3МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа №7 (1)**

**по курсу «Параллельная Обработка Данных»**

**Message Passing Interface (MPI)**

Выполнил: Иларионов Д.А.

Группа: М8О-408Б-17

Преподаватели: Крашенинников К.Г.,

Морозов А.Ю.

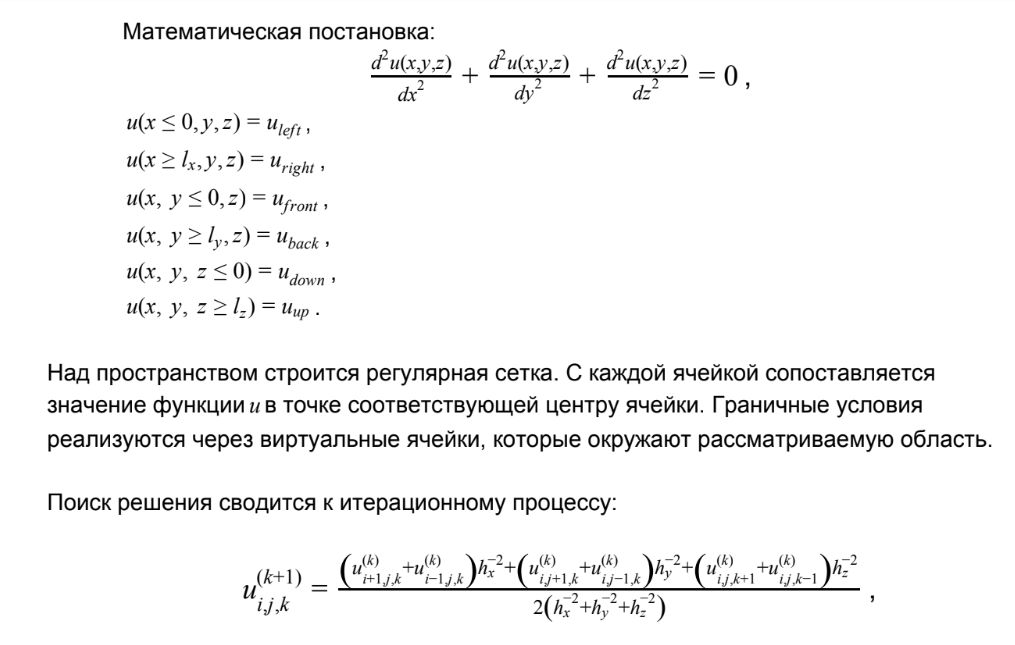
Москва, 2020

**Условие**

Знакомство с технологией MPI. Реализация метода Якоби.

Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными

условиями первого рода.



Запись результатов в файл должна осуществляться одним процессом.

Необходимо использовать последовательную пересылку данных по частям на

пишущий процесс.

**Вариант 4. обмен граничными слоями через isend/irecv, контроль сходимости allgather**

**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:**

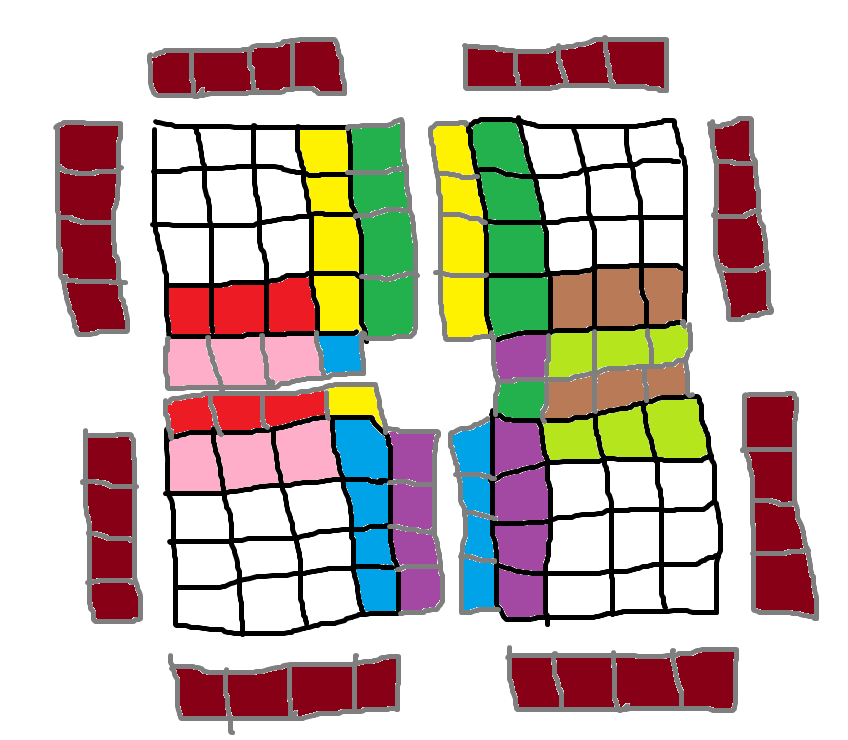
* Процессор: Intel Core i7-8750H 2.20GHz x 6
* Оперативная память: 16 ГБ
* SSD: 128 ГБ
* HDD: 1000 ГБ

**Программное обеспечение:**

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

**Метод решения**

Для удобства, я использовал макросы, с помощью которых можно определить текущую позицию элемента по x, y, z, или наоборот – его номер. У нас есть один длинный массив, в котором хранятся данные. Он одномерный, поэтому, обратиться к элементу можно только по формуле, чем и помогают макросы. Для каждого блока процесса у нас данные разные. Поэтому, сначала мы инициализируем MPI, вводим данные, если у нас нулевой (главный) процесс. Передаем их всем остальным процессам с помощью BCast. В некоторых местах ставим барьеры, чтобы процессы не опережали другие и не делали то, что еще рано делать. Потом идет цикл, до тех пор, пока у нас разница больше eps. Если количество процессов, задаваемых в аргументах не равно произведению блоков, то программа выводит, что неправильные данные и прекращает свою работу. В цикле мы сначала передаем данные – буфера граничных элементов, которые нужны нам для расчетов. В качестве такого буфера выступает двумерная сетка – передняя и задняя сторона блока, левая и правая. Верхняя и нижняя. Так как трехмерную визуализацию нарисовать трудно, я покажу на двухмерном примере.



В пэинте получилось довольно плохо, надо было Adobe Animate использовать.. Но ладно. Одинаковыми цветами (кроме белого и бордового) обозначены одинаковые группы элементов. То есть – буфера, граничные элементы. В файле они тоже присутствуют, поэтому к размеру блока по каждому измерению прибавляется 2. Макросом индексация таких элементов (-1 и N). N – размерность блока по опред. измерению. И суть самой программы и MPI в целом состоит как раз в передаче таких вот стенок. Сначала идет передача всех элементов для определенных процессов. Далее, чтобы не терять время – вычисляем внутренность блока – то есть элементы, для которых нам в вычислении не нужны граничные элементы. А вот ожидание и принятие идет в 2 этапа. Тк если сразу все стороны вычислять, то почему-то программа виснет на 30 тесте при размере сетки 2х2х2 и блоков 200х200х200. Не знаю честно, чем это вызвано, но заметил, что это происходит, когда одновременно принимаются и четные и нечетные элементы. Поэтому, я сделал так. Сначала мы принимаем все четные стороны (правая, нижняя, задняя). Кладем в data. Если нет блока по соседству (бордовые элементы), то эти элементы нам даны в начальном условии. Потом уже, мы принимаем нечетные элементы, и довычисляем наш блок. Ищем края (в 3 этапа – передняя и задняя стенка, затем – верхняя и нижнаяя, и потом левая и правая) – элементы для одного измерения – 1 и N-1. После этого, с помощью Allgather собираем все ошибки, ищем максимальную по всем процессам и продолжаем наш цикл во всех процессах, пока ошибка не будем меньше eps. Затем – отсылаем построчно наши элементы и выводим их в файл. Затем – finalize и освобождаем память.

**Описание программы**

**Файл main.cpp**

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #include <mpi.h>
5. #include <iostream>
6. #include <string>
7. #include <algorithm>
9. using namespace std;
11. int p1, p2, p3;
12. int g1, g2, g3;
14. // Index inside the block
15. #define \_i(i, j, k) ((k + 1) \* ((g2 + 2) \* (g1 + 2)) + (j + 1) \* (g1 + 2) + i + 1)
16. #define \_ix(id) (((id) % (g1 + 2)) - 1)
17. #define \_iy(id) ((((id) % ((g1 + 2) \* (g2 + 2))) / (g1 + 2)) - 1)
18. #define \_iz(id) (((id) / ((g1 + 2)\*(g2 + 2))) - 1)
20. // Index by processes
21. #define \_ib(i, j, k) ((k) \* (p1 \* p2) + (j) \* p1 + (i))
22. #define \_ibx(id) ((id) % p1)
23. #define \_iby(id) (((id) % (p1 \* p2)) / p1)
24. #define \_ibz(id) ((id) / (p1 \* p2))

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

30. int main(int argc, char\*\* argv) {
31. std::ios::sync\_with\_stdio(false);
32. string outFile;
34. int id;
35. int ib, jb , kb;
36. int i, j, k, iter;
37. int numproc, proc\_name\_len;
38. char proc\_name[MPI\_MAX\_PROCESSOR\_NAME];
40. double eps;
41. double lx, ly, lz;
42. double hx, hy, hz;
43. double Udown, Uup, Uleft, Uright, Ufront, Uback;
44. double U0;
45. double \*data, \*temp, \*next;
46. double \*bufferUPOut, \*bufferRightOut, \*bufferFrontOut;
47. double\* bufferUPIn, \* bufferRightIn, \* bufferFrontIn;
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);
75. //send data to all processes
76. MPI\_Bcast(&p1, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
77. MPI\_Bcast(&p2, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
78. MPI\_Bcast(&p3, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
80. MPI\_Bcast(&g1, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
81. MPI\_Bcast(&g2, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
82. MPI\_Bcast(&g3, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
84. MPI\_Bcast(&eps, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
86. MPI\_Bcast(&lx, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
87. MPI\_Bcast(&ly, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
88. MPI\_Bcast(&lz, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
90. MPI\_Bcast(&Udown, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
91. MPI\_Bcast(&Uup, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
92. MPI\_Bcast(&Uleft, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
93. MPI\_Bcast(&Uright, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
94. MPI\_Bcast(&Ufront, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
95. MPI\_Bcast(&Uback, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);
96. MPI\_Bcast(&U0, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);


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


111. //block id by coordinates
112. ib = \_ibx(id);
113. jb = \_iby(id);
114. kb = \_ibz(id);
116. iter = 0;
118. //find hs
119. hx = lx / ((double)p1 \* (double)g1);
120. hy = ly / ((double)p2 \* (double)g2);
121. hz = lz / ((double)p3 \* (double)g3);

124. data = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2) \* (g3 + 2));
125. next = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2) \* (g3 + 2));
127. bufferFrontOut = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2));  // wall1
128. bufferRightOut = (double\*)malloc(sizeof(double) \* (g2 + 2) \* (g3 + 2));  // wall2
129. bufferUPOut = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g3 + 2));  // wall3
131. bufferFrontIn = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g2 + 2));  // wall1
132. bufferRightIn = (double\*)malloc(sizeof(double) \* (g2 + 2) \* (g3 + 2));  // wall2
133. bufferUPIn = (double\*)malloc(sizeof(double) \* (g1 + 2) \* (g3 + 2));  // wall3
135. bufferIString = (double\*)malloc(sizeof(double) \* g1);  // wall3
137. //make buffer
139. int buffer\_size;
141. MPI\_Pack\_size((g1 + 2) \* (g2 + 2) \* (g3 + 2), MPI\_DOUBLE, MPI\_COMM\_WORLD, &buffer\_size);
143. buffer\_size = 2 \* (buffer\_size + MPI\_BSEND\_OVERHEAD); //6 edges

146. double\* buffer = (double\*)malloc(buffer\_size);
148. MPI\_Buffer\_attach(buffer, buffer\_size);
150. //block init
151. for (int a = 0; a < g1; ++a) {
152. for (int b = 0; b < g2; ++b) {
153. for (int c = 0; c < g3; ++c) {
154. data[\_i(a, b, c)] = U0;
155. }
156. }
157. }
158. //requests
159. MPI\_Request send\_request1, recv\_request1; //output
161. MPI\_Request send\_request1\_1, recv\_request1\_1;
162. MPI\_Request send\_request2\_1, recv\_request2\_1;
163. MPI\_Request send\_request3\_1, recv\_request3\_1;
165. MPI\_Request send\_request1\_2, recv\_request1\_2;
166. MPI\_Request send\_request2\_2, recv\_request2\_2;
167. MPI\_Request send\_request3\_2, recv\_request3\_2;

170. double\* errors;
171. errors = (double\*)malloc(numproc \* sizeof(double));
173. //string debug\_name = "process\_debug" + to\_string(id) + ".txt";


177. double maxErr = 0;
178. do  {
179. //send and get data
180. MPI\_Barrier(MPI\_COMM\_WORLD);

183. if (ib > 0 && ib + 1 < p1) { //both left and right
184. for (k = 0; k < g3; ++k) {
185. for (j = 0; j < g2; ++j) {
186. bufferRightIn[j + k \* g2] = data[\_i(0, j, k)];
187. bufferRightOut[j + k \* g2] = data[\_i(g1 - 1, j, k)];
188. }
189. }
190. MPI\_Isend(bufferRightIn, g2\* g3, MPI\_DOUBLE, \_ib(ib - 1, jb, kb), 0, MPI\_COMM\_WORLD, &send\_request1\_1);
191. MPI\_Isend(bufferRightOut, g2\* g3, MPI\_DOUBLE, \_ib(ib + 1, jb, kb), 0, MPI\_COMM\_WORLD, &send\_request1\_2);
192. }
193. else if (ib > 0) { //only left side
194. for (k = 0; k < g3; ++k) {
195. for (j = 0; j < g2; ++j) {
196. bufferRightIn[j + k \* g2] = data[\_i(0, j, k)];
197. }
198. }
199. MPI\_Isend(bufferRightIn, g2 \* g3, MPI\_DOUBLE, \_ib(ib - 1, jb, kb), 0, MPI\_COMM\_WORLD, &send\_request1\_1);
200. }
201. else if (ib + 1 < p1) { //only right side
202. for (k = 0; k < g3; ++k) {
203. for (j = 0; j < g2; ++j) {
204. bufferRightOut[j + k \* g2] = data[\_i(g1 - 1, j, k)];
205. }
206. }
207. MPI\_Isend(bufferRightOut, g2 \* g3, MPI\_DOUBLE, \_ib(ib + 1, jb, kb), 0, MPI\_COMM\_WORLD, &send\_request1\_2);
208. }

211. if (jb + 1 < p2 && jb > 0) { //both down and up
212. for (k = 0; k < g3; ++k) {
213. for (i = 0; i < g1; ++i) {
214. bufferUPIn[i + k \* g1] = data[\_i(i, 0, k)];
215. bufferUPOut[i + k \* g1] = data[\_i(i, g2 - 1, k)];
216. }
217. }
218. MPI\_Isend(bufferUPIn, g1\* g3, MPI\_DOUBLE, \_ib(ib, jb - 1, kb), 0, MPI\_COMM\_WORLD, &send\_request2\_2);
219. MPI\_Isend(bufferUPOut, g1\* g3, MPI\_DOUBLE, \_ib(ib, jb + 1, kb), 0, MPI\_COMM\_WORLD, &send\_request2\_1);
220. }
221. else if (jb > 0) { //only up side
222. for (k = 0; k < g3; ++k) {
223. for (i = 0; i < g1; ++i) {
224. bufferUPIn[i + k \* g1] = data[\_i(i, 0, k)];
225. }
226. }
227. MPI\_Isend(bufferUPIn, g1 \* g3, MPI\_DOUBLE, \_ib(ib, jb - 1, kb), 0, MPI\_COMM\_WORLD, &send\_request2\_2);
228. }
229. else if (jb + 1 < p2) { //only down side
230. for (k = 0; k < g3; ++k) {
231. for (i = 0; i < g1; ++i) {
232. bufferUPOut[i + k \* g1] = data[\_i(i, g2 - 1, k)];
233. }
234. }
235. MPI\_Isend(bufferUPOut, g1\* g3, MPI\_DOUBLE, \_ib(ib, jb + 1, kb), 0, MPI\_COMM\_WORLD, &send\_request2\_1);
236. }

239. if (kb + 1 < p3 && kb > 0) { //both back and front
240. for (j = 0; j < g2; ++j) {
241. for (i = 0; i < g1; ++i) {
242. bufferFrontIn[i + j \* g1] = data[\_i(i, j, 0)];
243. bufferFrontOut[i + j \* g1] = data[\_i(i, j, g3 - 1)];
244. }
245. }
246. MPI\_Isend(bufferFrontIn, g1\* g2, MPI\_DOUBLE, \_ib(ib, jb, kb - 1), 0, MPI\_COMM\_WORLD, &send\_request3\_2);
247. MPI\_Isend(bufferFrontOut, g1\* g2, MPI\_DOUBLE, \_ib(ib, jb, kb + 1), 0, MPI\_COMM\_WORLD, &send\_request3\_1);
248. }
249. else if (kb > 0) { //only front side
250. for (j = 0; j < g2; ++j) {
251. for (i = 0; i < g1; ++i) {
252. bufferFrontIn[i + j \* g1] = data[\_i(i, j, 0)];
253. }
254. }
255. MPI\_Isend(bufferFrontIn, g1\* g2, MPI\_DOUBLE, \_ib(ib, jb, kb - 1), 0, MPI\_COMM\_WORLD, &send\_request3\_2);
256. }
257. else if (kb + 1 < p3) { //only back side
258. for (j = 0; j < g2; ++j) {
259. for (i = 0; i < g1; ++i) {
260. bufferFrontOut[i + j \* g1] = data[\_i(i, j, g3 - 1)];
261. }
262. }
263. MPI\_Isend(bufferFrontOut, g1 \* g2, MPI\_DOUBLE, \_ib(ib, jb, kb + 1), 0, MPI\_COMM\_WORLD, &send\_request3\_1);
264. }

267. //while wait for data
269. //iterational function
270. double epsTemp[1];
271. epsTemp[0] = 0;
273. for (k = 1; k < g3 - 1; ++k) {
274. for (j = 1; j < g2 - 1; ++j) {
275. for (i = 1; i < g1 - 1; ++i) {
276. next[\_i(i, j, k)] = 0.5 \* ((data[\_i(i + 1, j, k)] + data[\_i(i - 1, j, k)]) / (hx \* hx) +
277. (data[\_i(i, j + 1, k)] + data[\_i(i, j - 1, k)]) / (hy \* hy) +
278. (data[\_i(i, j, k + 1)] + data[\_i(i, j, k - 1)]) / (hz \* hz)) /
279. (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));
280. epsTemp[0] = max(epsTemp[0], abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]));
281. }
282. }
283. }
285. //wait for data
287. if (ib > 0) { //only left side
288. MPI\_Wait(&send\_request1\_1, &status);
289. }
291. if (jb > 0) { //only up side
292. MPI\_Wait(&send\_request2\_2, &status);
293. }
295. if (kb > 0) { //only front side
296. MPI\_Wait(&send\_request3\_2, &status);
297. }


301. //set new data
303. if (ib + 1 < p1) { //get right side
304. MPI\_Irecv(bufferRightIn, g2 \* g3, MPI\_DOUBLE, \_ib(ib + 1, jb, kb), 0, MPI\_COMM\_WORLD, &recv\_request1\_2);
305. MPI\_Wait(&recv\_request1\_2, &status);
306. for (k = 0; k < g3; ++k) {
307. for (j = 0; j < g2; ++j) {
308. data[\_i(g1, j, k)] = bufferRightIn[j + k \* g2];
309. }
310. }
311. }
312. else {
313. for (k = 0; k < g3; ++k) {
314. for (j = 0; j < g2; ++j) {
315. data[\_i(g1, j, k)] = Uright;
316. next[\_i(g1, j, k)] = Uright;
317. }
318. }
319. }
321. if (jb + 1 < p2) { //get down side
322. MPI\_Irecv(bufferUPIn, g1 \* g3, MPI\_DOUBLE, \_ib(ib, jb + 1, kb), 0, MPI\_COMM\_WORLD, &recv\_request2\_1);
323. MPI\_Wait(&recv\_request2\_1, &status);
324. for (k = 0; k < g3; ++k) {
325. for (i = 0; i < g1; ++i) {
326. data[\_i(i, g2, k)] = bufferUPIn[i + k \* g1];
327. }
328. }
329. }
330. else {
331. for (k = 0; k < g3; ++k) {
332. for (i = 0; i < g1; ++i) {
333. data[\_i(i, g2, k)] = Udown;
334. next[\_i(i, g2, k)] = Udown;
335. }
336. }
337. }

340. if (kb + 1 < p3) { //get back side
341. MPI\_Irecv(bufferFrontIn, g1 \* g2, MPI\_DOUBLE, \_ib(ib, jb, kb + 1), 0, MPI\_COMM\_WORLD, &recv\_request3\_1);
342. MPI\_Wait(&recv\_request3\_1, &status);
343. for (j = 0; j < g2; ++j) {
344. for (i = 0; i < g1; ++i) {
345. data[\_i(i, j, g3)] = bufferFrontIn[i + j \* g1];
346. }
347. }
348. }
349. else {
350. for (j = 0; j < g2; ++j) {
351. for (i = 0; i < g1; ++i) {
352. data[\_i(i, j, g3)] = Uback;
353. next[\_i(i, j, g3)] = Uback;
354. }
355. }
356. }


360. if (ib + 1 < p1) { //only right side
361. MPI\_Wait(&send\_request1\_2, &status);
362. }
363. if (jb + 1 < p2) { //only down side
364. MPI\_Wait(&send\_request2\_1, &status);
365. }
366. if (kb + 1 < p3) { //only back side
367. MPI\_Wait(&send\_request3\_1, &status);
368. }


372. if (ib > 0) { //get left side
373. MPI\_Irecv(bufferRightOut, g2\* g3, MPI\_DOUBLE, \_ib(ib - 1, jb, kb), 0, MPI\_COMM\_WORLD, &recv\_request1\_1);
374. MPI\_Wait(&recv\_request1\_1, &status);
375. for (k = 0; k < g3; ++k) {
376. for (j = 0; j < g2; ++j) {
377. data[\_i(-1, j, k)] = bufferRightOut[j + k \* g2];
378. }
379. }
380. }
381. else {
382. for (k = 0; k < g3; ++k) {
383. for (j = 0; j < g2; ++j) {
384. data[\_i(-1, j, k)] = Uleft;
385. next[\_i(-1, j, k)] = Uleft;
386. }
387. }
388. }

391. if (jb > 0) { //get up side
392. MPI\_Irecv(bufferUPOut, g1\* g3, MPI\_DOUBLE, \_ib(ib, jb - 1, kb), 0, MPI\_COMM\_WORLD, &recv\_request2\_2);
393. MPI\_Wait(&recv\_request2\_2, &status);
394. for (k = 0; k < g3; ++k) {
395. for (i = 0; i < g1; ++i) {
396. data[\_i(i, -1, k)] = bufferUPOut[i + k \* g1];
397. }
398. }
399. }
400. else {
401. for (k = 0; k < g3; ++k) {
402. for (i = 0; i < g1; ++i) {
403. data[\_i(i, -1, k)] = Uup;
404. next[\_i(i, -1, k)] = Uup;
405. }
406. }
407. }

410. if (kb > 0) { //get front side
411. MPI\_Irecv(bufferFrontOut, g1\* g2, MPI\_DOUBLE, \_ib(ib, jb, kb - 1), 0, MPI\_COMM\_WORLD, &recv\_request3\_2);
412. MPI\_Wait(&recv\_request3\_2, &status);
413. for (j = 0; j < g2; ++j) {
414. for (i = 0; i < g1; ++i) {
415. data[\_i(i, j, -1)] = bufferFrontOut[i + j \* g1];
416. }
417. }
418. }
419. else {
420. for (j = 0; j < g2; ++j) {
421. for (i = 0; i < g1; ++i) {
422. data[\_i(i, j, -1)] = Ufront;
423. next[\_i(i, j, -1)] = Ufront;
424. }
425. }
426. }

429. //MPI\_Barrier(MPI\_COMM\_WORLD);
431. //for edges
433. int\* i\_start, \* j\_start, \* k\_start;
435. k\_start = (int\*)malloc(sizeof(int) \* 2);
436. k\_start[0] = 0;
437. k\_start[1] = g3 - 1;
439. j\_start = (int\*)malloc(sizeof(int) \* 2);
440. j\_start[0] = 0;
441. j\_start[1] = g2 - 1;
443. i\_start = (int\*)malloc(sizeof(int) \* 2);
444. i\_start[0] = 0;
445. i\_start[1] = g1 - 1;
447. //k-edges
448. for (int k\_s = 0; k\_s < 2; ++k\_s) {
449. k = k\_start[k\_s];
450. for (j = 0; j < g2; ++j) {
451. for (i = 0; i < g1; ++i) {
452. next[\_i(i, j, k)] = 0.5 \* ((data[\_i(i + 1, j, k)] + data[\_i(i - 1, j, k)]) / (hx \* hx) +
453. (data[\_i(i, j + 1, k)] + data[\_i(i, j - 1, k)]) / (hy \* hy) +
454. (data[\_i(i, j, k + 1)] + data[\_i(i, j, k - 1)]) / (hz \* hz)) /
455. (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));
456. epsTemp[0] = max(epsTemp[0], abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]));
457. }
458. }
459. }


463. //j-edges
464. for (k = 0; k < g3; ++k) {
465. for (int j\_s = 0; j\_s < 2; ++j\_s) {
466. j = j\_start[j\_s];
467. for (i = 0; i < g1; ++i) {
468. next[\_i(i, j, k)] = 0.5 \* ((data[\_i(i + 1, j, k)] + data[\_i(i - 1, j, k)]) / (hx \* hx) +
469. (data[\_i(i, j + 1, k)] + data[\_i(i, j - 1, k)]) / (hy \* hy) +
470. (data[\_i(i, j, k + 1)] + data[\_i(i, j, k - 1)]) / (hz \* hz)) /
471. (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));
472. epsTemp[0] = max(epsTemp[0], abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]));
473. }
474. }
475. }

478. //i-edges
479. for (k = 0; k < g3; ++k) {
480. for (j = 0; j < g2; ++j) {
481. for (int i\_s = 0; i\_s < 2; ++i\_s) {
482. i = i\_start[i\_s];
483. next[\_i(i, j, k)] = 0.5 \* ((data[\_i(i + 1, j, k)] + data[\_i(i - 1, j, k)]) / (hx \* hx) +
484. (data[\_i(i, j + 1, k)] + data[\_i(i, j - 1, k)]) / (hy \* hy) +
485. (data[\_i(i, j, k + 1)] + data[\_i(i, j, k - 1)]) / (hz \* hz)) /
486. (1.0 / (hx \* hx) + 1.0 / (hy \* hy) + 1.0 / (hz \* hz));
487. epsTemp[0] = max(epsTemp[0], abs(next[\_i(i, j, k)] - data[\_i(i, j, k)]));
488. }
489. }
490. }

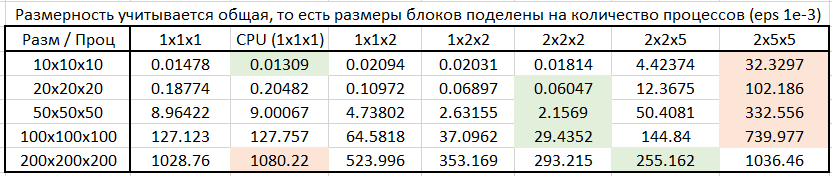
493. MPI\_Allgather(epsTemp, 1, MPI\_DOUBLE, errors, 1, MPI\_DOUBLE, MPI\_COMM\_WORLD);
494. epsTemp[0] = 0;
495. for (i = 0; i < numproc; ++i) {
496. epsTemp[0] = max(epsTemp[0], errors[i]);
497. }
499. temp = next;
500. next = data;
501. data = temp;
503. maxErr = epsTemp[0];
505. iter += 1;
507. }while (maxErr >= eps);

510. MPI\_Barrier(MPI\_COMM\_WORLD);
512. if (id != 0) {
513. for (k = 0 ; k < g3 ; ++k) {
514. for (j = 0; j < g2; ++j) {
515. for (i = 0; i < g1; ++i) {
516. bufferIString[i] = data[\_i(i, j, k)];
517. }
518. MPI\_Isend(bufferIString, g1, MPI\_DOUBLE, 0, id, MPI\_COMM\_WORLD, &send\_request1);
519. MPI\_Wait(&send\_request1, &status);
520. }
521. }
522. }
523. else {
524. cerr << "Process GRID: " << p1 << "x" << p2 << "x" << p3 << "**\n**";
525. cerr << "Num GRID: " << g1 << "x" << g2 << "x" << g3 << "**\n**";
526. cerr << "Eps: " << eps << "**\n**";
527. cerr << "lx: " << lx << " ly: " << ly << " lz: " << lz << "**\n**";
528. cerr << "Us: " << Ufront << " , " << Uback << " , " << Uleft << " , " << Uright << " , " << Uup << " , " << Udown << "**\n**";
529. cerr << "U0: " << U0 << "**\n**";
530. cerr << "Iterations: " << iter << "**\n**";
532. FILE\* File = fopen(outFile.c\_str(), "w+");
534. for (kb = 0; kb < p3; ++kb) {
535. for (k = 0; k < g3; ++k) {
536. for (jb = 0; jb < p2; ++jb) {
537. for (j = 0; j < g2; ++j) {
538. for (ib = 0; ib < p1; ++ib) {
539. if (\_ib(ib, jb, kb) == 0) {
540. for (i = 0; i < g1; ++i) {
541. bufferIString[i] = data[\_i(i, j, k)];
542. printf("%.6e ", bufferIString[i]);
543. }
544. if (ib + 1 == p1) {
545. printf("**\n**");
546. if (j + 1 == g2) {
547. printf("**\n**");
548. }
549. }
550. }
551. else {
552. MPI\_Irecv(bufferIString, g1, MPI\_DOUBLE, \_ib(ib, jb, kb), \_ib(ib, jb, kb), MPI\_COMM\_WORLD, &recv\_request1);
553. MPI\_Wait(&recv\_request1, &status);
554. for (i = 0; i < g1; ++i) {
555. printf("%.6e ", bufferIString[i]);
556. }
557. if (ib + 1 == p1) {
558. printf("**\n**");
559. if (j + 1 == g2) {
560. printf("**\n**");
561. }
562. }
563. }
564. }
565. }
566. }
567. }
568. }
569. fclose(File);
570. }


574. MPI\_Buffer\_detach(buffer, &buffer\_size);
575. MPI\_Finalize();
577. free(bufferRightIn);
578. free(bufferUPIn);
579. free(bufferFrontIn);
581. free(bufferRightOut);
582. free(bufferUPOut);
583. free(bufferFrontOut);
585. free(data);
586. free(next);
587. free(buffer);
589. return 0;
590. }

**Результаты**

Как-то так.



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**

Сущий ад. Серьезно. Такое ощущение, как будто я делал целую курсовую. Если честно, то некоторые курсовые были даже легче. С данной лабораторной я возился целые две недели. И постоянно она не проходила на каких-то тестах. Были проблемы и с тем, что неправильно переносил элементы. Но чаще всего – проблема в том, что процессы не успевали копировать все данные, и в итоге граничные элементы заполнялись всяким мусором. В итоге, и выходило намного больше итераций, и ответ не совпадал совсем, потому что с других элементов шло приближение.

Почему она висла при конфигурации 2x2x2, 200х200х200, я честно до сих пор не понимаю. Может, снова, не успевало все копировать. Но как-то раз ко мне пришла идея поделить на четные и нечетные и получать их раздельно, и вот чудо, она прошла! Еще мне однокурсник посоветовал сначала вычислять внутренность блока, а затем края – это немного должно повысить производительность. Пока идет передача элементов, мы не теряем время и вычисляем большую часть значений (ну если куб не 3х3х3 или 5х5х5 или еще меньше). Короче, убил оочень много нервов с этой лабой, однако, я узнал некоторые новые вещи. Научился распараллеливать программу, при 8 процессах идет намного быстрее, чем при одном. Еще дело было в моем варианте – Isend – неблокирующая передача, поэтому приходилось ставить постоянно Wait, с которым очень легко запутаться. При обычном BSend, у меня бы таких проблем не возникло. Но я очень рад, что эта седьмая лаба уже позади, и мне осталось ее только защитить. Вот только теперь траблы с 8-ой лабой (с 9 не было так сложно). Думаю, что похожая технология может пригодиться при вычислении математических задач с огромными данными (как эта). Кстати, огромное спасибо преподавателю за то, что посоветовал мне cerr. Могу теперь смотреть на каких тестах не проходит, и именно это мне помогло найти ошибки.