МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Параллельные алгоритмы».
Тема: Виртуальные топологии.

Студентка гр. 0382 _____ Кривенцова Л.С.

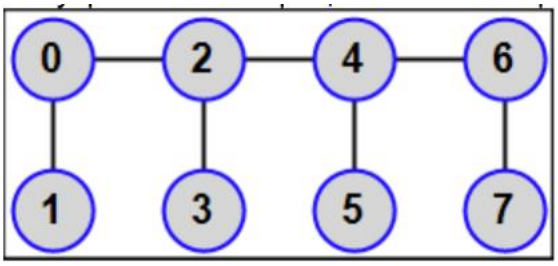
Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2022

Задание.

Вариант 12.

Число процессов K является четным: K = 2N (1 < N < 6); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой все процессы четного ранга (включая главный процесс) связаны в цепочку: 0 — 2 — 4 — 6 — … — (2N − 2), а каждый процесс нечетного ранга R (1, 3, …, 2N − 1) связан с процессом ранга R − 1 (в результате каждый процесс нечетного ранга будет иметь единственного соседа, первый и последний процессы четного ранга будут иметь по два соседа, а остальные — «внутренние» — процессы четного ранга будут иметь по три соседа).



Переслать число A из каждого процесса всем процессам-соседям. Для определения количества процессов-соседей и их рангов использовать функции MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

Листинг программы см. в Приложении А.

Выполнение работы.

В начале программы инициализируются переменные количества процессов и номер текущего, выделяется память для массивов, необходимых для работы с функцией *MPI_Graph_create*.

Вызов конструктора универсальной (графовой) топологии производится следующим образом:

MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, false, &newcomm);

Где *MPI_COMM_WORLD* - входной коммуникатор, *size* - количество узлов графа, *index* - массив целочисленных значений, описывающий степени вершин (структура описана ниже), *edges* - массив целочисленных значений, описывающий ребра графа (структура описана ниже), *false* говорит о том, что номера не могут быть переупорядочены, *newcomm* – выходной коммуникатор с графовой топологией (который создаётся в начале программы).

Массив *index* заполняется до вызова функции следующим образом: i-ый элемент *index* хранит общее число соседей первых i вершин графа. Подсчёт ведётся так: для крайних чётных узлов (*0* и *size - 2*) количество рёбер равно двум (суммируется с предыдущим показателем), иначе, если узел четный – прибавляется 3, иначе (нечётный) 2.

Списки соседей вершин *0, 1, ..., size-1* хранятся в последовательности ячеек массива *edges*. Если узел *[i]* нечётный, то у него один сосед – узел *[i - 1]*. Иначе (для чётного узла), вычисляется сосед *[i + 1]*, а если *[i + 2]* и *[i - 2]* не выходят за рамки графа, то они тоже записываются в массив в качестве соседей *[i]*.

Далее в переменные *number_of_neighbours_retrieved* и *neighbours_retrieved* записываются количество процессов-соседей текущего и их ранги с помощью функций *MPI_Graph_neighbors_count* и *MPI_Graph_neighbors*. Далее запускается цикл *for*, в каждой итерации которого, с помощью вызова *MPI_Sendrecv* процесс обменивается числом A (передаёт своё значение числа, и принимает значение соседа) с очередным узлом-соседом, так как функция комбинирует в одном обращении и посылку, и прием сообщения от отправителя. Для наглядности сообщения число A равняется номеру процесса + 1.

MPI_Sendrecv(&a, 1, MPI_INT, neighbours_retrieved[n], 1,

&a_resv, 1, MPI_INT, neighbours_retrieved[n], 1, newcomm, &status);

Где *&a* и *&a_resv* – начальные адреса буфера отправления и приёма, 1 – длина сообщения, *MPI_INT* – тип передаваемого и получаемого сообщений, *neighbours_retrieved[n]* – номер процесса-отправителя и получателя, 1 – тэг для отправки и приёма сообщения, *newcomm* - коммуникатор, *&status* – статус.

Номер процесса и полученные им числа выводятся в консоль.

Результаты работы программы на 1,2 …. N процессах.

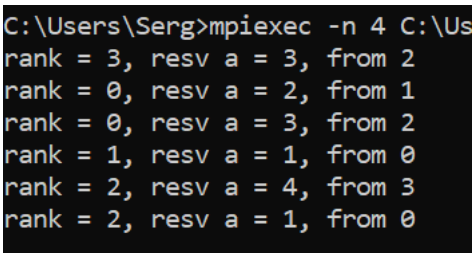


Рис. 1 - Результаты работы программы на 4 процессах.

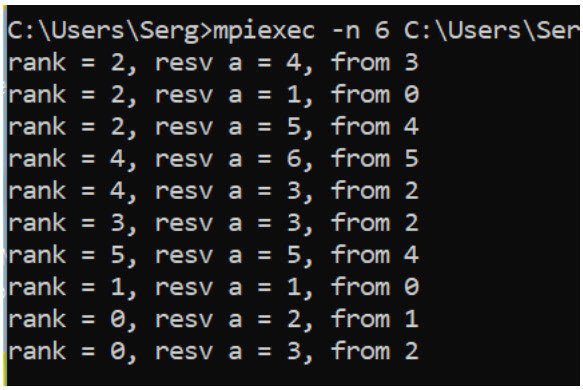


Рис. 2 - Результаты работы программы на 6 процессах.

График зависимости времени выполнения программы от числа процессов.

Так как длина сообщения фиксированная (одно число A), исследуем только зависимость времени от числа процессов.

Для получения экспериментальных данных модифицируем программу так, чтобы выводились также нужные для построения графика данные: время работы программы на каждом процессоре. Для этого добавим в программу следующие строчки:

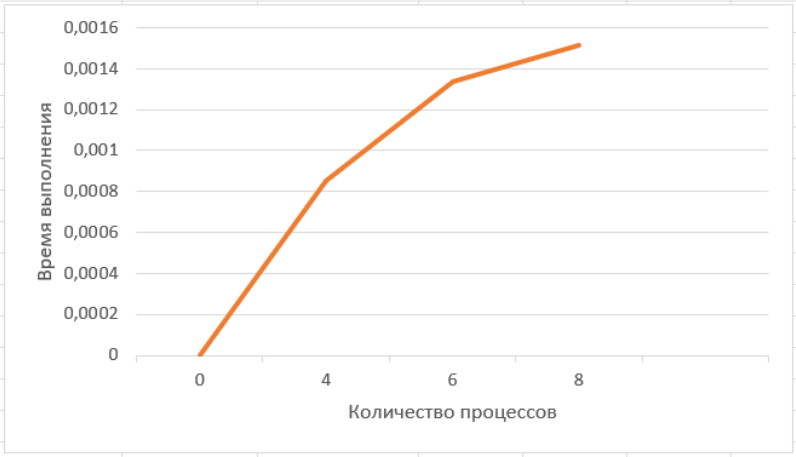
time_start = MPI_Wtime();

…

printf("\nTime: %f, rank %d\n", MPI_Wtime() - time_start, rank);

Для выявления зависимости получим значения времени каждого запуска программы с различным количеством процессов.

Число процессов	Время выполнения
0	0
4	0,000854667
6	0,001336333
8	0,001516125



Номер процесса	Время выполнения программы, с
0	0
Среднее	0
0	0,001352
1	0,001253
2	0,001307
3	0,001216
Среднее	0,000854667
0	0,001344
1	0,001316
2	0,001269
3	0,001472
4	0,001472
5	0,001145
Среднее	0,001336333
0	0,001668
1	0,001254
2	0,001792
3	0,001302
4	0,001748
5	0,001256
6	0,00182
7	0,001289
Среднее	0,001516125

Рис.3 – График зависимости времени выполнения программы от числа процессов (и данные для построения).

Следовательно, выполняемое время напрямую зависит от количества зависимых процессов: чем больше процессов, тем больше строимый граф: и дольше идёт работа с этим графом и пересылом данных от вершины к соседям. А чем больше процессов, тем больше работы для программы: повышается количество выполняемых действий (отправок и приёмов сообщения).

Сети Петри.

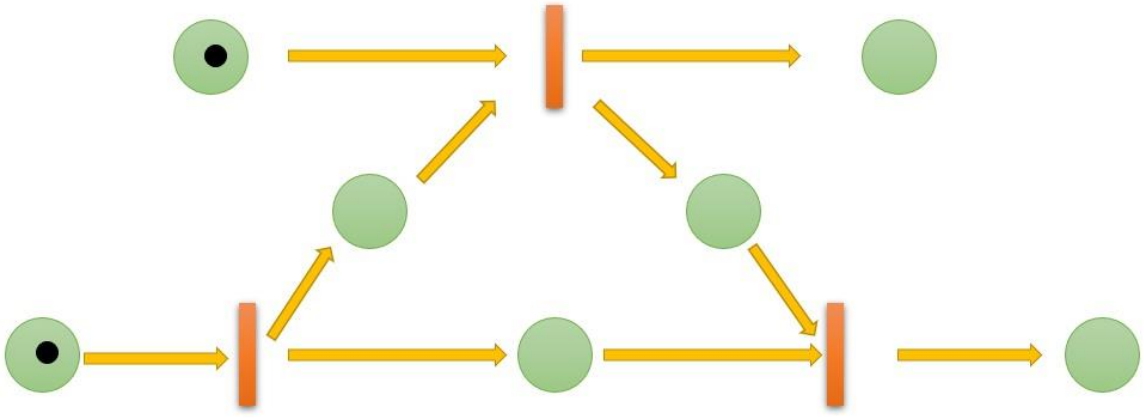


Рис.4 – Сети Петри, на которых отображен процесс пересыла сообщениями

между двумя процессами (вершинами-соседями), где с помощью MPI_Sendrecv процессы «обмениваются» сообщениями друг с другом. Аналогичная работа происходит между всеми соседними вершинами графа(процессами).

Выводы.

Написана программа, определяющая топологию заданного графа, осуществляющая отправку и приём сообщения между процессами в определенном порядке.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <stdio.h>
#include <mpi.h>
#include <cstdlib>
#include <ctime>
#define MAX_LENGTH 1000

int main(int argc, char* argv[])
{
  double time_start;
  int size, rank, n,a_resv, newrank, new_a;
  MPI_Status status;
  MPI_Comm newcomm;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int a = rank + 1;
  int* index = new int[size];
  int* edges = new int[size - 1];
  int count = 0;
  time_start = MPI_Wtime();
  for (int i = 0; i < size; i++) {
    if (i == 0) {
      index[0] = 2;
    }
    else if (i == size - 2) {
      index[count] = index[count - 1] + 2;
    }
    else if (i % 2 == 0) {
      index[count] = index[count - 1] + 3;
    }
    else {
      index[count] = index[count - 1] + 1;
    }
    count++;
  }

  count = 0;
  for (int i = 0; i < size; i++) {
    if (i % 2 != 0) {
      edges[count] = i - 1;
```

```
      count++;
    }
    else {
      edges[count] = i + 1;
      count++;
      if (i - 2 >= 0) {
        edges[count] = i - 2;
        count++;
      }
      if (i + 2 < size - 1) {
        edges[count] = i + 2;
        count++;
      }
    }
  }
  int number_of_neighbours_retrieved;
  MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, false, &newcomm);
  MPI_Graph_neighbors_count(newcomm, rank,
&number_of_neighbours_retrieved);
  int* neighbours_retrieved = new int[number_of_neighbours_retrieved];
  MPI_Graph_neighbors(newcomm, rank, number_of_neighbours_retrieved,
neighbours_retrieved);
  for (int n = 0; n < number_of_neighbours_retrieved; n++) {
    MPI_Sendrecv(&a, 1, MPI_INT, neighbours_retrieved[n], 1,
      &a_resv, 1, MPI_INT, neighbours_retrieved[n], 1, newcomm,
&status);
    printf("rank = %d, resv a = %d, from %d\n", rank, a_resv,
neighbours_retrieved[n]);
  }
  MPI_Finalize();
  return 0;
}
```