МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ М.В. ЛОМОНОСОВА

КУРС: «ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ И СУПЕРКОМПЬЮТЕРЫ»

**Рекурсивная координатная бисекция**

**Выполнил:**

**Сактаганов Нуржан (аспирант)**

**Дата подачи: 05.01.2018**

**Москва 2018**

Table of Contents

[Постановка задачи 3](#__RefHeading__113_1817480585)

[Метод решения 3](#__RefHeading___Toc1120_59742566)

[Описание вычислительной системы 4](#__RefHeading___Toc1122_59742566)

[Анализ полученных результатов 5](#__RefHeading___Toc1124_59742566)

[Приложение А. Последовательная программа 7](#__RefHeading___Toc1126_59742566)

[Приложение Б. Параллельная программа 8](#__RefHeading___Toc1128_59742566)

## Постановка задачи

Дана двумерная сетка, топологически эквивалентная индексному прямоугольнику n1xn2. Необходимо разбить сетку на k частей (доменов). Узел сетки с индексами (i, j) связан с существующими соседними по i и j узлами: (i-1, j), (i+1, j), (i, j-1), (i, j+1).

Сетка представляется в виде одмерного массива структур Point.

Point{

float coord[2];

int index;

}

P[n1\*n2];

Точки данной сетки имеют координаты P[i\*n2+j].coord[0] = x(i, j), P[i\*n2+j].coord[1] = y(i,j), где i = 0, …, n1-1, j = 0, …, n2-1. Индекс определяется соотношением P[i\*n2+j].index = i\*n2+j.

Для разбиения сетки нужно написать параллельную программу для вычислительной системы с распределенной памятью, обеспечивающую разбиение методом рекурсивной координатной бисекции узлов сетки на k доменов. Число вершин в доменах должно быть равно с точностью до одной вершины величине (n1\*n2/k).

На выходе должны получить:

1. файл, содержащий n1\*n2 строк, в каждой из которых 5 чисел: i j Xij Yij d // номера вершины в сетке, координаты вершины, номер домена [0,...,k-1]
2. число разрезанных рёбер
3. время выполнения декомпозиции (только декомпозиции, без учета формирования сетки и вывода результатов).

## Метод решения

Сначала опишем последовательный алгоритм. Далее на основе последовательного алгоритма будем описывать параллельный алгоритм.

**Последовательный алгоритм** очень простой. На вход даются массив структур в описанном выше формате и количество доменов на которое нужно разбить. Мы сначала сортируем точки по заданной координате. После этого раскладываем k = k1 + k2, где k1 = (k + 1) / 2, k2 = k-k1. Далее делим массив в пропорции k1:k2. Меняем координату, по которой нужно сортировать массив, и рекурсивно вызываем на первой половине с параметром k := k1, и на второй половине k := k2. После чего обратно вооставнавливаем координату для сортировки. Рекурсия останавливается в моменте, когда у нас k = 1. Это означает, что все точки массива принадлежат одному и тому же домену.

**Параллельный алгоритм** делает по сути точно то же самое что и последовательный. Будем считать, что каждый процесс запущен на отдельном узле, и что на узле находится один процессор.

Допустим, что распределенный массив расположен на n процессорах. На каждом процессоре находится по part\_size точек. И пусть имеется число k — количество доменов, на которое нужно разбить. Тогда, сначала сортируем глобальный массив по заданной координате (алгоритм сортировки Бэтчера). Далее, точно так же раскладываем k=k1+k2. Делим глобальный массив в пропорции k1:k2. Находим разбивающий элемент — первый элемент второй половинки глобального массива. Множество процессоров надо разбить на два подмножества, первое подмножество будет обрабатывать первую половинку, второе — вторую. Узел, который содержит в себе разбивающий элемент отнесем ко второму подмножеству. Поэтому, все элементы до разбивающего элемента нужно равномерно распределить по процессорам первого подножества. Причем, некоторые процессоры первого подмножества нужно дополнить фиктивными элементами, чтобы у всех было одинаковое количество элементов (это требуется для сортировки бэтчера). А в процессорах второго подмножества нужно сделать перераспределение элементов в связи с уменьшением их количества в этом множестве. После этого меняем координату для соортировки и применяем описанный алгоритм рекурсивно для двух подмножеств процессоров с соответствующими параметрами k1 и k2. После возврата из рекурсии надо не забыть воостановить координату для сортировки. Когда k или количество процессоров будет равно 1 — это означает конец рекурсии параллельного алгоритма, теперь уже на каждом узле выполним последовательный алгоритм с имеющим значением k.

Случай, когда разбивающий элемент будет на нулевом процессоре обрабатывается чуть по-другому. В этом случае нулевой процессор отнесем к первому подмножеству, а остальные — ко второму. Все элементы, начиная с разбивающего, распределим по процессорам из второго подмножества, пометив отправленные как фиктивные и добавив фиктивные по необходимости.

**Подсчет числа разрезанных рёбер**. Возьмем произвольный домен. Для каждой его точки проверим принадлежность каждой из четырех соседних точек данному домену. Если какой-то сосед не будет принадлежать данному домену — это означает что есть разрез ребра. Если для каждого домена посчитаем количество разрезов, и потом просуммируем — получим удвоенное количество всех разрезанных рёбер.

Будем считать, что k больше количества процессоров. Тогда не будет случая, когда один домен будет размещен на нескольких процессорах. Это означает, что каждый домен будет целиком помещаться внутри одного узла. Тогда мы можем для каждого домена локально посчитать количество разрезов. После чего легко получить количество всех разрезанных ребер.

## Описание вычислительной системы

Вычислительная система представляет собой комплекс IBM Blue Gene/P. В составе имеется две стойки по 1024 четырехъядерных процессоров в каждой стойке. Пиковая производтельность — 27.9 терафлопс. Размер оперативной памяти — 2 Гб на каждый процессор. Каждое ядро модели PowerPC 450 с рабочей частотой 850 МГц, с 32 битной адресацией. Производительность ядра — 3.4 Гфлопс.

Описание коммуникационной сети:

* сеть общего назначения, объединяющие все вычислительные узлы; предназначена для операций типа «точка-точка»
* вычислительный узел имеет двунаправленные связи с шестью соседями
* пропускная способность каждого соединения — 425 MB/s (5,1 GB/s для всех 12 каналов)
* латентность (ближайший сосед):
  + 32-байтный пакет: 0,1 μs
  + 256-байтный пакет: 0,8 μs

## Анализ полученных результатов

Ниже приведена таблица времен (в секундах) запусков на различном наборе сеток и процессоров (число доменов k = 256).

| Сетка | 1 процессор | 8 процессоров | 32 процессоров | 128 процессоров |
| --- | --- | --- | --- | --- |
| 4k x 2.5k | 79.0919 | 11.055 | 3.56367 | 1.3142 |
| 4k x 5k | 167.652 | 23.4848 | 7.44001 | 2.71177 |

Посчитаем ускорение.

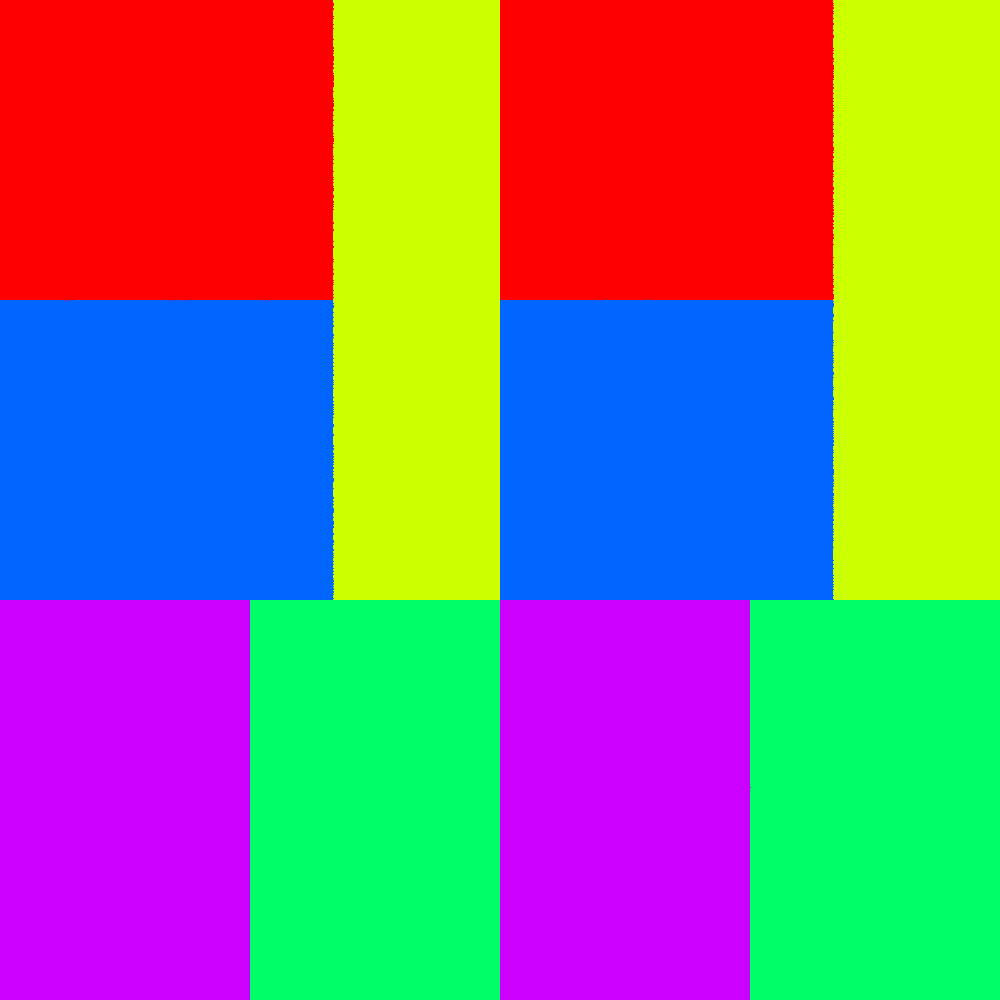
| Сетка | 1 процессор | 8 процессоров | 32 процессоров | 128 процессоров |
| --- | --- | --- | --- | --- |
| 4k x 2.5k | 1.0 | 7.1544 | 22.194 | 60.183 |
| 4k x 5k | 1.0 | 7.1387 | 22.5338 | 61.8238 |

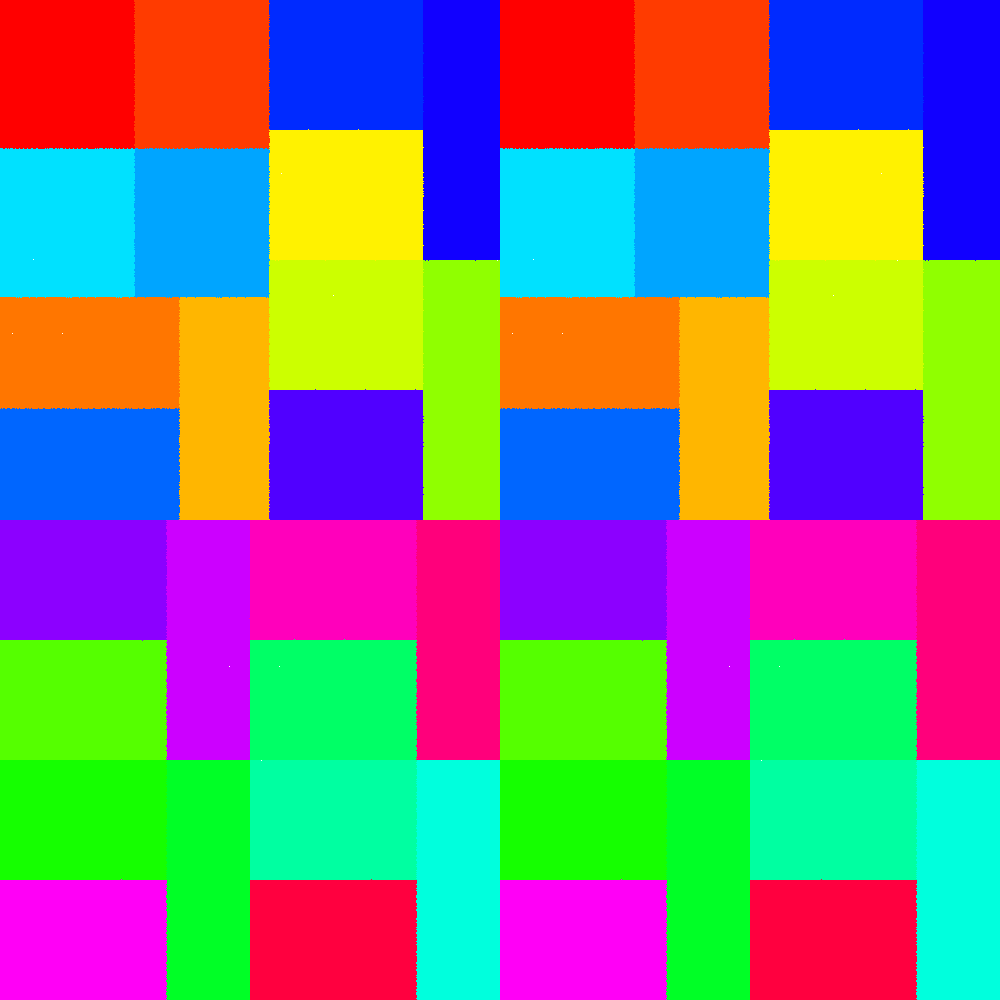
Посчитаем эффективность.

| Сетка | 1 процессор | 8 процессоров | 32 процессоров | 128 процессоров |
| --- | --- | --- | --- | --- |
| 4k x 2.5k | 100% | 89.43% | 69.36% | 48.02% |
| 4k x 5k | 100% | 89.23% | 70.42% | 48.30% |

Из таблиц видим, что программа хорошо параллелится и время выполнения продолжает сокращаться с ростом числа процессоров.

В качестве примера работы алгоритма на сетке 1000х1000 приведены две иллюстрации. В первом случае разбиение на 10 доменов с использованием 5 процессов. Во втором случае разбиение на 50 доменов с использованием 20 процессов.

  
Illustration 1: 1000x1000,d=10,p=5

  
Illustration 2: 1000x1000,d=50,p=20

## Приложение А. Последовательная программа

1. #pragma once
2. #include <algorithm>
3. #include "Point.hpp"
5. //struct represents that point with given index
6. //belongs to given domain
7. **typedef** **struct** \_point\_domain\_t {
8. Point point;
9. **int** domain;
10. } point\_domain\_t;
12. /\*
13. points - given array of points
14. points\_domain - elements belonging
15. length - length of points array
16. k - wanted domains count
17. lowest\_domain - lowest available domain number
18. \*/
19. **void** decompose(
20. Point \*points,
21. point\_domain\_t \*points\_domain,
22. **const** **int** length,
23. **const** **int** k,
24. **const** **int** lowest\_domain)
25. {
26. **if** (k == 1) {
27. **for** (**int** i = 0; i < length; ++i) {
28. points\_domain[i].domain = lowest\_domain;
29. points\_domain[i].point = points[i];
30. }
31. **return**;
32. }
34. std::sort(points, points + length);
36. **const** **int** k1 = (k + 1) / 2;
37. **const** **int** k2 = k - k1;
38. **const** **int** l1 = 1.0 \* k1/k \* length;
39. **const** **int** l2 = length - l1;
41. Point::switch\_sort\_way();
42. decompose(points, points\_domain, l1, k1, lowest\_domain);
43. decompose(points + l1, points\_domain + l1, l2, k2, lowest\_domain + k1);
44. Point::switch\_sort\_way();
45. }

## Приложение Б. Параллельная программа

1. #include <cstring>
2. #include "Point.hpp"
3. #include "decompose.hpp"
4. #include "parallel\_sort.hpp"
6. //represents a 'vector' with given length (i.e. size) and ptr
7. **typedef** **struct** \_points\_domain\_t {
8. **int** k;
9. **int** procs\_num;
10. **int** size;
11. point\_domain\_t \*info;
12. } points\_domain\_t;
14. points\_domain\_t \_parallel\_decompose(
15. MPI::Intracomm &comm,
16. MPI::Datatype &datatype,
17. Point \*points,
18. **int** part\_size,
19. **int** k,
20. **int** lowest\_domain,
21. **int** \*sendcounts\_buffer, // for using in Scatterv
22. **int** \*displacements\_buffer // for using in Scatterv
23. );
25. **int** calculate\_max\_part\_size(**int** procs\_num, **int** part\_size, **int** k);
27. points\_domain\_t parallel\_decompose(
28. MPI::Intracomm &comm,
29. MPI::Datatype &datatype,
30. Point \*points,
31. **int** part\_size,
32. **int** k,
33. **int** lowest\_domain)
34. {
35. points\_domain\_t points\_domain;
37. **const** **int** procs\_num = comm.Get\_size();
38. **const** **int** max\_part\_size = calculate\_max\_part\_size(procs\_num, part\_size, k);
40. Point \*points\_copy = **new** Point[max\_part\_size];
41. memcpy(points\_copy, points, part\_size \* **sizeof**(Point));
43. //Now we can do everyting with this copy.
44. //However, passed user's array will remain unchanged.
46. **int** \*sendcounts\_buffer = **new** **int** [procs\_num];
47. **int** \*displacements\_buffer = **new** **int** [procs\_num];
49. points\_domain = \_parallel\_decompose(
50. comm, datatype, points\_copy,
51. part\_size, k, lowest\_domain,
52. sendcounts\_buffer,
53. displacements\_buffer
54. );
56. **delete** [] sendcounts\_buffer;
57. **delete** [] displacements\_buffer;
59. **delete** [] points\_copy;
61. **return** points\_domain;
62. };
64. points\_domain\_t \_parallel\_decompose(
65. MPI::Intracomm &comm,
66. MPI::Datatype &datatype,
67. Point \*points,
68. **int** part\_size,
69. **int** k,
70. **int** lowest\_domain,
71. **int** \*sendcounts\_buffer, // for using in Scatterv
72. **int** \*displacements\_buffer // for using in Scatterv
73. )
74. {
75. **const** **int** procs\_num = comm.Get\_size();
76. **const** **int** rank = comm.Get\_rank();
78. points\_domain\_t points\_domain;
80. **if** (procs\_num == 1 || k == 1) { // there is no more processes or domains
81. **if** (k > 1) {
82. **for** (**int** i = part\_size - 1; i >= 0; --i) {
83. **if** (points[i].index != -1) **continue**;
84. std::swap(points[i], points[part\_size - 1]);
85. --part\_size;
86. }
87. }
88. points\_domain.k = k;
89. points\_domain.procs\_num = procs\_num;
90. points\_domain.size = part\_size;
91. points\_domain.info = **new** point\_domain\_t[part\_size];
92. decompose(points, points\_domain.info, part\_size, k, lowest\_domain);
93. **return** points\_domain;
94. }
96. std::sort(points, points + part\_size); // sort each local array
97. parallel\_sort(comm, datatype, points, part\_size);
99. //now we have global sorted array of points
101. // searching global middle element's coordinates
102. **const** **long** **long** global\_length = part\_size \* procs\_num;
103. **const** **int** k1 = (k + 1) / 2;
104. **const** **int** k2 = k - k1;
105. **const** **long** **long** l1 = 1.0 \* k1 / k \* global\_length;
106. //const long long l2 = global\_length - l1;
108. **const** **int** middle\_proc = l1 / part\_size;
109. **const** **int** middle\_index = l1 % part\_size;
111. MPI::Cartcomm new\_comm;
112. **int** color; // 0 - left half, 1 - right half
113. **bool** i\_am\_sender = rank == middle\_proc;
114. **bool** i\_am\_receiver;
115. **int** send\_count;
116. **int** recv\_procs;
117. Point \*send\_points\_start;
119. **if** (middle\_proc > 0) {
120. // we have to send to left
121. i\_am\_receiver = rank < middle\_proc;
122. color = rank >= middle\_proc;
123. send\_count = middle\_index; // elements in position 0, ..., middle\_index-1
124. send\_points\_start = points; // send from begining
125. recv\_procs = middle\_proc;
126. } **else** { // middle\_proc == 0
127. // no way to send to left, only to right
128. i\_am\_receiver = rank > middle\_proc;
129. color = rank > middle\_proc;
130. send\_count = part\_size - middle\_index; // elements in position middle\_index, ..., part\_size-1
131. send\_points\_start = points + middle\_index; // send from middle\_index and so on
132. recv\_procs = procs\_num - 1;
133. }
135. // communicate only if required
136. **if** (send\_count > 0) {
137. memset(sendcounts\_buffer, 0, procs\_num \* **sizeof**(**int**));
138. memset(displacements\_buffer, 0, procs\_num \* **sizeof**(**int**));
140. **int** offset = middle\_proc == 0;
142. // init sendcounts
143. **for** (**int** i = 0; i < recv\_procs; ++i) {
144. sendcounts\_buffer[i + offset] = send\_count / recv\_procs;
145. }
147. **for** (**int** i = 0; i < send\_count % recv\_procs; ++i) {
148. sendcounts\_buffer[i + offset] += 1;
149. }
151. // init displacements
152. displacements\_buffer[offset] = 0;
153. **for** (**int** i = 1; i < recv\_procs; ++i) {
154. displacements\_buffer[i + offset] =
155. displacements\_buffer[i-1 + offset]
156. + sendcounts\_buffer[i-1 + offset];
157. }
159. comm.Scatterv(
160. send\_points\_start, // we send from here
161. sendcounts\_buffer,
162. displacements\_buffer,
163. datatype,
164. points + part\_size, // we receive to this buffer
165. sendcounts\_buffer[rank], // we receive this amount of points
166. datatype,
167. middle\_proc
168. );
170. **int** receiver\_part\_size = 0;
171. // in all receivers indentify phantom elements
172. **if** (i\_am\_receiver) {
173. **const** **int** recv\_n = send\_count / recv\_procs + (send\_count % recv\_procs > 0);
174. receiver\_part\_size = part\_size + recv\_n;
175. **const** **int** phantom\_elements = recv\_n - sendcounts\_buffer[rank];
177. **for** (**int** i = 0; i < phantom\_elements; ++i) {
178. points[receiver\_part\_size-1-i].index = -1;
179. }
180. }
182. **const** **bool** can\_compensate = middle\_proc > 0 && (procs\_num - recv\_procs > 1);
184. **if** (i\_am\_sender && !can\_compensate) {
185. // make sent elements phantom
186. **for** (**int** i = 0; i < send\_count; ++i) send\_points\_start[i].index = -1;
187. }
189. **if** (can\_compensate) {
190. // compensate sent elements
191. **const** **int** new\_procs\_num = procs\_num - recv\_procs;
192. **const** **long** **long** new\_length = 1L \* part\_size \* new\_procs\_num - send\_count;
193. **const** **int** new\_part\_size = new\_length / new\_procs\_num + (new\_length % new\_procs\_num > 0);
194. **const** **int** new\_middle\_proc\_points = part\_size - send\_count;
195. **const** **int** to\_compensate = new\_part\_size - new\_middle\_proc\_points;
196. **const** **int** compensators = new\_procs\_num - 1;
198. **int** \*recv\_counts = sendcounts\_buffer;
199. memset(recv\_counts, 0, procs\_num \* **sizeof**(**int**));
200. memset(displacements\_buffer, 0, procs\_num \* **sizeof**(**int**));
202. **const** **int** offset = middle\_proc + 1;
203. **for** (**int** i = 0; i < compensators; ++i) {
204. recv\_counts[offset + i] = to\_compensate / compensators;
205. }
207. **for** (**int** i = 0; i < to\_compensate % compensators; ++i) {
208. recv\_counts[offset + i] += 1;
209. }
211. displacements\_buffer[offset] = 0;
212. **for** (**int** i = 1; i < compensators; ++i) {
213. displacements\_buffer[offset + i] =
214. displacements\_buffer[offset + i-1]
215. + recv\_counts[offset + i-1];
216. }
218. **if** (rank == middle\_proc) {
219. memmove(points, points + send\_count, new\_middle\_proc\_points \* **sizeof**(Point));
220. }
222. comm.Gatherv(
223. points + part\_size - recv\_counts[rank], // from the tail
224. recv\_counts[rank], // how many
225. datatype,
226. points + new\_middle\_proc\_points,
227. recv\_counts,
228. displacements\_buffer,
229. datatype,
230. middle\_proc
231. );
233. // make redistributed as phantom
234. **for** (**int** i = 0; i < recv\_counts[rank]; ++i){
235. points[part\_size - 1 - i].index = -1;
236. }
238. **if** (rank >= middle\_proc) {
239. part\_size = new\_part\_size;
240. }
241. }
243. **if** (i\_am\_receiver) {
244. part\_size = receiver\_part\_size;
245. }
246. }
248. **if** (color == 0) {
249. k = k1;
250. } **else** { // color == 1
251. k = k2;
252. lowest\_domain += k1;
253. }
255. new\_comm = comm.Split(color, rank);
257. Point::switch\_sort\_way();
258. points\_domain = \_parallel\_decompose(
259. new\_comm, datatype, points,
260. part\_size, k, lowest\_domain,
261. sendcounts\_buffer,
262. displacements\_buffer
263. );
264. Point::switch\_sort\_way();
266. new\_comm.Free();
268. **return** points\_domain;
269. }
271. /\*      Trying predict max memory amount to prevent array reallocation.
272. So, algorithm is recursive. In worst case we have to redistribute
273. send\_n := part\_size-1 elements from sender. There is always only one sender
274. in each recursion step. Let's consider that the amount of receivers is recv\_p.
275. It means, every receiver has to add at most ceil(send\_n / recv\_p) elements.
276. There are at least recv\_p := n\_procs / 2 receivers, we finally get new part\_size
277. for each receiver: part\_size := part\_size + ceil(send\_n / (n\_procs / 2)).
278. Let's consider that we already have max recursion depth in recursion\_depth variable.
279. Then we can describe algorithm in pseudocode.
281. def max\_part\_size(part\_size, recursion\_depth, n\_procs):
282. while recursion\_depth > 0:
283. send\_n = part\_size - 1
284. recv\_p = n\_procs / 2
285. part\_size += ceil(send\_n / recv\_p)
286. n\_procs = ceil(n\_procs / 2)
287. return part\_size
289. Recursion depth depends from k and n\_procs.
290. Exactly, it equals to rounded up value of log(min(k, procs\_num)):
292. recursion\_depth := ceil(log(min(k, n\_procs), base=2)).
294. But it's too roughly. So, by modelying part\_size changing,
295. we can get exact value of max part size.
296. \*/
297. **int** calculate\_max\_part\_size(**int** procs\_num, **int** part\_size, **int** k)
298. {
299. **if** (procs\_num == 1 || k == 1) **return** part\_size;
300. // searching global middle element's coordinates
301. **const** **long** **long** global\_length = part\_size \* procs\_num;
302. **const** **int** k1 = (k + 1) / 2;
303. **const** **int** k2 = k - k1;
304. **const** **long** **long** l1 = 1.0 \* k1 / k \* global\_length;
305. //const long long l2 = global\_length - l1;
307. **const** **int** middle\_proc = l1 / part\_size;
308. **const** **int** middle\_index = l1 % part\_size;
310. **int** procs1, procs2;
311. **int** p1\_size, p2\_size;
312. p1\_size = p2\_size = part\_size;
314. **if** (middle\_proc > 0) {
315. **const** **int** send\_n = middle\_index;
316. **const** **int** recv\_p = middle\_proc;
317. **const** **int** recv\_n = send\_n / recv\_p + (send\_n % recv\_p > 0);
318. p1\_size += recv\_n;
320. procs1 = middle\_proc;
321. procs2 = procs\_num - procs1;
322. } **else** {// middle\_proc == 0
323. **const** **int** send\_n = part\_size - middle\_index;
324. **const** **int** recv\_p = procs\_num - 1;
325. **const** **int** recv\_n = send\_n / recv\_p + (send\_n % recv\_p > 0);
326. p2\_size += recv\_n;
328. procs1 = 1;
329. procs2 = procs\_num - 1;
330. }
332. p1\_size = calculate\_max\_part\_size(procs1, p1\_size, k1);
333. p2\_size = calculate\_max\_part\_size(procs2, p2\_size, k2);
335. part\_size = p1\_size > p2\_size ? p1\_size : p2\_size;
336. **return** part\_size;
337. }