ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**КУРСОВОЙ ПРОЕКТ**

по дисциплине “Параллельные вычислительные технологии”

на тему

**Разработка параллельной MPI-программы реализации алгоритма Флойда**

|  |  |
| --- | --- |
| Выполнил студент | Дьяченко Даниил Вадимович |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-621 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | доцент д.т.н. М.Г. Курносов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2018

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc532242751)

[1 Последовательный алгоритм 4](#_Toc532242752)

[2 Параллельный алгоритм 7](#_Toc532242754)

[3 Результаты эксперимента 9](#_Toc532242755)

[ЗАКЛЮЧЕНИЕ 12](#_Toc532242756)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 13](#_Toc532242757)

[ПРИЛОЖЕНИЕ 14](#_Toc532242758)

ВВЕДЕНИЕ

Математические модели в виде графов широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов.

Далее рассмотрим реализацию параллельного алгоритма Флойда на MPI, задача которого состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

1. Последовательный алгоритм

Граф хранится в матрице смежности.

Пусть *G* есть граф , для которого набор вершин , задается множеством , а список дуг графа , определяется множеством . В общем случае дугам графа могут приписываться некоторые числовые характеристики (веса) (взвешенный граф). Пример взвешенного графа приведен на рис. 1.

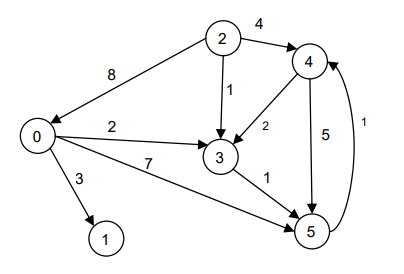


Рисунок 1 - Пример взвешенного ориентированного графа

Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. ), может быть эффективно обеспечено при помощи матрицы смежности , ненулевые значения элементов которой соответствуют дугам графа

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 1, приведена на рис. 2.

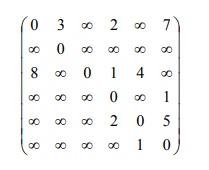


Рисунок 2 - Матрица смежности для графа из рис. 1

Исходной информацией для задачи поиска кратчайших путей является взвешенный граф , содержащий n вершин , в котором каждому ребру графа приписан неотрицательный вес. Граф будем полагать ориентированным, т.е., если из вершины i есть ребро в вершину j, то из этого не следует наличие ребра из j в i. В случае, если вершины все же соединены взаимообратными ребрами, то веса, приписанные им, могут не совпадать.

Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок . В общем виде данный алгоритм может быть представлен следующим образом:

// Serial Floyd algorithm

for (k = 0; k < n; k++)

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

A[i,j] = min(A[i,j], A[i,k]+A[k,j]);

(реализация операции выбора минимального значения min должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

1. Параллельный алгоритм

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значения. Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A.

Выполнение вычислений в подзадачах становится возможным только тогда, когда каждая подзадача содержит необходимые для расчетов элементы матрицы . Для исключения дублирования данных размещу в подзадаче единственный элемент , тогда получение всех остальных необходимых значений может быть обеспечено только при помощи передачи данных. Таким образом, каждый элемент строки матрицы должен быть передан всем подзадачам , а каждый элемент столбца матрицы должен быть передан всем подзадачам (см. рис. 3).

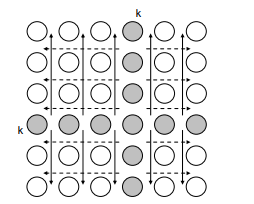
**

Рисунок 3 - Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач . Возможный способ укрупнения вычислений состоит в использовании ленточной схемы разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк.

В данной реализации матрица смежности заполнялась функцией dummy\_data, то есть фиктивными данными с полезной информацией, которая служит для возвращения формально верного результата.

В качестве бесконечности берется значение INT\_MAX (+2 147 483 647) из библиотеки limits.h.

Замер времени проводился функцией MPI\_Wtime. Замерялась вся работа программы, то есть время вычислений и пересылок.

1. Результаты эксперимента

Вычисления проводились на кластере Jet, оборудованным 18 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания. Узлы построены на базе серверной платформы Intel SR2520SAF. На каждом узле размещено два процессора Intel Quad Xeon E5420 с тактовой частотой 2.5 GHz. Пиковая производительность кластера – 1,44 TFLOPS.

Конфигурация вычислительного узла:

|  |  |
| --- | --- |
| Системная плата | Intel S5000VSA (Серверная платформа Intel SR2520SAF) |
| Процессор | 2 x Intel Xeon E5420 (2,5 GHz; Intel-64) |
| Оперативная память | 8 GB (4 x 2GB PC-5300) |
| Жесткий диск | SATAII 500GB (Seagate 500Gb Barracuda) |
| Сетевая карта | 2 x Intel Gigabit Ethernet (Integrated Intel PRO/1000 EB, 80003ES2LAN Gigabit Ethernet Controller) 1 x Intel PRO/1000 MT Server Adapter (PWLA8490MT, 82572EI Gigabit Ethernet Controller) |
| Корпус | Rack mount 2U |

Конфигурация коммуникационной среды:

|  |  |
| --- | --- |
| Сервисная сеть | Коммутатор Gigabit Ethernet (D-Link DGS-1224T) |
| Коммутатор | Fast Ethernet (3Com OfficeConnect, 8 ports) |
| Вычислительная сеть | Коммутатор Gigabit Ethernet (D-Link DGS-1224T) |

Все узлы работают операционной системой GNU/Linux Fedora 27 (ядро 4.18.16-100.fc27.x86\_64). В качестве системы распределения ресурсов используется TORQUE. Компилятор – mpicc for MPICH version 3.2.1, как обертка для gcc version 7.3.1 20180712 (Red Hat 7.3.1-6).

|  |  |  |
| --- | --- | --- |
| Количество узлов | Количество процессов на узел | Количество процессов всего |
| 1 | 1 | 1 |
| 1 | 2 | 2 |
| 1 | 4 | 4 |
| 1 | 8 | 8 |
| 2 | 8 | 16 |
| 4 | 8 | 32 |
| 10 | 4 | 40 |
| 13 | 4 | 52 |
| 8 | 8 | 64 |

Таблица 1 – конфигурация узлов и процессов.

Для эксперимента взяли следующие конфигурации вершин: 100, 400, 1600, 3000, 4000.

Далее приведены результаты экспериментов. Полученное ускорение есть отношение времени выполнения последовательного алгоритма (serial\_Floyd) к времени выполнения параллельного алгоритма (par\_Floyd).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество процессов | Количество вершин | | | | |
| 100 | 400 | 1600 | 3000 | 4000 |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1.4126 | 2.0353 | 2.1701 | 0.7 | 0.7412 |
| 4 | 1.6918 | 3.7435 | 4.4344 | 0.9613 | 0.9454 |
| 8 | 1.3410 | 6.1023 | 9.042 | 1.4589 | 1.4838 |
| 16 | 0.1865 | 4.5199 | 15.4624 | 15.7879 | 2.9042 |
| 32 | 0.0089 | 0.4278 | 23.6203 | 27.0325 | 19.0773 |
| 40 | 0.0484 | 2.1757 | 17.0141 | 23.7285 | 10.1377 |
| 52 | 0.0444 | 1.8898 | 12.8530 | 24.7172 | 10.2694 |
| 64 | 0.0158 | 1.2724 | 19.9229 | 33.1028 | 18.9263 |

Таблица 2 – ускорение, как результат, в отношении количества процессов к вершинам

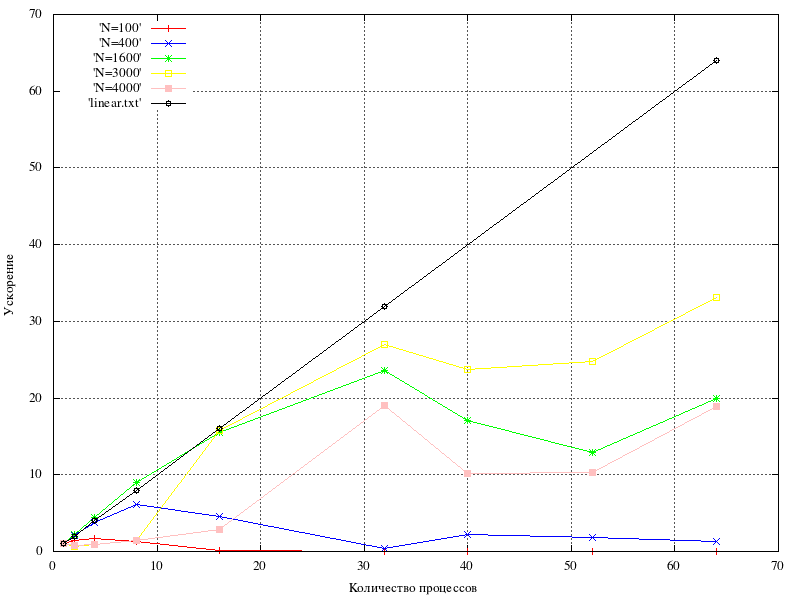


Рисунок 4 - график ускорения

Из результатов эксперимента видно, что при малом количестве вершин (100, 400) неэффективно использовать больше 8 процессов, так как время затрачиваемое на пересылку данных (функции MPI) превышает время вычислений, а при большом количестве вершин (1600, 3000, 4000) не следует брать меньше 16 процессов.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был реализован и исследован алгоритм Флойда для поиска кратчайших путей на графе между всеми парами вершин.

Осуществлено моделирование разработанного алгоритма. Показано, что данный алгоритм хорошо подлежит распараллеливанию, так как имеет участки, которые можно выполнять независимо, не требующий больших затрат на пересылку данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Википедия // Алгоритм Флойда-Уоршелла [электронный ресурс] <https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BB%D0%BE%D0%B9%D0%B4%D0%B0_%E2%80%94_%D0%A3%D0%BE%D1%80%D1%88%D0%B5%D0%BB%D0%BB%D0%B0> (дата обращения 2.12.2018)
2. Kvodo // Алгоритм Флойда-Уоршелла [электронный ресурс] <http://kvodo.ru/algoritm-floyda-uorshella.html> (дата обращения 2.12.2018)
3. Презентация професора Гергеля В.П. // Параллельные алгоритмы обработки графов [электронный ресурс] <http://www.hpcc.unn.ru/mskurs/LAB/RUS/PPT/Lab05.pdf> (дата обращения 2.12.2018)

ПРИЛОЖЕНИЕ

1. Исходный код

|  |
| --- |
| 1. /\* 2. \* main.c 3. \* 4. \*/ 5. #include <stdio.h> 6. #include <stdlib.h> 7. #include <mpi.h> 8. #include <limits.h> 9. #include <time.h> 10. #define INF\_PERC 30 11. static int rank; 12. static int commsize; 13. static double mpi\_total\_time; 14. int min(int a, int b) { 15. return (a < b) ? a : b; 16. } 17. void data\_random\_init(int \*arr, int n) { 18. srand(time(0)); 19. int rnd; 20. for (int i = 0; i < n; i++) { 21. for (int j = 0; j < n; j++) { 22. if (i != j) { 23. rnd = rand() % 1000; 24. if ((rnd % 100) < INF\_PERC) { 25. arr[i \* n + j] = INT\_MAX; 26. } else { 27. arr[i \* n + j] = rnd + 1; 28. } 29. } else { 30. arr[i \* n + j] = 0; 31. } 32. } 33. } 34. } 35. void dummy\_data\_init(int \*arr, int n) { 36. for (int i = 0; i < n; i++) { 37. for (int j = i; j < n; j++) { 38. if (i == j) { 39. arr[i \* n + j] = 0; 40. } else if (i == 0) { 41. arr[i \* n + j] = j; 42. } else { 43. arr[i \* n + j] = INT\_MAX; 44. } 45. arr[j \* n + i] = arr[i \* n + j]; 46. } 47. } 48. } 49. void serial\_Floyd(int \*arr, int n) { 50. for (int k = 0; k < n; k++) { 51. for (int i = 0; i < n; i++) { 52. for (int j = 0; j < n; j++) { 53. if ((arr[i \* n + k] != INT\_MAX) && (arr[k \* n + j] != INT\_MAX)) { 54. arr[i \* n + j] = min(arr[i \* n + j], arr[i \* n + k] + arr[k \* n + j]); 55. } 56. } 57. } 58. } 59. } 60. void print\_matrix(int \*arr, int n) { 61. for (int i = 0; i < n; i++) { 62. for (int j = 0; j < n; j++) { 63. if (arr[i \* n + j] == INT\_MAX) { 64. printf("INF\t"); 65. } else { 66. printf("%d\t", arr[i \* n + j]); 67. } 68. } 69. printf("\n"); 70. } 71. } 72. void par\_print\_matrix(int \*arr, int n, int count\_rows, int rank, int commsize) { 73. for (int p = 0; p < commsize; p++) { 74. if (p == rank) { 75. printf("rank = %d\n", rank); 76. for (int i = 0; i < count\_rows; i++) { 77. for (int j = 0; j < n; j++) { 78. if (arr[i \* n + j] == INT\_MAX) { 79. printf("INF\t"); 80. } else { 81. printf("%d\t", arr[i \* n + j]); 82. } 83. } 84. printf("\n"); 85. } 86. } 87. MPI\_Barrier(MPI\_COMM\_WORLD); 88. } 89. } 90. void copy(int \*start, int n, int \*out) { 91. if (!start || !out) { 92. return; 93. } 94. for (int i = 0; i < n; i++) { 95. out[i] = start[i]; 96. } 97. } 98. void row\_distr(int \*arr, int n, int k, int \*row) { 99. int num[commsize]; 100. int ind[commsize]; 101. int rem = n % commsize; 102. int count\_rows = n / commsize; 103. for (int i = 0; i < commsize; i++) { 104. num[i] = count\_rows; 105. if (rem > 0) { 106. num[i]++; 107. rem--; 108. } 109. ind[i] = (i > 0) ? ind[i - 1] + num[i - 1] : 0; 110. #if 0 111. if (rank == 0) { 112. printf("k = %d [%d] num[%d] = %d\t", k, rank, i, num[i]); 113. printf("ind[%d] = %d\n", i, ind[i]); 114. } 115. #endif 116. } 117. int row\_rank = -1; 118. for (int i = 0; i < commsize; i++) { 119. if (k < ind[i] + num[i]) { 120. row\_rank = i; 121. break; 122. } 123. } 124. #if 0 125. printf("[%d] k = %d row\_rank = %d\n", rank, k, row\_rank); 126. #endif 127. if (row\_rank == rank) { 128. copy(&arr[(k - ind[rank]) \* n], n, row); 129. } 130. mpi\_total\_time -= MPI\_Wtime(); 131. MPI\_Bcast(row, n, MPI\_INT, row\_rank, MPI\_COMM\_WORLD); 132. mpi\_total\_time += MPI\_Wtime(); 133. } 134. void par\_Floyd(int \*arr, int n, int count\_rows) { 135. int \*row = calloc(n, sizeof(int)); 136. if (!row) { 137. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 138. } 139. for (int k = 0; k < n; k++) { 140. row\_distr(arr, n, k, row); 141. for (int i = 0; i < count\_rows; i++) { 142. for (int j = 0; j < n; j++) { 143. if ((arr[i \* n + k] != INT\_MAX) && (row[j] != INT\_MAX)) { 144. arr[i \* n + j] = min(arr[i \* n + j], arr[i \* n + k] + row[j]); 145. } 146. } 147. } 148. } 149. free(row); 150. } 151. int compare(int \*a, int \*b, int n) { 152. for (int i = 0; i < n; i++) { 153. for (int j = 0; j < n; j++) { 154. if (a[i \* n + j] != b[i \* n + j]) { 155. return -1; 156. } 157. } 158. } 159. return 0; 160. } 161. int main(int argc, char \*\*argv) { 162. MPI\_Init(&argc, &argv); 163. mpi\_total\_time = 0; 164. double t = 0; 165. t -= MPI\_Wtime(); 167. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); 168. MPI\_Comm\_size(MPI\_COMM\_WORLD, &commsize); 169. int n; 170. int rem; 171. int \*arr; 172. // int \*cp\_arr; 173. int \*recv\_arr; 174. int count\_rows; 175. int real\_count\_rows; 176. if (rank == 0) { 177. n = (argc > 1) ? atoi(argv[1]) : 0; 178. if (n == 0) { 179. printf("How to run:\nmpiexec ./main <number of vertices>\n"); 180. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 181. } else if (n < commsize) { 182. printf("Need number of vertices bigger then number of processors\n"); 183. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 184. } 185. arr = calloc(n \* n, sizeof(int)); 186. if (!arr) { 187. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 188. } 189. // data\_random\_init(arr, n); 190. dummy\_data\_init(arr, n); 191. #if 0 192. printf("arr:\n"); 193. print\_matrix(arr, n); 194. #endif 195. #if 0 196. cp\_arr = calloc(n \* n, sizeof(int)); 197. if (!cp\_arr) { 198. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 199. } 200. copy(arr, n \* n, cp\_arr); 201. // printf("\nbefore floyd cp\_arr:\n"); 202. // print\_matrix(cp\_arr, n); 203. serial\_Floyd(cp\_arr, n); 204. // printf("\nafter floyd cp\_arr:\n"); 205. // print\_matrix(cp\_arr, n); 206. #endif 207. } 208. mpi\_total\_time -= MPI\_Wtime(); 209. MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); 210. mpi\_total\_time += MPI\_Wtime(); 211. real\_count\_rows = n / commsize; 212. rem = n % commsize; 214. if (rem > rank) { 215. real\_count\_rows++; 216. } 217. count\_rows = real\_count\_rows; 218. #if 0 219. printf("[%d] count\_rows = %d\n", rank, count\_rows); 220. #endif 221. recv\_arr = calloc(n \* count\_rows, sizeof(int)); 222. if (!recv\_arr) { 223. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 224. } 225. int \*send\_num = calloc(commsize, sizeof(int)); 226. int \*send\_ind = calloc(commsize, sizeof(int)); 227. if (!send\_num || !send\_ind) { 228. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 229. } 230. int old\_rem = rem; 231. #if 0 232. if (rank == 0) { 233. printf("rem = %d\n", rem); 234. } 235. #endif 236. count\_rows = n / commsize; 237. for (int i = 0; i < commsize; i++) { 238. send\_num[i] = count\_rows \* n; 239. if (rem > 0) { 240. send\_num[i] += n; 241. rem--; 242. } 243. send\_ind[i] = (i > 0) ? send\_ind[i - 1] + send\_num[i - 1] : 0; 244. #if 0 245. if (rank == 0) { 246. printf("rem = %d\n", rem); 247. } 248. #endif 249. } 250. rem = old\_rem; 251. count\_rows = real\_count\_rows; 253. mpi\_total\_time -= MPI\_Wtime(); 254. MPI\_Scatterv(arr, send\_num, send\_ind, MPI\_INT, recv\_arr, send\_num[rank], MPI\_INT, 0, MPI\_COMM\_WORLD); 255. mpi\_total\_time += MPI\_Wtime(); 256. #if 0 257. if (rank == 0) { 258. printf("\nprint before par\_Floyd\n"); 259. } 260. par\_print\_matrix(recv\_arr, n, count\_rows, rank, commsize); 261. printf("\n"); 262. #endif 263. count\_rows = real\_count\_rows; 265. par\_Floyd(recv\_arr, n, count\_rows); 267. #if 0 268. if (rank == 0) { 269. printf("\nprint after par\_Floyd\n"); 270. } 271. MPI\_Barrier(MPI\_COMM\_WORLD); 272. par\_print\_matrix(recv\_arr, n, count\_rows, rank, commsize); 273. #endif 274. int \*recv\_num = calloc(commsize, sizeof(int)); 275. int \*recv\_ind = calloc(commsize, sizeof(int)); 276. if (!recv\_num || !recv\_ind) { 277. MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); 278. } 279. #if 0 280. if (rank == 0) { 281. printf("rem = %d\n", rem); 282. } 283. #endif 284. count\_rows = n / commsize; 285. for (int i = 0; i < commsize; i++) { 286. recv\_num[i] = count\_rows \* n; 287. if (rem > 0) { 288. recv\_num[i] += n; 289. rem--; 290. } 291. recv\_ind[i] = (i > 0) ? recv\_ind[i - 1] + recv\_num[i - 1] : 0; 292. #if 0 293. if (rank == 0) { 294. printf("rem = %d\n", rem); 295. } 296. #endif 297. } 298. count\_rows = real\_count\_rows; 299. #if 0 300. if (rank == 0) { 301. for (int i = 0; i < commsize; i++) { 302. printf("recv\_ind[%d] = %d\n", i, recv\_ind[i]); 303. printf("recv\_num[%d] = %d\n", i, recv\_num[i]); 304. } 305. } 306. #endif 307. mpi\_total\_time -= MPI\_Wtime(); 308. MPI\_Gatherv(recv\_arr, real\_count\_rows, MPI\_INT, arr, recv\_num, recv\_ind, MPI\_INT, 0, MPI\_COMM\_WORLD); 309. mpi\_total\_time += MPI\_Wtime(); 310. #if 0 311. if (rank == 0) { 312. printf("\nprint ser after gather\n"); 313. print\_matrix(arr, n); 314. } 315. #endif 316. #if 0 317. if (rank == 0) { 318. if (compare(arr, cp\_arr, n) == 0) { 319. printf("Compare is bad\n"); 320. } else { 321. printf("Compare is good\n"); 322. // printf("Elapsed time is %.5f sec\n", t); 323. } 324. } 325. #endif 326. t += MPI\_Wtime(); 327. double total\_time\_max = 0; 328. MPI\_Reduce(&t, &total\_time\_max, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD); 329. double total\_wtime\_max = 0; 330. MPI\_Reduce(&mpi\_total\_time, &total\_wtime\_max, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD); 331. #if 1 332. if (rank == 0) { 333. printf("Total time is %.5f sec\n", total\_time\_max); 334. printf("Total mpi time is %.5f sec\n", total\_wtime\_max); 335. printf("Total compute time is %.5f sec\n", total\_time\_max - total\_wtime\_max); 336. printf("coefficient is %.4f \n", (double)total\_wtime\_max / (double)(total\_time\_max - total\_wtime\_max)); 337. } 338. #endif 339. MPI\_Finalize(); 340. return 0; 341. } |