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](#_Toc532150679)

[1 Постановка задачи 4](#_Toc532150680)

[2 Способ решения 6](#_Toc532150681)

[3 Конфигурации оборудования 7](#_Toc532150682)

[4 Результаты 8](#_Toc532150683)

[ЗАКЛЮЧЕНИЕ 12](#_Toc532150684)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 13](#_Toc532150685)

[ПРИЛОЖЕНИЕ 14](#_Toc532150686)

[1. Исходный код 14](#_Toc532150687)

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 рассмотрю способ параллельной реализации на MPI алгоритма Флойда (Floyd) на графе на примере задачи поиска кратчайших путей между всеми парами пунктов назначения. 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.

1 Постановка задачи

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.

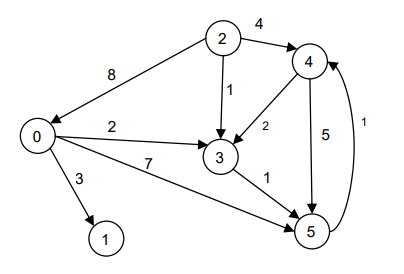


Рисунок 1 - Пример взвешенного ориентированного графа

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.

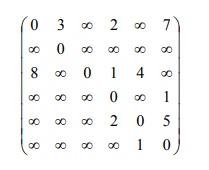


Рисунок 2 - Матрица смежности для графа из рис. 1

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.

1. Способ решения

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.

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).

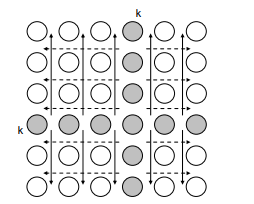


Рисунок Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

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.

1. Конфигурации оборудования

|  |  |  |
| --- | --- | --- |
| 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 |
| 8 | 8 | 64 |

Таблица 1 – конфигурация узлов и процессов.

1. Результаты

Для эксперимента были выбраны следующие количества вершин: 100, 400, 1600, 6400.

Далее приведены результаты экспериментов. Полученное ускорение есть отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма.

|  |  |
| --- | --- |
| Количество процессов | Ускорение |
| 1 | 1 |
| 2 | 1.526570048 |
| 4 | 1.858823529 |
| 8 | 1.915151515 |
| 16 | 1.389752205 |
| 32 | 1.271714066 |
| 64 | 0.717009751 |

Таблица 2 – результат эксперимента при 100 вершинах графа

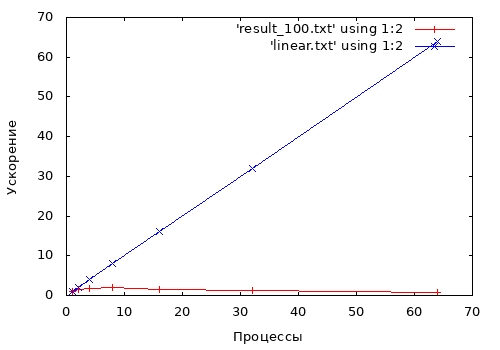


Рисунок 4 - результат эксперимента при 100 вершинах графа

|  |  |
| --- | --- |
| Количество процессов | Ускорение |
| 1 | 1 |
| 2 | 1.905673929 |
| 4 | 3.497363796 |
| 8 | 5.289036545 |
| 16 | 4.615830676 |
| 32 | 1.509591338 |
| 64 | 1.450088596 |

Таблица 3 – результат эксперимента при 400 вершинах графа

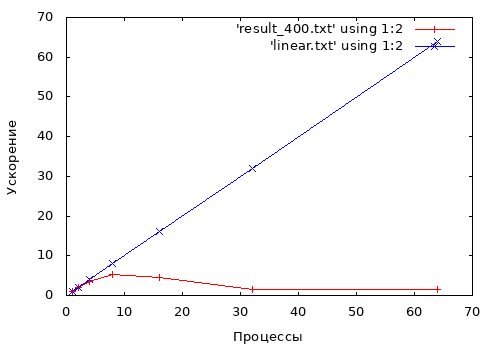


Рисунок 5- результат эксперимента при 400 вершинах графа

|  |  |
| --- | --- |
| Количество процессов | Ускорение |
| 1 | 1 |
| 2 | 2.069957071 |
| 4 | 4.100460951 |
| 8 | 8.029468621 |
| 16 | 12.756780286 |
| 32 | 16.333410741 |
| 64 | 18.825259176 |

Таблица 4 – результат эксперимента при 1600 вершинах графа

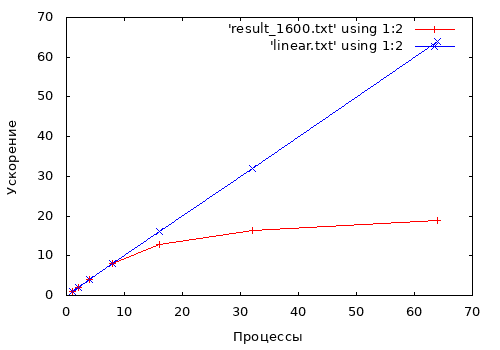


Рисунок 6 - результат эксперимента при 1600 вершинах графа

|  |  |
| --- | --- |
| Количество процессов | Ускорение |
| 1 | 1 |
| 2 | 1.394333926 |
| 4 | 1.115419639 |
| 8 | 2.114706752 |
| 16 | 4.089244011 |
| 32 | 7.319424121 |
| 64 | 46.772870629 |

Таблица 5 – результат эксперимента при 6400 вершинах графа

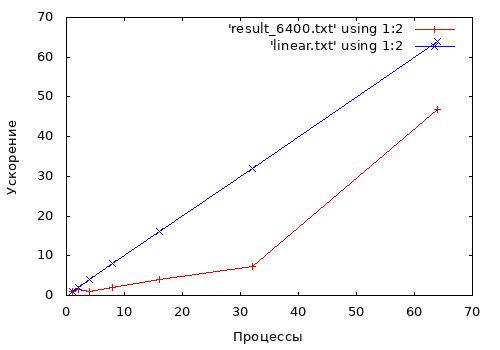


Рисунок 7 - результат эксперимента при 6400 вершинах графа

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработан 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. Хорошевский В.Г. Архитектура вычислительных систем. – М.: МГТУ им. Н.Э. Баумана,  
   2008. – 520 с.
2. Евреинов Э.В., Хорошевский В.Г. Однородные вычислительные системы. – Новосибирск: Наука, 1978. – 320 с.
3. Rabenseifner R.. Automatic MPI Counter Profiling // Proceedings of the 42nd Cray User Group. – Noorwijk, The Netherlands, 2000. – 19 pp.
4. Han D., Jones T.. MPI Profiling // Technical Report UCRL-MI-209658 – Lawrence Livermore National Laboratory, USA, 2004. – 15 pp.
5. Thakur R., Rabenseifner R., and Gropp W. Optimization of collective communication operations in MPICH // Int. Journal of High Performance Computing Applications. – 2005. – Vol. 19, No. 1. – P. 49‑66.
6. Balaji P., Buntinas D., Goodell D., Gropp W., Kumar S., Lusk E., Thakur R. and Traff J. L. MPI on a Million Processors // Proc. of the PVM/MPI – Berlin: Springer-Verlag, 2009. – P. 20‑30.
7. Khoroshevsky V., Kurnosov M. Mapping Parallel Programs into Hierarchical Distributed Computer Systems // Proc. of “Software and Data Technologies”.  Sofia: INSTICC, 2009.  Vol. 2.  P. 123‑128.

ПРИЛОЖЕНИЕ

* 1. Исходный код

|  |
| --- |
| /\*  \* main.c  \*  \*/  #include <stdio.h>  #include <stdlib.h>  #include <mpi.h>  #include <limits.h>  #include <time.h>  #define INF\_PERC 30  static int rank;  static int commsize;  int min(int a, int b) {  return (a < b) ? a : b;  }  int Min(int A, int B) {  int Result = (A < B) ? A : B;  if((A < 0) && (B >= 0)) Result = B;  if((B < 0) && (A >= 0)) Result = A;  if((A < 0) && (B < 0)) Result = -1;  return Result;  }  void data\_random\_init(int \*arr, int n) {  srand(time(0));  int rnd;  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (i != j) {  rnd = rand() % 1000;  if ((rnd % 100) < INF\_PERC) {  arr[i \* n + j] = INT\_MAX;  } else {  arr[i \* n + j] = rnd + 1;  }  } else {  arr[i \* n + j] = 0;  }  }  }  }  void dummy\_data\_init(int \*arr, int n) {  for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  if (i == j) {  arr[i \* n + j] = 0;  } else if (i == 0) {  arr[i \* n + j] = j;  } else {  arr[i \* n + j] = INT\_MAX;  }  arr[j \* n + i] = arr[i \* n + j];  }  }  }  void serial\_Floyd(int \*arr, int n) {  for (int k = 0; k < n; k++) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if ((arr[i \* n + k] != INT\_MAX) && (arr[k \* n + j] != INT\_MAX)) {  arr[i \* n + j] = min(arr[i \* n + j], arr[i \* n + k] + arr[k \* n + j]);  }  }  }  }  }  void print\_matrix(int \*arr, int n) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (arr[i \* n + j] == INT\_MAX) {  printf("INF\t");  } else {  printf("%d\t", arr[i \* n + j]);  }  }  printf("\n");  }  }  void par\_print\_matrix(int \*arr, int n, int count\_rows, int rank, int commsize) {  for (int p = 0; p < commsize; p++) {  if (p == rank) {  printf("rank = %d\n", rank);  for (int i = 0; i < count\_rows; i++) {  for (int j = 0; j < n; j++) {  if (arr[i \* n + j] == INT\_MAX) {  printf("INF\t");  } else {  printf("%d\t", arr[i \* n + j]);  }  }  printf("\n");  }  }  MPI\_Barrier(MPI\_COMM\_WORLD);  }  }  void copy(int \*start, int n, int \*out) {  if (!start || !out) {  return;  }  for (int i = 0; i < n; i++) {  out[i] = start[i];  }  }  void row\_distr(int \*arr, int n, int k, int \*row) {  int num[commsize];  int ind[commsize];  int rem = n % commsize;  int count\_rows = n / commsize;  for (int i = 0; i < commsize; i++) {  num[i] = count\_rows;  if (rem > 0) {  num[i]++;  rem--;  }  ind[i] = (i > 0) ? ind[i - 1] + num[i - 1] : 0;  #if 0  if (rank == 0) {  printf("k = %d [%d] num[%d] = %d\t", k, rank, i, num[i]);  printf("ind[%d] = %d\n", i, ind[i]);  }  #endif  }  int row\_rank = -1;  for (int i = 0; i < commsize; i++) {  if (k < ind[i] + num[i]) {  row\_rank = i;  break;  }  }  #if 0  printf("[%d] k = %d row\_rank = %d\n", rank, k, row\_rank);  #endif  if (row\_rank == rank) {  copy(&arr[(k - ind[rank]) \* n], n, row);  }  MPI\_Bcast(row, n, MPI\_INT, row\_rank, MPI\_COMM\_WORLD);  }  void par\_Floyd(int \*arr, int n, int count\_rows) {  int \*row = calloc(n, sizeof(int));  if (!row) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  for (int k = 0; k < n; k++) {  row\_distr(arr, n, k, row);  #if 0  for (int p = 0; p < commsize; p++) {  if (p == rank) {  printf("rank %d = ", rank);  for (int i = 0; i < n; i++) {  if (row[i] == INT\_MAX) {  printf("INF\t");  } else {  printf("%d\t", row[i]);  }  }  printf("\n");  }  MPI\_Barrier(MPI\_COMM\_WORLD);  }  #endif  for (int i = 0; i < count\_rows; i++) {  for (int j = 0; j < n; j++) {  if ((arr[i \* n + k] != INT\_MAX) && (row[j] != INT\_MAX)) {  arr[i \* n + j] = min(arr[i \* n + j], arr[i \* n + k] + row[j]);  }  }  }  }  free(row);  }  int compare(int \*a, int \*b, int n) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  if (a[i \* n + j] != b[i \* n + j]) {  return -1;  }  }  }  return 0;  }  int main(int argc, char \*\*argv) {  MPI\_Init(&argc, &argv);  double t = 0;  t -= MPI\_Wtime();    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &commsize);  int n;  int rem;  int \*arr;  // int \*cp\_arr;  int \*recv\_arr;  int count\_rows;  int real\_count\_rows;  if (rank == 0) {  n = (argc > 1) ? atoi(argv[1]) : 0;  if (n == 0) {  printf("How to run:\nmpiexec ./main <number of vertices>\n");  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  } else if (n < commsize) {  printf("Need number of vertices bigger then number of processors\n");  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  arr = calloc(n \* n, sizeof(int));  if (!arr) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  data\_random\_init(arr, n);  // dummy\_data\_init(arr, n);  #if 0  printf("arr:\n");  print\_matrix(arr, n);  #endif  #if 0  cp\_arr = calloc(n \* n, sizeof(int));  if (!cp\_arr) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  copy(arr, n \* n, cp\_arr);  // printf("\nbefore floyd cp\_arr:\n");  // print\_matrix(cp\_arr, n);  serial\_Floyd(cp\_arr, n);  // printf("\nafter floyd cp\_arr:\n");  // print\_matrix(cp\_arr, n);  #endif  }  MPI\_Barrier(MPI\_COMM\_WORLD);  MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  real\_count\_rows = n / commsize;  rem = n % commsize;    if (rem > rank) {  real\_count\_rows++;  }  count\_rows = real\_count\_rows;  #if 0  printf("[%d] count\_rows = %d\n", rank, count\_rows);  #endif  recv\_arr = calloc(n \* count\_rows, sizeof(int));  if (!recv\_arr) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  int \*send\_num = calloc(commsize, sizeof(int));  int \*send\_ind = calloc(commsize, sizeof(int));  if (!send\_num || !send\_ind) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  int old\_rem = rem;  #if 0  if (rank == 0) {  printf("rem = %d\n", rem);  }  #endif  count\_rows = n / commsize;  for (int i = 0; i < commsize; i++) {  send\_num[i] = count\_rows \* n;  if (rem > 0) {  send\_num[i] += n;  rem--;  }  send\_ind[i] = (i > 0) ? send\_ind[i - 1] + send\_num[i - 1] : 0;  #if 0  if (rank == 0) {  printf("rem = %d\n", rem);  }  #endif  }  rem = old\_rem;  count\_rows = real\_count\_rows;  #if 0  if (rank == 0) {  for (int i = 0; i < commsize; i++) {  printf("send\_num[%d] = %d\n", i, send\_num[i]);  printf("send\_ind[%d] = %d\n", i, send\_ind[i]);  }  }  #endif    MPI\_Scatterv(arr, send\_num, send\_ind, MPI\_INT, recv\_arr, send\_num[rank], MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Barrier(MPI\_COMM\_WORLD);  #if 0  if (rank == 0) {  printf("\nprint before par\_Floyd\n");  }  par\_print\_matrix(recv\_arr, n, count\_rows, rank, commsize);  printf("\n");  #endif  count\_rows = real\_count\_rows;  MPI\_Barrier(MPI\_COMM\_WORLD);  par\_Floyd(recv\_arr, n, count\_rows);  MPI\_Barrier(MPI\_COMM\_WORLD);  #if 0  printf("[%d] FLOYD OK\n", rank);  #endif  #if 0  if (rank == 0) {  printf("\nprint after par\_Floyd\n");  }  MPI\_Barrier(MPI\_COMM\_WORLD);  par\_print\_matrix(recv\_arr, n, count\_rows, rank, commsize);  #endif  int \*recv\_num = calloc(commsize, sizeof(int));  int \*recv\_ind = calloc(commsize, sizeof(int));  if (!recv\_num || !recv\_ind) {  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  #if 0  if (rank == 0) {  printf("rem = %d\n", rem);  }  #endif  count\_rows = n / commsize;  for (int i = 0; i < commsize; i++) {  recv\_num[i] = count\_rows \* n;  if (rem > 0) {  recv\_num[i] += n;  rem--;  }  recv\_ind[i] = (i > 0) ? recv\_ind[i - 1] + recv\_num[i - 1] : 0;  #if 0  if (rank == 0) {  printf("rem = %d\n", rem);  }  #endif  }  count\_rows = real\_count\_rows;  #if 0  if (rank == 0) {  for (int i = 0; i < commsize; i++) {  printf("recv\_ind[%d] = %d\n", i, recv\_ind[i]);  printf("recv\_num[%d] = %d\n", i, recv\_num[i]);  }  }  #endif  #if 0  if (rank == 0) {  MPI\_Gatherv(MPI\_IN\_PLACE, real\_count\_rows, MPI\_INT, arr, recv\_num, recv\_index, MPI\_INT, 0, MPI\_COMM\_WORLD);  } else {  // MPI\_Gatherv(recv\_arr, recv\_num[rank], MPI\_INT, arr, recv\_num, recv\_index, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Gatherv(recv\_arr, real\_count\_rows, MPI\_INT, NULL, NULL, NULL, MPI\_INT, 0, MPI\_COMM\_WORLD);  }  #endif  if (rank == 0) {  for (int i = 0; i < count\_rows; i++) {  for (int j = 0; j < n; j++) {  arr[i \* n + j] = recv\_arr[i \* n + j];  }  }  MPI\_Status status;  for (int i = 1; i < commsize; i++) {  MPI\_Recv(arr + recv\_ind[i], recv\_num[i], MPI\_INT, i, 0, MPI\_COMM\_WORLD, &status);  }  } else {  MPI\_Send(recv\_arr, count\_rows \* n, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);  }  MPI\_Barrier(MPI\_COMM\_WORLD);  #if 0  if (rank == 0) {  printf("\nprint ser after gather\n");  print\_matrix(arr, n);  }  #endif  t += MPI\_Wtime();  #if 0  if (rank == 0) {  if (compare(arr, cp\_arr, n)) {  printf("Compare is bad\n");  } else {  printf("Compare is good\n");  printf("Elapsed time is %.5f sec\n", t);  }  }  #endif  if (rank == 0) {  printf("Elapsed time is %.5f sec\n", t);  }  MPI\_Finalize();  return 0;  } |