

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

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Параллельные алгоритмы».**  
**Тема: ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ-ДЖОКЕРОВ.**

Студентка гр. 0382

Кривенцова Л.С.

Преподаватель

Татаринов Ю.С.

Санкт-Петербург

2022

## **Задание.**

### **Вариант 4.**

Раздача и сборка массива. Процесс 0 генерирует целочисленный массив и раздает его по частям в остальные процессы; порядок раздачи определяется случайным образом, размер каждого следующего передаваемого фрагмента в 2 раза меньше предыдущего. Процессы-получатели выполняют обработку массива и возвращают результат в процесс 0. Процесс 0 должен собрать массив результатов обработки с сохранением последовательности элементов.

Листинг программы см. в Приложении А.

### **Выполнение работы.**

В программе осуществляется раздача и сборка массива со следующей логикой. Если проверкой выяснилось, что программа выполняется нулевым процессом, то формируется массив (*int\* random*), случайным образом определяющий порядок раздачи частей массива остальным процессам. Затем создаётся основной массив (каждый элемент принимает случайное значение от 1 до 100). Затем рассчитывается длина части массива, которую процесс 0 должен передать первому (по сформированному порядку, а не по значению) процессу таким образом, чтобы с условием передачи (порядок раздачи определяется случайным образом, размер каждого следующего передаваемого фрагмента в 2 раза меньше предыдущего) и текущим количеством процессов им всего было передано наибольшее количество элементов массива. Запускается цикл *for*, проходящийся по остальным процессам в порядке, заданном массивом *random*. В нём формируется массив *int\* message*, являющий собой часть массива, которую нужно передать данному процессу – копируется следующая часть массива вычисленной длины (изначально вычисляется по формуле, на каждой итерации цикла делится пополам). Собранное сообщение передается процессу как:  
*MPI\_Isend(message, MAX\_LENGTH, MPI\_INT, random[i], 0, MPI\_COMM\_WORLD, &req);*

Затем сообщением принимается ответ, содержащий ту же часть исходного массива, но отсортированную: *MPI\_Recv(message, MAX\_LENGTH, MPI\_INT,*

*random[i], 0, MPI\_COMM\_WORLD, &status);* На этом цикл завершается. Все полученные назад части массива по мере поступления записываются в массив *int\* result*, и затем выводятся на экран.

Если процесс не нулевой, так же объявляется массив *message*, в который принимается сообщение от нулевого процесса: *MPI\_Recv(message, MAX\_LENGTH, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);* После этого полученная часть массива сортируется «пузырьком» и передаётся обратно 0 процессу: *MPI\_Isend(message, MAX\_LENGTH, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &req);*

Для отправки сообщения используется стандартная функция *MPI\_Isend*, так как это неблокирующая отправка, позволяющая осуществить обмен сообщениями с заданными условиями так как вызов функции возвращает управление к программе и можно задействовать тем же процессом также функцию *MPI\_Recv*.

### Результаты работы программы на 1,2 .... N процессорах.

```
C:\Users\Serg>mpiexec -n 4 C:\Users\Serg\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe

process 2 Sorted part of the array
message[0] = 0
message[1] = 37
message[2] = 58
message[3] = 80
message[4] = 84

process 1 Sorted part of the array
message[0] = 71

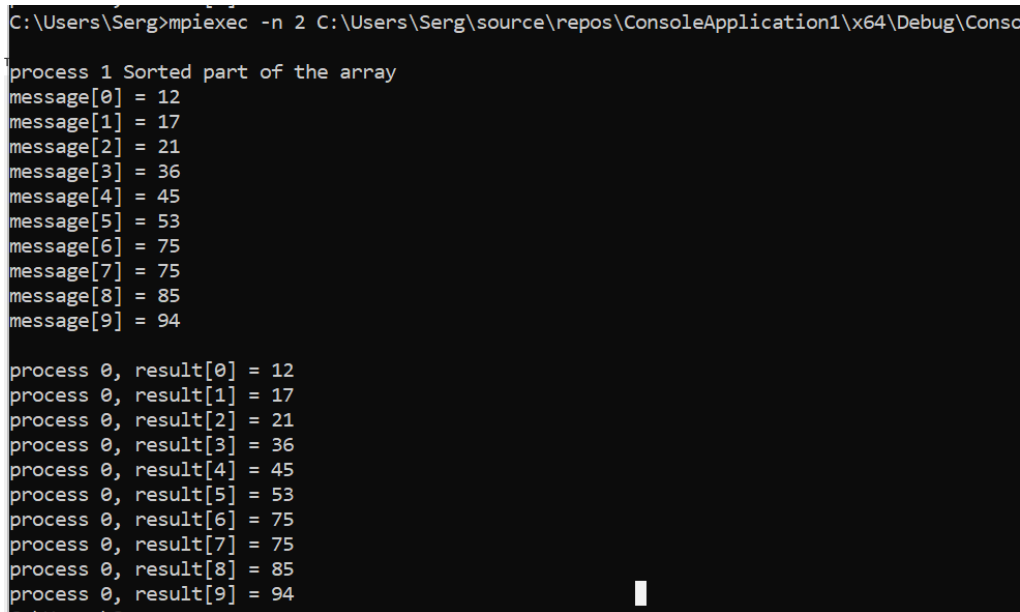
process 3 Sorted part of the array
message[0] = 38
message[1] = 68

process 0, result[0] = 0
process 0, result[1] = 37
process 0, result[2] = 58
process 0, result[3] = 80
process 0, result[4] = 84
process 0, result[5] = 38
process 0, result[6] = 68
process 0, result[7] = 71
C:\Users\Serg>
```

Рис. 1 - Результаты работы программы на 4 процессорах с длиной массива 10.

Из вывода данных видно, что сначала 0 процесс передает большую часть массива процессу 2, принимает её в отсортированном виде и записывает в

результатирующий массив. Затем то же происходит с процессами 3 и 1. В результате получаем массив отсортированный по частям (в результате склеивания получается не точно отсортированный массив) из 8 элементов. 2 элемента потеряны из соображений, что при 10 элементах и 3 работающих (кроме нулевого) процессах разделить массив по частям с заданным условием (размер каждого следующего передаваемого фрагмента в 2 раза меньше предыдущего) невозможно, и ближайший к 10 возможный вариант деления – это 8 элементов.



```
C:\Users\Serg>mpirun -n 2 C:\Users\Serg\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe

process 1 Sorted part of the array
message[0] = 12
message[1] = 17
message[2] = 21
message[3] = 36
message[4] = 45
message[5] = 53
message[6] = 75
message[7] = 75
message[8] = 85
message[9] = 94

process 0, result[0] = 12
process 0, result[1] = 17
process 0, result[2] = 21
process 0, result[3] = 36
process 0, result[4] = 45
process 0, result[5] = 53
process 0, result[6] = 75
process 0, result[7] = 75
process 0, result[8] = 85
process 0, result[9] = 94
```

Рис. 2 - Результаты работы программы на 2 процессорах, длина массива 10.

Так как принимающий часть массива для обработки процесс всего один, то он принимает и сортирует массив целиком.

При запуске на одном процессоре не происходит никакой отправки, так как в программе стоит ограничение на количество участвующих в обмене процессов. При запуске на минимальном возможном (2) отправка и приём сообщения осуществляется корректно.

**График зависимости времени выполнения программы от числа процессов для разных длин пересылаемых сообщений.**

Для получения экспериментальных данных модифицируем программу так, чтобы выводились только нужные для построения графика данные: время работы программы на каждом процессоре и длина обмениваемого сообщения.

Для выявления зависимости получим значения времени каждого запуска программы с различной длиной сообщения и получим данные:

Количество процессов	Время выполнения программы, с	Длина сообщения	
1	0,000001	1	
2	0,000262	1	
5	0,000797	1	
10	0,001537	1	
Количество процессов	Время выполнения программы, с	Длина сообщения	
1	0,000003	100	
2	0,000707	100	
5	0,001563	100	
10	0,002358	100	
Количество процессов	Время выполнения программы, с	Длина сообщения	
1	0,000027	1 000	
2	0,004408	1 000	
5	0,006318	1 000	
10	0,009909	1 000	

Рис.3 – Экспериментальные данные запусков программы на 1,2,5,10 процессах с различной длиной сообщения.

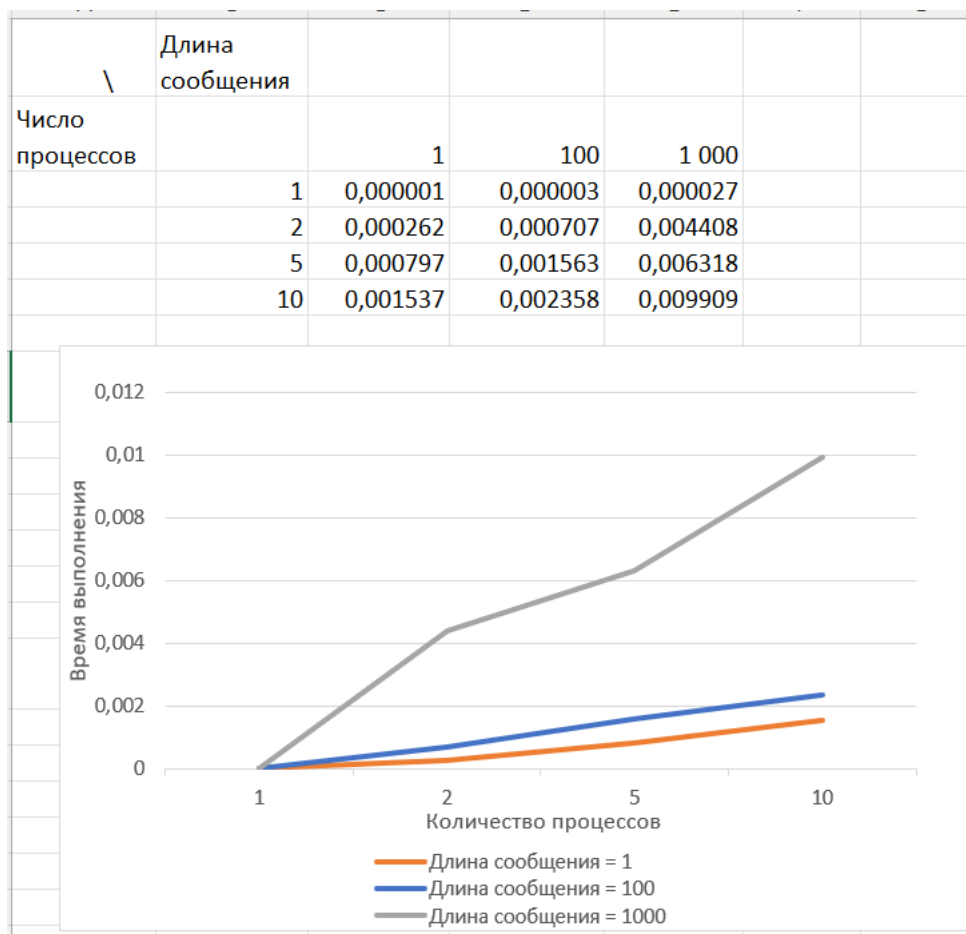


Рис.4 – Графики зависимости времени выполнения программы от числа процессов для разных длин пересылаемых сообщений (и данные для их построения).

Следовательно, выполняемое время напрямую зависит от количества зависимых процессов: чем больше процессов, тем дольше идёт работа с обменами сообщением. Это происходит, потому что чем больше процессов, тем больше работы для программы: повышается количество выполняемых действий (отправок и приёмов сообщения). Между длиной сообщения и временем выполнения так же прямо-пропорциональная связь, т.к. чем больше массив, тем больше итераций циклов при обходе массива и его частей.

### **Сети Петри.**

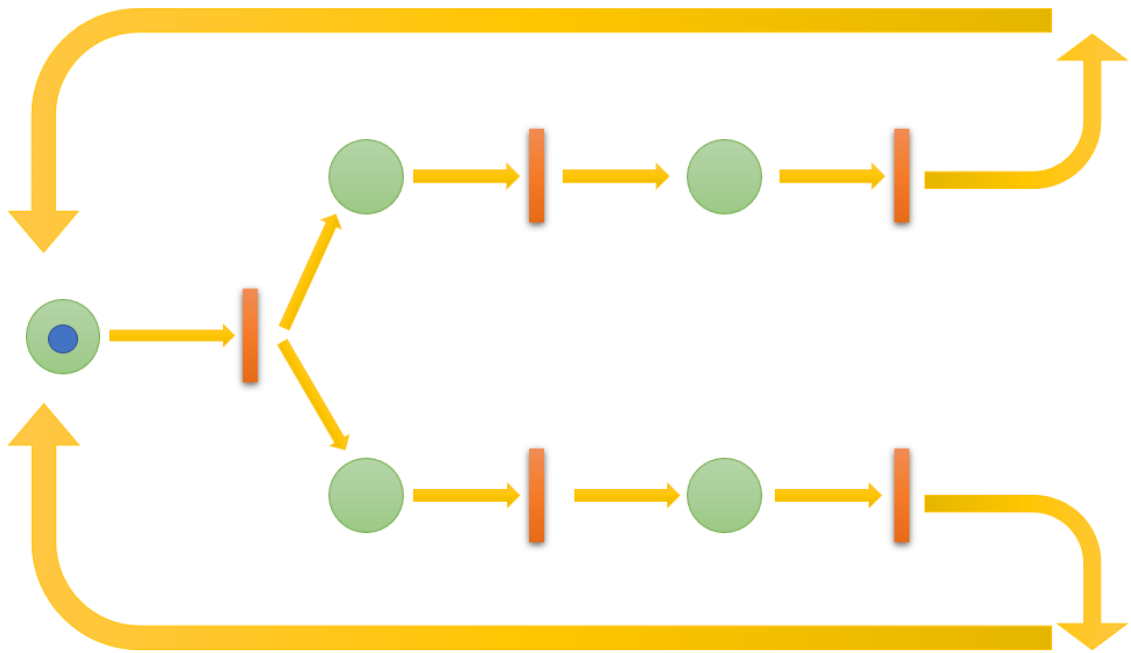


Рис.6 – Сети Петри.

**Выводы.**

Написана программа обмена сообщениями, осуществляющая раздачу массива по частям от процесса 0 к остальным и сборку исходного массива по обработанным переданным назад частям.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <stdio.h>
#include <mpi.h>
#include <cstdlib>
#include <ctime>
#define MAX_LENGTH 10

int main(int argc, char* argv[])
{
    srand(time(0));
    double time_start, time;
    int size, rank, count = 0, index = 0, index_res = 0;
    double double_x;
    MPI_Status status;
    int* array = new int[MAX_LENGTH];
    int* result = new int[MAX_LENGTH];
    int* message;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int* random = new int[size];

    MPI_Request req;
    time_start = MPI_Wtime();
    if (rank == 0 && rank < size) {
        for (int i = 1; i < size; ++i)
            random[i - 1] = i;
        for (int i = 0; i < size - 1; ++i)
        {
            int temp = random[i];
            int for_rand = rand() % (size - 1);
            random[i] = random[for_rand];
            random[for_rand] = temp;
        }
        for (int i = 0; i < MAX_LENGTH; i++) {
            array[i] = rand() % 100;
        }
        if (size <= 2) count = MAX_LENGTH;
        else {
            double summ = 0;
            for (int x = 1; x < size; x++) {
```



```

        summ += double(1)/x;
    }
    count = round(MAX_LENGTH / summ);
}
for (int i = 0; i < size - 1; ++i) {
    message = (int*)(calloc(MAX_LENGTH, sizeof(int)));
    for (int k = 0; k < MAX_LENGTH; k++) {
        if (k < count) message[k] = array[index++];
        else {
            if (count == 0 && index < MAX_LENGTH)
                message[k] = array[index++];
            else message[k] = -1;
        }
    }
    MPI_Isend(message, MAX_LENGTH, MPI_INT, random[i],
        0, MPI_COMM_WORLD, &req);
    MPI_Recv(message, MAX_LENGTH, MPI_INT, random[i], 0,
MPI_COMM_WORLD,
        &status);
    for (int t = 0; t < count && message[t] >= 0; t++) {
        result[index_res] = message[t];
        index_res++;
    }
    count = round(count / 2);
}
for (int res = index_res; res < MAX_LENGTH; res++) result[res] = -1;
for (int t = 0; t < MAX_LENGTH && result[t] >= 0; t++) {
    printf("\nprocess %d, result[%d] = %d", rank, t, result[t]);
}
}
else if (rank > 0) {
    message = new int[MAX_LENGTH];
    MPI_Recv(message, MAX_LENGTH, MPI_INT, 0, 0, MPI_COMM_WORLD,
        &status);
    int temp;
    for (int i = 0; i < MAX_LENGTH - 1; i++) {
        for (int j = 0; j < MAX_LENGTH - i - 1; j++) {
            if (message[j] > message[j + 1] && message[j] >= 0 && message[j
+ 1] >= 0) {
                temp = message[j];
                message[j] = message[j + 1];
                message[j + 1] = temp;
            }
        }
    }
}

```

```

    }
}

    printf("\nprocess %d Sorted part of the array", rank);
    for (int t = 0; t < MAX_LENGTH && message[t] >= 0; t++)
        printf("\nmessage[%d] = %d", t, message[t]);

    MPI_Isend(message, MAX_LENGTH, MPI_INT, 0,
              0, MPI_COMM_WORLD, &req);
}
MPI_Finalize();
return 0;
}

```