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>
8. #include <omp.h>
9. #include <mpi.h>
11. using namespace std;
13. int p1, p2, p3;
14. int g1, g2, g3;
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)
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))

29. #define printf(...) fprintf(File, \_\_VA\_ARGS\_\_)

32. int main(int argc, char\*\* argv) {
33. std::ios::sync\_with\_stdio(false);
34. string outFile;
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];
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;
50. MPI\_Status status;
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);
57. MPI\_Barrier(MPI\_COMM\_WORLD);
59. //input data for 0 process
60. if (id == 0) {
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;
70. }

73. MPI\_Barrier(MPI\_COMM\_WORLD);

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);
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);
85. MPI\_Bcast(&eps, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
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);
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);


101. if (p1 \* p2 \* p3 != numproc) {
102. MPI\_Finalize();
103. if (id == 0) {
104. cout << "ERROR: proc.grid != processes**\n**";
105. }
107. return -1;
108. }


112. //block id by coordinates
113. ib = \_ibx(id);
114. jb = \_iby(id);
115. kb = \_ibz(id);
117. iter = 0;
119. //find hs
120. hx = lx / ((double)p1 \* (double)g1);
121. hy = ly / ((double)p2 \* (double)g2);
122. hz = lz / ((double)p3 \* (double)g3);
124. int max\_threads = omp\_get\_max\_threads(); //calculate max threads
126. data = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2) \* (g3 + 2));
127. next = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2) \* (g3 + 2));
129. //datatypes (NEW)
131. MPI\_Datatype sendLeft, sendRight, recvLeft, recvRight;
132. MPI\_Datatype sendUP, sendDown, recvUP, recvDown;
133. MPI\_Datatype sendFront, sendBack, recvFront, recvBack;
135. int countFB = g1 \* g2;
136. int countUD = g1 \* g3;
137. int countLR = g2 \* g3;
139. int\* lenFB = new int[countFB];
140. int\* lenUD = new int[countUD];
141. int\* lenLR = new int[countLR];
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;
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];
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];
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];
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. }
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. }
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. }
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);
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);
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);

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);
209. bufferIString = (double\*)malloc(sizeof(double) \* g1);  // wall3
211. //make buffer
213. int buffer\_size;
215. MPI\_Pack\_size((g1 + 2) \* (g2 + 2) \* (g3 + 2), MPI\_DOUBLE, MPI\_COMM\_WORLD, &buffer\_size);
217. buffer\_size = 2 \* (buffer\_size + MPI\_BSEND\_OVERHEAD); //6 edges

220. double\* buffer = (double\*)malloc(buffer\_size);
222. double\* epsTemp = new double[max\_threads];
224. MPI\_Buffer\_attach(buffer, buffer\_size);
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
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;
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;

246. double\* errors;
247. errors = (double\*)malloc(numproc \* sizeof(double));
249. int\* i\_start, \* j\_start, \* k\_start;
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";

257. omp\_set\_dynamic(0); // OMP static threads
258. double maxErr = 0;

261. do  {
262. //send and get data
263. MPI\_Barrier(MPI\_COMM\_WORLD);
265. for (int yy = 0; yy < max\_threads; ++yy) {
266. epsTemp[yy] = 0;
267. }
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. }

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. }

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. }

305. //while wait for data
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();
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. } \
326. int i\_ = 1;
327. int j\_ = 1;
328. int k\_ = 1;
329. ijk\_next(i\_, j\_, k\_, thread\_id); //like in CUDA

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));
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. }

344. //wait for data
346. if (ib > 0) { //only left side
347. MPI\_Wait(&send\_request1\_1, &status);
348. }
350. if (jb > 0) { //only up side
351. MPI\_Wait(&send\_request2\_2, &status);
352. }
354. if (kb > 0) { //only front side
355. MPI\_Wait(&send\_request3\_2, &status);
356. }
358. //getting 1
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. }
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. }
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. }
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();
381. #define ij\_next(i, j, diff) { \
382. i += diff; \
383. while(i > g1 - 1){ \
384. i -= g1; \
385. ++j; \
386. } \
387. } \
389. #define ik\_next(i, k, diff) { \
390. i += diff; \
391. while(i > g1 - 1){ \
392. i -= g1; \
393. ++k; \
394. } \
395. } \
397. #define jk\_next(j, k, diff) { \
398. j += diff; \
399. while(j > g2 - 1){ \
400. j -= g2; \
401. ++k; \
402. } \
403. } \
405. //set new data
406. int i\_ = 0;
407. int j\_ = 0; //int helps this variable be local for each thread
408. int k\_ = 0;
410. jk\_next(j\_, k\_, thread\_id);
412. if (ib + 1 < p1) { //get right side
413. }
414. else {
415. while (k\_ < g3) {
416. data[\_i(g1, j\_, k\_)] = Uright;
417. next[\_i(g1, j\_, k\_)] = Uright;
418. jk\_next(j\_, k\_, threads);
419. }
420. }
422. i\_ = 0;
423. k\_ = 0;
424. ik\_next(i\_, k\_, thread\_id);
426. if (jb + 1 < p2) { //get down side
427. }
428. else {
429. while (k\_ < g3) {
430. data[\_i(i\_, g2, k\_)] = Udown;
431. next[\_i(i\_, g2, k\_)] = Udown;
432. ik\_next(i\_, k\_, threads);
433. }
434. }
436. i\_ = 0;
437. j\_ = 0;
438. ij\_next(i\_, j\_, thread\_id);
440. if (kb + 1 < p3) { //get back side
441. }
442. else {
443. while (j\_ < g2) {
444. data[\_i(i\_, j\_, g3)] = Uback;
445. next[\_i(i\_, j\_, g3)] = Uback;
446. ij\_next(i\_, j\_, threads);
447. }
448. }
449. }


453. if (ib + 1 < p1) { //only right side
454. MPI\_Wait(&send\_request1\_2, &status);
455. }
456. 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. }
463. //getting 2
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. }

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. }

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. }


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();
490. #define ij\_next(i, j, diff) { \
491. i += diff; \
492. while(i > g1 - 1){ \
493. i -= g1; \
494. ++j; \
495. } \
496. } \
498. #define ik\_next(i, k, diff) { \
499. i += diff; \
500. while(i > g1 - 1){ \
501. i -= g1; \
502. ++k; \
503. } \
504. } \
506. #define jk\_next(j, k, diff) { \
507. j += diff; \
508. while(j > g2 - 1){ \
509. j -= g2; \
510. ++k; \
511. } \
512. } \
514. int i\_ = 0;
515. int j\_ = 0;
516. int k\_ = 0;
518. jk\_next(j\_, k\_, thread\_id);
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. }

531. i\_ = 0;
532. k\_ = 0;
533. ik\_next(i\_, k\_, thread\_id);
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. }

546. i\_ = 0;
547. j\_ = 0;
548. ij\_next(i\_, j\_, thread\_id);

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. }

562. }
564. //for edges


568. k\_start[0] = 0;
569. k\_start[1] = g3 - 1;
571. j\_start[0] = 0;
572. j\_start[1] = g2 - 1;
574. i\_start[0] = 0;
575. i\_start[1] = g1 - 1;



580. #pragma omp parallel
581. {
582. //parallel calculate edges
583. int threads = omp\_get\_num\_threads();
584. int thread\_id = omp\_get\_thread\_num();
586. #define ij\_next(i, j, diff) { \
587. i += diff; \
588. while(i > g1 - 1){ \
589. i -= g1; \
590. ++j; \
591. } \
592. } \
594. #define ik\_next(i, k, diff) { \
595. i += diff; \
596. while(i > g1 - 1){ \
597. i -= g1; \
598. ++k; \
599. } \
600. } \
602. #define jk\_next(j, k, diff) { \
603. j += diff; \
604. while(j > g2 - 1){ \
605. j -= g2; \
606. ++k; \
607. } \
608. } \

611. int i\_ = 0;
612. int j\_ = 0;
613. int k\_ = 0;

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);
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\_)]));
630. ij\_next(i\_, j\_, threads);
631. }
633. }

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);
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\_)]));
650. ik\_next(i\_, k\_, threads);
651. }
652. }

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);
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)) /
666. (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));
667. epsTemp[thread\_id] = max(epsTemp[thread\_id], abs(next[\_i(i\_, j\_, k\_)] - data[\_i(i\_, j\_, k\_)]));
669. jk\_next(j\_, k\_, threads);
670. }
671. }
673. }



678. double maxE[1];
679. maxE[0] = 0;
681. for (int t = 0; t < max\_threads; ++t) {
682. maxE[0] = max(maxE[0], epsTemp[t]);
683. }

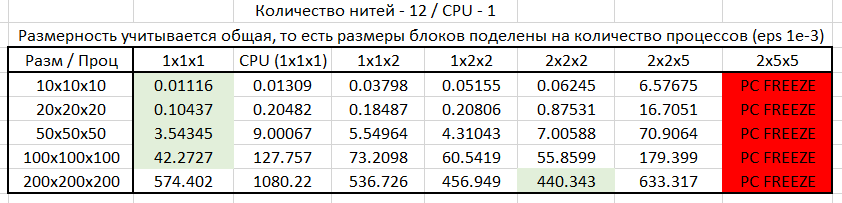
686. MPI\_Allgather(maxE, 1, MPI\_DOUBLE, errors, 1, MPI\_DOUBLE, MPI\_COMM\_WORLD);
687. maxE[0] = 0;
688. for (i = 0; i < numproc; ++i) {
689. maxE[0] = max(maxE[0], errors[i]);
690. }

693. temp = next;
694. next = data;
695. data = temp;
697. maxErr = maxE[0];
699. iter += 1;
701. }while (maxErr >= eps);
703. //fclose(File);
704. //cout << iter << "\n";
706. MPI\_Barrier(MPI\_COMM\_WORLD);
708. if (id != 0) {
709. for (k = 0 ; k < g3 ; ++k) {
710. for (j = 0; j < g2; ++j) {
711. for (i = 0; i < g1; ++i) {
712. bufferIString[i] = data[\_i(i, j, k)];
713. }
714. MPI\_Isend(bufferIString, g1, MPI\_DOUBLE, 0, id, MPI\_COMM\_WORLD, &send\_request1);
715. MPI\_Wait(&send\_request1, &status);
716. }
717. }
718. }
719. else {
720. cerr << "Process GRID: " << p1 << "x" << p2 << "x" << p3 << "**\n**";
721. cerr << "Num GRID: " << g1 << "x" << g2 << "x" << g3 << "**\n**";
722. cerr << "Eps: " << eps << "**\n**";
723. cerr << "lx: " << lx << " ly: " << ly << " lz: " << lz << "**\n**";
724. cerr << "Us: " << Ufront << " , " << Uback << " , " << Uleft << " , " << Uright << " , " << Uup << " , " << Udown << "**\n**";
725. cerr << "U0: " << U0 << "**\n**";
726. cerr << "Iterations: " << iter << "**\n**";
728. FILE\* File = fopen(outFile.c\_str(), "w+");
730. for (kb = 0; kb < p3; ++kb) {
731. for (k = 0; k < g3; ++k) {
732. for (jb = 0; jb < p2; ++jb) {
733. for (j = 0; j < g2; ++j) {
734. for (ib = 0; ib < p1; ++ib) {
735. if (\_ib(ib, jb, kb) == 0) {
736. for (i = 0; i < g1; ++i) {
737. bufferIString[i] = data[\_i(i, j, k)];
738. printf("%.6e ", bufferIString[i]);
739. }
740. if (ib + 1 == p1) {
741. printf("**\n**");
742. if (j + 1 == g2) {
743. printf("**\n**");
744. }
745. }
746. }
747. else {
748. MPI\_Irecv(bufferIString, g1, MPI\_DOUBLE, \_ib(ib, jb, kb), \_ib(ib, jb, kb), MPI\_COMM\_WORLD, &recv\_request1);
749. MPI\_Wait(&recv\_request1, &status);
750. for (i = 0; i < g1; ++i) {
751. printf("%.6e ", bufferIString[i]);
752. }
753. if (ib + 1 == p1) {
754. printf("**\n**");
755. if (j + 1 == g2) {
756. printf("**\n**");
757. }
758. }
759. }
760. }
761. }
762. }
763. }
764. }
765. fclose(File);
766. }
768. MPI\_Buffer\_detach(buffer, &buffer\_size);

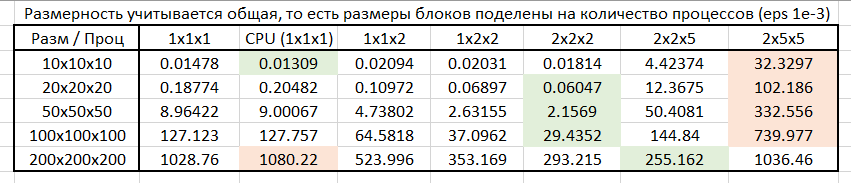
771. MPI\_Finalize();
773. free(lenFB);
774. free(lenUD);
775. free(lenLR);
777. free(adr\_sLeft);
778. free(adr\_rLeft);
779. free(adr\_sRight);
780. free(adr\_rRight);
782. free(adr\_sUP);
783. free(adr\_rUP);
784. free(adr\_sDown);
785. free(adr\_rDown);
787. free(adr\_sFront);
788. free(adr\_rFront);
789. free(adr\_sBack);
790. free(adr\_rBack);
792. free(data);
793. free(next);
794. free(buffer);
796. return 0;
797. }

**Результаты**

Как-то так.



Для сравнения – результаты из ЛР7



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.