чем обычное вычитание. Была одна проблема, что индексы определялись неправильно. А дело в локальной и глобальной областях памяти. У меня переменные были определены вне блока потока, поэтому каждый поток изменял одну и ту же переменную. А нужно было переопределить эти переменные внутри каждого блока потока. Тогда для каждого потока была своя переменная. Еще. Сначала, я делал как в ЛР7 передачу через буферы. Но пока разбирался с ЛР8, я неправильно прочитал условие, и думал, что там передачу нужно делать через производные типы данных. А оказалось, что это нужно только для вывода. Поэтому, я такую передачу решил добавить в ЛР9 и снова послать на чекер. Теперь не нужно каждый раз заполнять буфера, а мы просто в производных типах данных указываем, какие элементы нам нужно передавать и принимать. Однако, это лишило мою ЛР параллельного заполнения буферов, что тоже подходит к теме, ну да ладно. А вообще, кроме параллельной обработки данных и передачи граничных стен через производные типы данных, тут ничего не поменялось. Поэтому, особо трудностей не возникло, а лабу я сделал за пару дней, если не меньше.

Описание программы

Файл main.cpp

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <time.h>
4.  #include <iostream>
5.  #include <string>
6.  #include <algorithm>
7.
8.  #include <omp.h>
9.  #include <mpi.h>
10.
11. using namespace std;
12.
13. int p1, p2, p3;
14. int g1, g2, g3;
15.
16. // Index inside the block
17. #define _i(i, j, k) ((k + 1) * ((g2 + 2) * (g1 + 2)) + (j + 1) * (g1 + 2) + i + 1)
18. #define _ix(id) (((id) % (g1 + 2)) - 1)
19. #define _iy(id) ((((id) % ((g1 + 2) * (g2 + 2))) / (g1 + 2)) - 1)
20. #define _iz(id) (((id) / ((g1 + 2)*(g2 + 2))) - 1)
21.
22. // Index by processes
23. #define _ib(i, j, k) ((k) * (p1 * p2) + (j) * p1 + (i))
24. #define _ibx(id) ((id) % p1)
25. #define _iby(id) (((id) % (p1 * p2)) / p1)
26. #define _ibz(id) ((id) / (p1 * p2))
27.
28.
29. #define printf(...) fprintf(File, __VA_ARGS__)
30.
31.
32. int main(int argc, char** argv) {
33.     std::ios::sync_with_stdio(false);
34.     string outFile;
35.
36.     int id;
37.     int ib, jb , kb;
38.     int i, j, k, iter;
39.     int numproc, proc_name_len;
40.     char proc_name[MPI_MAX_PROCESSOR_NAME];
41.
42.     double eps;
```

```
43.         double lx, ly, lz;
44.         double hx, hy, hz;
45.         double Udown, Uup, Uleft, Uright, Ufront, Uback;
46.         double U0;
47.         double *data, *temp, *next;
48.         double *bufferIString;
49.
50.         MPI_Status status;
51.
52.         MPI_Init(&argc, &argv);
53.         MPI_Comm_rank(MPI_COMM_WORLD, &id);
54.         MPI_Comm_size(MPI_COMM_WORLD, &numproc);
55.         MPI_Get_processor_name(proc_name, &proc_name_len);
56.
57.         MPI_Barrier(MPI_COMM_WORLD);
58.
59.         //input data for 0 process
60.         if (id == 0) {
61.
62.             cin >> p1 >> p2 >> p3;
63.             cin >> g1 >> g2 >> g3;
64.             cin >> outFile;
65.             cin >> eps;
66.             cin >> lx >> ly >> lz;
67.             cin >> Ufront >> Uback >> Uleft >> Uright >> Uup >> Udown;
68.             cin >> U0;
69.
70.         }
71.
72.
73.         MPI_Barrier(MPI_COMM_WORLD);
74.
75.
76.         //send data to all processes
77.         MPI_Bcast(&p1, 1, MPI_INT, 0, MPI_COMM_WORLD);
78.         MPI_Bcast(&p2, 1, MPI_INT, 0, MPI_COMM_WORLD);
79.         MPI_Bcast(&p3, 1, MPI_INT, 0, MPI_COMM_WORLD);
80.
81.         MPI_Bcast(&g1, 1, MPI_INT, 0, MPI_COMM_WORLD);
82.         MPI_Bcast(&g2, 1, MPI_INT, 0, MPI_COMM_WORLD);
83.         MPI_Bcast(&g3, 1, MPI_INT, 0, MPI_COMM_WORLD);
84.
85.         MPI_Bcast(&eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
86.
87.         MPI_Bcast(&lx, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
88.         MPI_Bcast(&ly, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
89.         MPI_Bcast(&lz, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
90.
91.         MPI_Bcast(&Udown, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
92.         MPI_Bcast(&Uup, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
93.         MPI_Bcast(&Uleft, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
94.         MPI_Bcast(&Uright, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
95.         MPI_Bcast(&Ufront, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
96.         MPI_Bcast(&Uback, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
97.         MPI_Bcast(&U0, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
98.
99.
100.
101.        if (p1 * p2 * p3 != numproc) {
102.            MPI_Finalize();
103.            if (id == 0) {
104.                cout << "ERROR: proc.grid != processes\n";
105.            }
106.
107.            return -1;
108.        }
109.
110.
111.
112.        //block id by coordinates
113.        ib = _ibx(id);
114.        jb = _iby(id);
```

```
115.        kb = _ibz(id);
116.
117.        iter = 0;
118.
119.        //find hs
120.        hx = lx / ((double)p1 * (double)g1);
121.        hy = ly / ((double)p2 * (double)g2);
122.        hz = lz / ((double)p3 * (double)g3);
123.
124.        int max_threads = omp_get_max_threads(); //calculate max threads
125.
126.        data = (double*)malloc(sizeof(double) * (g1 + 2) * (g2 + 2) * (g3 + 2));
127.        next = (double*)malloc(sizeof(double) * (g1 + 2) * (g2 + 2) * (g3 + 2));
128.
129.        //datatypes (NEW)
130.
131.        MPI_Datatype sendLeft, sendRight, recvLeft, recvRight;
132.        MPI_Datatype sendUP, sendDown, recvUP, recvDown;
133.        MPI_Datatype sendFront, sendBack, recvFront, recvBack;
134.
135.        int countFB = g1 * g2;
136.        int countUD = g1 * g3;
137.        int countLR = g2 * g3;
138.
139.        int* lenFB = new int[countFB];
140.        int* lenUD = new int[countUD];
141.        int* lenLR = new int[countLR];
142.
143.        for (int i = 0; i < countFB; ++i) lenFB[i] = 1;
144.        for (int i = 0; i < countUD; ++i) lenUD[i] = 1;
145.        for (int i = 0; i < countLR; ++i) lenLR[i] = 1;
146.
147.        MPI_Aint* adr_sLeft = new MPI_Aint[countLR];
148.        MPI_Aint* adr_rLeft = new MPI_Aint[countLR];
149.        MPI_Aint* adr_sRight = new MPI_Aint[countLR];
150.        MPI_Aint* adr_rRight = new MPI_Aint[countLR];
151.
152.        MPI_Aint* adr_sUP = new MPI_Aint[countUD];
153.        MPI_Aint* adr_rUP = new MPI_Aint[countUD];
154.        MPI_Aint* adr_sDown = new MPI_Aint[countUD];
155.        MPI_Aint* adr_rDown = new MPI_Aint[countUD];
156.
157.        MPI_Aint* adr_sFront = new MPI_Aint[countFB];
158.        MPI_Aint* adr_rFront = new MPI_Aint[countFB];
159.        MPI_Aint* adr_sBack = new MPI_Aint[countFB];
160.        MPI_Aint* adr_rBack = new MPI_Aint[countFB];
161.
162.        for (int k = 0; k < g3; ++k) {
163.            for (int j = 0; j < g2; ++j) {
164.                adr_sLeft[k * g2 + j] = _i(0, j, k) * sizeof(double);
165.                adr_rLeft[k * g2 + j] = _i(-1, j, k) * sizeof(double);
166.                adr_sRight[k * g2 + j] = _i(g1 - 1, j, k) * sizeof(double);
167.                adr_rRight[k * g2 + j] = _i(g1, j, k) * sizeof(double);
168.            }
169.        }
170.
171.        for (int k = 0; k < g3; ++k) {
172.            for (int i = 0; i < g1; ++i) {
173.                adr_sUP[k * g1 + i] = _i(i, 0, k) * sizeof(double);
174.                adr_rUP[k * g1 + i] = _i(i, -1, k) * sizeof(double);
175.                adr_sDown[k * g1 + i] = _i(i, g2 - 1, k) * sizeof(double);
176.                adr_rDown[k * g1 + i] = _i(i, g2, k) * sizeof(double);
177.            }
178.        }
179.
180.        for (int j = 0; j < g2; ++j) {
181.            for (int i = 0; i < g1; ++i) {
182.                adr_sFront[j * g1 + i] = _i(i, j, 0) * sizeof(double);
183.                adr_rFront[j * g1 + i] = _i(i, j, -1) * sizeof(double);
184.                adr_sBack[j * g1 + i] = _i(i, j, g3 - 1) * sizeof(double);
185.                adr_rBack[j * g1 + i] = _i(i, j, g3) * sizeof(double);
186.            }
```

```
187.        }
188.
189.        MPI_Type_create_hindexed(countFB, lenFB, adr_sFront, MPI_DOUBLE, &sendFront);
190.        MPI_Type_create_hindexed(countFB, lenFB, adr_rFront, MPI_DOUBLE, &recvFront);
191.        MPI_Type_create_hindexed(countFB, lenFB, adr_sBack, MPI_DOUBLE, &sendBack);
192.        MPI_Type_create_hindexed(countFB, lenFB, adr_rBack, MPI_DOUBLE, &recvBack);
193.
194.        MPI_Type_create_hindexed(countUD, lenUD, adr_sUP, MPI_DOUBLE, &sendUP);
195.        MPI_Type_create_hindexed(countUD, lenUD, adr_rUP, MPI_DOUBLE, &recvUP);
196.        MPI_Type_create_hindexed(countUD, lenUD, adr_sDown, MPI_DOUBLE, &sendDown);
197.        MPI_Type_create_hindexed(countUD, lenUD, adr_rDown, MPI_DOUBLE, &recvDown);
198.
199.        MPI_Type_create_hindexed(countLR, lenLR, adr_sLeft, MPI_DOUBLE, &sendLeft);
200.        MPI_Type_create_hindexed(countLR, lenLR, adr_rLeft, MPI_DOUBLE, &recvLeft);
201.        MPI_Type_create_hindexed(countLR, lenLR, adr_sRight, MPI_DOUBLE, &sendRight);
202.        MPI_Type_create_hindexed(countLR, lenLR, adr_rRight, MPI_DOUBLE, &recvRight);
203.
204.
205.        MPI_Type_commit(&sendFront); MPI_Type_commit(&recvFront); MPI_Type_commit(&sendBack);
        MPI_Type_commit(&recvBack);
206.        MPI_Type_commit(&sendUP); MPI_Type_commit(&recvUP); MPI_Type_commit(&sendDown);
        MPI_Type_commit(&recvDown);
207.        MPI_Type_commit(&sendLeft); MPI_Type_commit(&recvLeft); MPI_Type_commit(&sendRight);
        MPI_Type_commit(&recvRight);
208.
209.        bufferIString = (double*)malloc(sizeof(double) * g1);  // wall3
210.
211.        //make buffer
212.
213.        int buffer_size;
214.
215.        MPI_Pack_size((g1 + 2) * (g2 + 2) * (g3 + 2), MPI_DOUBLE, MPI_COMM_WORLD, &buffer_size);
216.
217.        buffer_size = 2 * (buffer_size + MPI_BSEND_OVERHEAD); //6 edges
218.
219.
220.        double* buffer = (double*)malloc(buffer_size);
221.
222.        double* epsTemp = new double[max_threads];
223.
224.        MPI_Buffer_attach(buffer, buffer_size);
225.
226.        //block init
227.        for (int a = 0; a < g1; ++a) {
228.            for (int b = 0; b < g2; ++b) {
229.                for (int c = 0; c < g3; ++c) {
230.                    data[_i(a, b, c)] = U0;
231.                }
232.            }
233.        }
234.        //requests
235.        MPI_Request send_request1, recv_request1; //output
236.
237.        MPI_Request send_request1_1, recv_request1_1;
238.        MPI_Request send_request2_1, recv_request2_1;
239.        MPI_Request send_request3_1, recv_request3_1;
240.
241.        MPI_Request send_request1_2, recv_request1_2;
242.        MPI_Request send_request2_2, recv_request2_2;
243.        MPI_Request send_request3_2, recv_request3_2;
244.
245.
246.        double* errors;
247.        errors = (double*)malloc(numproc * sizeof(double));
248.
249.        int* i_start, * j_start, * k_start;
250.
251.        k_start = (int*)malloc(sizeof(int) * 2);
252.        j_start = (int*)malloc(sizeof(int) * 2);
253.        i_start = (int*)malloc(sizeof(int) * 2);
254.        //string debug_name = "process_debug" + to_string(id) + ".txt";
255.
```

```
256.
257.       omp_set_dynamic(0); // OMP static threads
258.       double maxErr = 0;
259.
260.
261.       do  {
262.           //send and get data
263.           MPI_Barrier(MPI_COMM_WORLD);
264.
265.           for (int yy = 0; yy < max_threads; ++yy) {
266.               epsTemp[yy] = 0;
267.           }
268.
269.           if (ib > 0 && ib + 1 < p1) { //both left and right
270.               MPI_Isend(data, 1, sendLeft, _ib(ib - 1, jb, kb), 0, MPI_COMM_WORLD,
      &send_request1_1);
271.               MPI_Isend(data, 1, sendRight, _ib(ib + 1, jb, kb), 0, MPI_COMM_WORLD,
      &send_request1_2);
272.           }
273.           else if (ib > 0) { //only left side
274.               MPI_Isend(data, 1, sendLeft, _ib(ib - 1, jb, kb), 0, MPI_COMM_WORLD,
      &send_request1_1);
275.           }
276.           else if (ib + 1 < p1) { //only right side
277.               MPI_Isend(data, 1, sendRight, _ib(ib + 1, jb, kb), 0, MPI_COMM_WORLD,
      &send_request1_2);
278.           }
279.
280.
281.           if (jb + 1 < p2 && jb > 0) { //both down and up
282.               MPI_Isend(data, 1, sendUP, _ib(ib, jb - 1, kb), 0, MPI_COMM_WORLD, &send_request2_2);
283.               MPI_Isend(data, 1, sendDown, _ib(ib, jb + 1, kb), 0, MPI_COMM_WORLD,
      &send_request2_1);
284.           }
285.           else if (jb > 0) { //only up side
286.               MPI_Isend(data, 1, sendUP, _ib(ib, jb - 1, kb), 0, MPI_COMM_WORLD, &send_request2_2);
287.           }
288.           else if (jb + 1 < p2) { //only down side
289.               MPI_Isend(data, 1, sendDown, _ib(ib, jb + 1, kb), 0, MPI_COMM_WORLD,
      &send_request2_1);
290.           }
291.
292.
293.           if (kb + 1 < p3 && kb > 0) { //both back and front
294.               MPI_Isend(data, 1, sendFront, _ib(ib, jb, kb - 1), 0, MPI_COMM_WORLD,
      &send_request3_2);
295.               MPI_Isend(data, 1, sendBack, _ib(ib, jb, kb + 1), 0, MPI_COMM_WORLD,
      &send_request3_1);
296.           }
297.           else if (kb > 0) { //only front side
298.               MPI_Isend(data, 1, sendFront, _ib(ib, jb, kb - 1), 0, MPI_COMM_WORLD,
      &send_request3_2);
299.           }
300.           else if (kb + 1 < p3) { //only back side
301.               MPI_Isend(data, 1, sendBack, _ib(ib, jb, kb + 1), 0, MPI_COMM_WORLD,
      &send_request3_1);
302.           }
303.
304.
305.           //while wait for data
306.
307.           //iterational function (inside of blocks)
308.           #pragma omp parallel
309.           {
310.               int threads = omp_get_num_threads();
311.               int thread_id = omp_get_thread_num();
312.
313.               //-2 because we don't check edges
314.               #define ijk_next(i, j, k, diff) { \
315.                   i += diff; \
316.                   while(i > g1 - 2){ \
317.                       i -= g1 - 2; \
```

```
318.                     ++j; \
319.                 } \
320.                 while(j > g2 - 2){ \
321.                     j -= g2 - 2; \
322.                     ++k; \
323.                 } \
324.             } \
325.
326.             int i_ = 1;
327.             int j_ = 1;
328.             int k_ = 1;
329.             ijk_next(i_, j_, k_, thread_id); //like in CUDA
330.
331.
332.             while (k_ < g3 - 1) {
333.                 next[_i(i_, j_, k_)] = 0.5 * ((data[_i(i_ + 1, j_, k_)] + data[_i(i_ - 1, j_,
    k_)]) / (hx * hx) +
334.                     (data[_i(i_, j_ + 1, k_)] + data[_i(i_, j_ - 1, k_)]) / (hy * hy) +
335.                     (data[_i(i_, j_, k_ + 1)] + data[_i(i_, j_, k_ - 1)]) / (hz * hz)) /
336.                     (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
337.
338.                 epsTemp[thread_id] = max(epsTemp[thread_id], abs(next[_i(i_, j_, k_)] -
    data[_i(i_, j_, k_)]));
339.                 ijk_next(i_, j_, k_, threads);
340.             }
341.         }
342.
343.
344.         //wait for data
345.
346.         if (ib > 0) { //only left side
347.             MPI_Wait(&send_request1_1, &status);
348.         }
349.
350.         if (jb > 0) { //only up side
351.             MPI_Wait(&send_request2_2, &status);
352.         }
353.
354.         if (kb > 0) { //only front side
355.             MPI_Wait(&send_request3_2, &status);
356.         }
357.
358.         //getting 1
359.
360.         if (ib + 1 < p1) { //get right side
361.             MPI_Irecv(data, 1, recvRight, _ib(ib + 1, jb, kb), 0, MPI_COMM_WORLD,
    &recv_request1_2);
362.             MPI_Wait(&recv_request1_2, &status);
363.         }
364.
365.         if (jb + 1 < p2) { //get down side
366.             MPI_Irecv(data, 1, recvDown, _ib(ib, jb + 1, kb), 0, MPI_COMM_WORLD,
    &recv_request2_1);
367.             MPI_Wait(&recv_request2_1, &status);
368.         }
369.
370.         if (kb + 1 < p3) { //get back side
371.             MPI_Irecv(data, 1, recvBack, _ib(ib, jb, kb + 1), 0, MPI_COMM_WORLD,
    &recv_request3_1);
372.             MPI_Wait(&recv_request3_1, &status);
373.         }
374.
375.         //parallel receiving step 1
376.         #pragma omp parallel
377.         {
378.             int threads = omp_get_num_threads();
379.             int thread_id = omp_get_thread_num();
380.
381.             #define ij_next(i, j, diff) { \
382.                 i += diff; \
383.                 while(i > g1 - 1){ \
384.                     i -= g1; \
```

```
                        ++j; \
                    } \
                } \

                #define ik_next(i, k, diff) { \
                    i += diff; \
                    while(i > g1 - 1){ \
                        i -= g1; \
                        ++k; \
                    } \
                } \

                #define jk_next(j, k, diff) { \
                    j += diff; \
                    while(j > g2 - 1){ \
                        j -= g2; \
                        ++k; \
                    } \
                } \

                //set new data
                int i_ = 0;
                int j_ = 0; //int helps this variable be local for each thread
                int k_ = 0;

                jk_next(j_, k_, thread_id);

                if (ib + 1 < p1) { //get right side
                }
                else {
                    while (k_ < g3) {
                        data[_i(g1, j_, k_)] = Uright;
                        next[_i(g1, j_, k_)] = Uright;
                        jk_next(j_, k_, threads);
                    }
                }

                i_ = 0;
                k_ = 0;
                ik_next(i_, k_, thread_id);

                if (jb + 1 < p2) { //get down side
                }
                else {
                    while (k_ < g3) {
                        data[_i(i_, g2, k_)] = Udown;
                        next[_i(i_, g2, k_)] = Udown;
                        ik_next(i_, k_, threads);
                    }
                }

                i_ = 0;
                j_ = 0;
                ij_next(i_, j_, thread_id);

                if (kb + 1 < p3) { //get back side
                }
                else {
                    while (j_ < g2) {
                        data[_i(i_, j_, g3)] = Uback;
                        next[_i(i_, j_, g3)] = Uback;
                        ij_next(i_, j_, threads);
                    }
                }
            }


            if (ib + 1 < p1) { //only right side
                MPI_Wait(&send_request1_2, &status);
            }
            if (jb + 1 < p2) { //only down side
```

```
457.                 MPI_Wait(&send_request2_1, &status);
458.             }
459.             if (kb + 1 < p3) { //only back side
460.                 MPI_Wait(&send_request3_1, &status);
461.             }
462.
463.             //getting 2
464.
465.             if (ib > 0) { //get left side
466.                 MPI_Irecv(data, 1, recvLeft, _ib(ib - 1, jb, kb), 0, MPI_COMM_WORLD,
    &recv_request1_1);
467.                 MPI_Wait(&recv_request1_1, &status);
468.             }
469.
470.
471.             if (jb > 0) { //get up side
472.                 MPI_Irecv(data, 1, recvUP, _ib(ib, jb - 1, kb), 0, MPI_COMM_WORLD, &recv_request2_2);
473.                 MPI_Wait(&recv_request2_2, &status);
474.             }
475.
476.
477.             if (kb > 0) { //get front side
478.                 MPI_Irecv(data, 1, recvFront, _ib(ib, jb, kb - 1), 0, MPI_COMM_WORLD,
    &recv_request3_2);
479.                 MPI_Wait(&recv_request3_2, &status);
480.             }
481.
482.
483.
484.             //parallel receiving step 2
485.             #pragma omp parallel
486.             {
487.                 int threads = omp_get_num_threads();
488.                 int thread_id = omp_get_thread_num();
489.
490.                 #define ij_next(i, j, diff) { \
491.                     i += diff; \
492.                     while(i > g1 - 1){ \
493.                         i -= g1; \
494.                         ++j; \
495.                     } \
496.                 } \
497.
498.                 #define ik_next(i, k, diff) { \
499.                     i += diff; \
500.                     while(i > g1 - 1){ \
501.                         i -= g1; \
502.                         ++k; \
503.                     } \
504.                 } \
505.
506.                 #define jk_next(j, k, diff) { \
507.                     j += diff; \
508.                     while(j > g2 - 1){ \
509.                         j -= g2; \
510.                         ++k; \
511.                     } \
512.                 } \
513.
514.                 int i_ = 0;
515.                 int j_ = 0;
516.                 int k_ = 0;
517.
518.                 jk_next(j_, k_, thread_id);
519.
520.                 if (ib > 0) { //get left side
521.                 }
522.                 else {
523.                     while (k_ < g3) {
524.                         data[_i(-1, j_, k_)] = Uleft;
525.                         next[_i(-1, j_, k_)] = Uleft;
526.                         jk_next(j_, k_, threads);
```

```
527.                     }
528.                 }
529.
530.
531.                 i_ = 0;
532.                 k_ = 0;
533.                 ik_next(i_, k_, thread_id);
534.
535.                 if (jb > 0) { //get up side
536.                 }
537.                 else {
538.                     while (k_ < g3) {
539.                         data[_i(i_, -1, k_)] = Uup;
540.                         next[_i(i_, -1, k_)] = Uup;
541.                         ik_next(i_, k_, threads);
542.                     }
543.                 }
544.
545.
546.                 i_ = 0;
547.                 j_ = 0;
548.                 ij_next(i_, j_, thread_id);
549.
550.
551.                 if (kb > 0) { //get front side
552.                 }
553.                 else {
554.                     while (j_ < g2) {
555.                         data[_i(i_, j_, -1)] = Ufront;
556.                         next[_i(i_, j_, -1)] = Ufront;
557.                         ij_next(i_, j_, threads);
558.                     }
559.                 }
560.
561.
562.             }
563.
564.             //for edges
565.
566.
567.
568.             k_start[0] = 0;
569.             k_start[1] = g3 - 1;
570.
571.             j_start[0] = 0;
572.             j_start[1] = g2 - 1;
573.
574.             i_start[0] = 0;
575.             i_start[1] = g1 - 1;
576.
577.
578.
579.
580.             #pragma omp parallel
581.             {
582.                 //parallel calculate edges
583.                 int threads = omp_get_num_threads();
584.                 int thread_id = omp_get_thread_num();
585.
586.                 #define ij_next(i, j, diff) { \
587.                     i += diff; \
588.                     while(i > g1 - 1){ \
589.                         i -= g1; \
590.                         ++j; \
591.                     } \
592.                 } \
593.
594.                 #define ik_next(i, k, diff) { \
595.                     i += diff; \
596.                     while(i > g1 - 1){ \
597.                         i -= g1; \
598.                         ++k; \
```

```
599.                 } \
600.             } \
601.
602.             #define jk_next(j, k, diff) { \
603.                 j += diff; \
604.                 while(j > g2 - 1){ \
605.                     j -= g2; \
606.                     ++k; \
607.                 } \
608.             } \
609.
610.
611.             int i_ = 0;
612.             int j_ = 0;
613.             int k_ = 0;
614.
615.
616.             //k-edges
617.             for (int k_s = 0; k_s < 2; ++k_s) {
618.                 k_ = k_start[k_s];
619.                 i_ = 0;
620.                 j_ = 0;
621.                 ij_next(i_, j_, thread_id);
622.
623.                 while (j_ < g2) {
624.                     next[_i(i_, j_, k_)] = 0.5 * ((data[_i(i_ + 1, j_, k_)] + data[_i(i_ - 1, j_,
    k_)]) / (hx * hx) +
625.                         (data[_i(i_, j_ + 1, k_)] + data[_i(i_, j_ - 1, k_)]) / (hy * hy) +
626.                         (data[_i(i_, j_, k_ + 1)] + data[_i(i_, j_, k_ - 1)]) / (hz * hz)) /
627.                         (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
628.                     epsTemp[thread_id] = max(epsTemp[thread_id], abs(next[_i(i_, j_, k_)] -
    data[_i(i_, j_, k_)]));
629.
630.                     ij_next(i_, j_, threads);
631.                 }
632.
633.             }
634.
635.
636.             //j-edges
637.             for (int j_s = 0; j_s < 2; ++j_s) {
638.                 j_ = j_start[j_s];
639.                 i_ = 0;
640.                 k_ = 0;
641.                 ik_next(i_, k_, thread_id);
642.
643.                 while (k_ < g3) {
644.                     next[_i(i_, j_, k_)] = 0.5 * ((data[_i(i_ + 1, j_, k_)] + data[_i(i_ - 1, j_,
    k_)]) / (hx * hx) +
645.                         (data[_i(i_, j_ + 1, k_)] + data[_i(i_, j_ - 1, k_)]) / (hy * hy) +
646.                         (data[_i(i_, j_, k_ + 1)] + data[_i(i_, j_, k_ - 1)]) / (hz * hz)) /
647.                         (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
648.                     epsTemp[thread_id] = max(epsTemp[thread_id], abs(next[_i(i_, j_, k_)] -
    data[_i(i_, j_, k_)]));
649.
650.                     ik_next(i_, k_, threads);
651.                 }
652.             }
653.
654.
655.             //i-edges
656.             for (int i_s = 0; i_s < 2; ++i_s) {
657.                 i_ = i_start[i_s];
658.                 j_ = 0;
659.                 k_ = 0;
660.                 jk_next(j_, k_, thread_id);
661.
662.                 while (k_ < g3) {
663.                     next[_i(i_, j_, k_)] = 0.5 * ((data[_i(i_ + 1, j_, k_)] + data[_i(i_ - 1, j_,
    k_)]) / (hx * hx) +
664.                         (data[_i(i_, j_ + 1, k_)] + data[_i(i_, j_ - 1, k_)]) / (hy * hy) +
665.                         (data[_i(i_, j_, k_ + 1)] + data[_i(i_, j_, k_ - 1)]) / (hz * hz)) /
```

```
                            (1.0 / (hx * hx) + 1.0 / (hy * hy) + 1.0 / (hz * hz));
                        epsTemp[thread_id] = max(epsTemp[thread_id], abs(next[_i(i_, j_, k_)] -
    data[_i(i_, j_, k_)]));

                        jk_next(j_, k_, threads);
                    }
                }

            }




            double maxE[1];
            maxE[0] = 0;

            for (int t = 0; t < max_threads; ++t) {
                maxE[0] = max(maxE[0], epsTemp[t]);
            }


            MPI_Allgather(maxE, 1, MPI_DOUBLE, errors, 1, MPI_DOUBLE, MPI_COMM_WORLD);
            maxE[0] = 0;
            for (i = 0; i < numproc; ++i) {
                maxE[0] = max(maxE[0], errors[i]);
            }


            temp = next;
            next = data;
            data = temp;

            maxErr = maxE[0];

            iter += 1;

        }while (maxErr >= eps);

        //fclose(File);
        //cout << iter << "\n";

        MPI_Barrier(MPI_COMM_WORLD);

        if (id != 0) {
            for (k = 0 ; k < g3 ; ++k) {
                for (j = 0; j < g2; ++j) {
                    for (i = 0; i < g1; ++i) {
                        bufferIString[i] = data[_i(i, j, k)];
                    }
                    MPI_Isend(bufferIString, g1, MPI_DOUBLE, 0, id, MPI_COMM_WORLD, &send_request1);
                    MPI_Wait(&send_request1, &status);
                }
            }
        }
        else {
            cerr << "Process GRID: " << p1 << "x" << p2 << "x" << p3 << "\n";
            cerr << "Num GRID: " << g1 << "x" << g2 << "x" << g3 << "\n";
            cerr << "Eps: " << eps << "\n";
            cerr << "lx: " << lx << " ly: " << ly << " lz: " << lz << "\n";
            cerr << "Us: " << Ufront << " , " << Uback << " , " << Uleft << " , " << Uright << " , "
    << Uup << " , " << Udown << "\n";
            cerr << "U0: " << U0 << "\n";
            cerr << "Iterations: " << iter << "\n";

            FILE* File = fopen(outFile.c_str(), "w+");

            for (kb = 0; kb < p3; ++kb) {
                for (k = 0; k < g3; ++k) {
                    for (jb = 0; jb < p2; ++jb) {
                        for (j = 0; j < g2; ++j) {
                            for (ib = 0; ib < p1; ++ib) {
                                if (_ib(ib, jb, kb) == 0) {
```

```
                                    for (i = 0; i < g1; ++i) {
                                        bufferIString[i] = data[_i(i, j, k)];
                                        printf("%.6e ", bufferIString[i]);
                                    }
                                    if (ib + 1 == p1) {
                                        printf("\n");
                                        if (j + 1 == g2) {
                                            printf("\n");
                                        }
                                    }
                                }
                                else {
                                    MPI_Irecv(bufferIString, g1, MPI_DOUBLE, _ib(ib, jb, kb), _ib(ib,
    jb, kb), MPI_COMM_WORLD, &recv_request1);
                                    MPI_Wait(&recv_request1, &status);
                                    for (i = 0; i < g1; ++i) {
                                        printf("%.6e ", bufferIString[i]);
                                    }
                                    if (ib + 1 == p1) {
                                        printf("\n");
                                        if (j + 1 == g2) {
                                            printf("\n");
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
            fclose(File);
        }

        MPI_Buffer_detach(buffer, &buffer_size);


        MPI_Finalize();

        free(lenFB);
        free(lenUD);
        free(lenLR);

        free(adr_sLeft);
        free(adr_rLeft);
        free(adr_sRight);
        free(adr_rRight);

        free(adr_sUP);
        free(adr_rUP);
        free(adr_sDown);
        free(adr_rDown);

        free(adr_sFront);
        free(adr_rFront);
        free(adr_sBack);
        free(adr_rBack);

        free(data);
        free(next);
        free(buffer);

        return 0;
    }
```

Результаты

Как-то так.

Количество нитей - 12 / CPU - 1							
Размерность учитывается общая, то есть размеры блоков поделены на количество процессов (eps 1e-3)							
Разм / Проц	1x1x1	CPU (1x1x1)	1x1x2	1x2x2	2x2x2	2x2x5	2x5x5
10x10x10	0.01116	0.01309	0.03798	0.05155	0.06245	6.57675	PC FREEZE
20x20x20	0.10437	0.20482	0.18487	0.20806	0.87531	16.7051	PC FREEZE
50x50x50	3.54345	9.00067	5.54964	4.31043	7.00588	70.9064	PC FREEZE
100x100x100	42.2727	127.757	73.2098	60.5419	55.8599	179.399	PC FREEZE
200x200x200	574.402	1080.22	536.726	456.949	440.343	633.317	PC FREEZE

Для сравнения – результаты из ЛР7

Размерность учитывается общая, то есть размеры блоков поделены на количество процессов (eps 1e-3)							
Разм / Проц	1x1x1	CPU (1x1x1)	1x1x2	1x2x2	2x2x2	2x2x5	2x5x5
10x10x10	0.01478	0.01309	0.02094	0.02031	0.01814	4.42374	32.3297
20x20x20	0.18774	0.20482	0.10972	0.06897	0.06047	12.3675	102.186
50x50x50	8.96422	9.00067	4.73802	2.63155	2.1569	50.4081	332.556
100x100x100	127.123	127.757	64.5818	37.0962	29.4352	144.84	739.977
200x200x200	1028.76	1080.22	523.996	353.169	293.215	255.162	1036.46

Посмотрим. Мы видим значительное улучшение производительности. Нет, нет. Оно только тогда, когда у нас всего 1 процесс. А вот, когда их становится больше, все идет только хуже и хуже. Дело в том, что требуется очень много вычислительных ресурсов, поэтому, и время выполнения, когда у нас больше 1 процесса – больше. Для последней сетки процессов мне даже не удалось выполнить тесты. У меня просто зависал компьютер, мышка еле двигалась, а CPU был нагружен на 100%. Поэтому, данная тема с потоками хороша только для обычных, однопроцессных конфигураций. При большем числе процессов, мы теряем в производительности. Однако, лучшего результата среди всех, нам удалось добиться только для блока 10x10x10. В остальных, при нескольких процессах работает лучше без потоков, чем с 1 процессом и потоками.

Заметим, что при большом количестве процессов MPI, программа начинает падать в производительности. Дело в том, что уходит очень много времени на обмен граничными слоями. Поэтому, необходимо найти баланс. Лучше всего в целом программа работает про конфигурации ядра 2x2x2. То есть, 8 процессов. Однако, MPI намного превосходит в производительности обычную программу без этой технологии. Особенно при больших данных, однако, слишком много процессов использовать не стоит.

Выводы

По сравнению с 7 ЛР, эта значительно легче. Но она же основана на самой ЛР 7, так что, без ЛР 7, тут бы я еще сильнее и дольше мучался. Потоки, как показали тесты, имеют эффективность только для 1 процесса. А при большом их количестве, только сильнее нагружается система, а каждая итерация требует дополнительного времени для выделения потоков для каждого процесса. Не вижу особого смысла в их использовании в данной задаче, ведь, запустив несколько процессов без потоков, мы быстрее добьемся результата, чем при запуске 1 процесса с несколькими потоками.