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

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Параллельные алгоритмы»
ТЕМА: Запуск параллельной программы на различном числе одновременно работающих процессов, упорядочение вывода результатов.

Студент Степаненко Д. В.
Преподаватель Татаринов Ю. С.

Санкт-Петербург
2023 г.

Цель

Ознакомиться с библиотекой MPI, написав параллельную программу с последующим анализом.

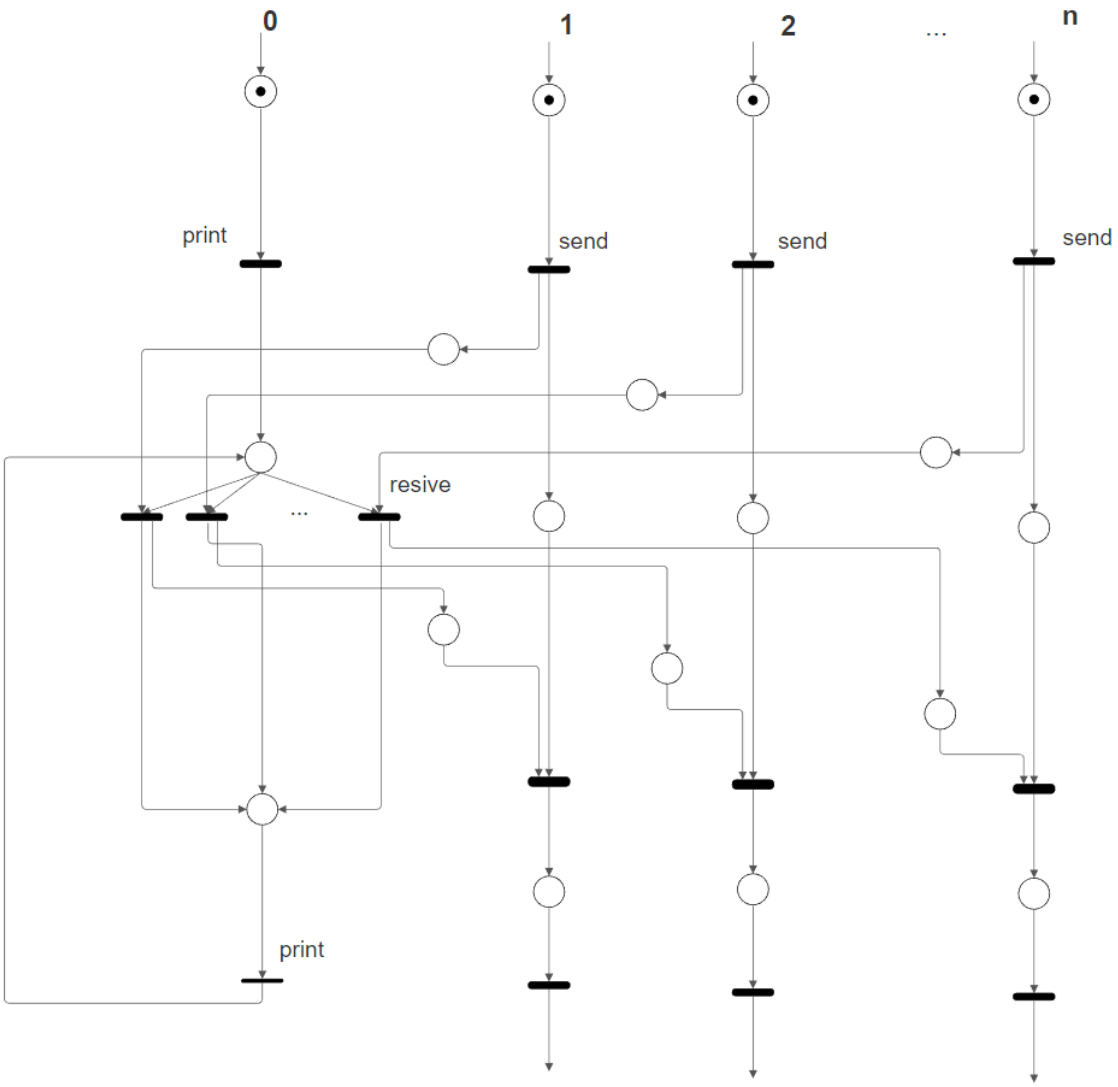
Постановка задачи

Запустить программу на 1,2 … N процессах несколько раз. Проанализировать порядок вывода сообщений на экран. Вывести правило, определяющее порядок вывода сообщений. Модифицировать программу таким образом, чтобы порядок вывода сообщений на экран соответствовал номеру соответствующего процесса.

Выполнение работы

Программа создает несколько процессов, считывает ранг каждого и общее количество процессов. Если процесс не 0 ранга, то он посылает сообщение со своим рангом, если ранг 0 – получает (количество получаемых сообщений = количество всех процессов – 1).

Сеть Петри основной части алгоритма:

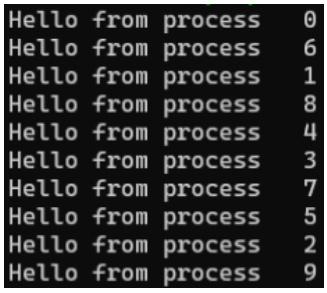


Листинг программы:

```
#include <time.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
        int ProcNum;
        int ProcRank;
        int RecvRank;
        MPI_Status Status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
        MPI_Comm_size(MPI_COMM_WORLD, & ProcNum);

        if (ProcRank != 0) {
                MPI_Send(&ProcRank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        else if(ProcRank == 0){
                printf("\n Hello from process %3d", ProcRank);
```

```
                for(int i=0; i< ProcNum -1; i++){
                        MPI_Recv(&RecvRank, 1, MPI_INT,
MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
                        printf("\n Hello from process %3d", RecvRank);
                }
        }


        MPI_Finalize();
        return 0;
}
```

Полученный вывод:



:

Проанализировав несколько выводов сообщений, можно сделать вывод, что они выводятся в порядке:

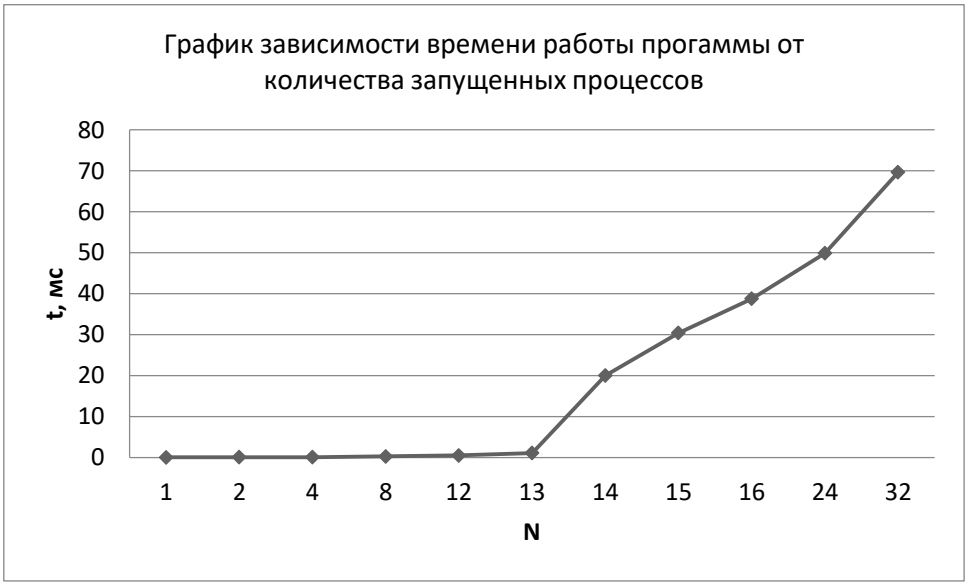
1 – нулевой процесс, тот, который получает сообщения

(2-N) – процессы, расположенные в случайном порядке, этот порядок зависит от состояния системы.

Количество процессов (шт)	Среднее затрачиваемое время (мс)
1	0.0403
2	0.0560
4	0.0670
8	0.2404
12	0.4966
13	1.0935
14	19.9861
15	30.4054

16	38.7814
24	49.8196
32	69.6338

Табл. 1 – Результаты работы программы на разном количестве процессов.



Расчеты ускорения программы выполним по формуле:

$$S_p(n) = T_1(n)/T_p(n)$$

Количество процессов P (шт)	Среднее затрачиваемое время (мс)	Ускорение S_p
1	0.0403	1
2	0.0560	0.71
4	0.0670	0.59
8	0.2404	0.17
12	0.4966	0.07
13	1.0935	0.03

14	19.9861	0.0020
15	30.4054	0.0013
16	38.7814	0.0010
24	49.8196	0.0008
32	69.6338	0.0005

Табл. 2 – Результаты расчетов ускорения программы.

Листинг модифицированной программы:

```
#include <time.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
        int ProcNum;
        int ProcRank;
        int RecvRank;
        double starttime, endtime;

        MPI_Status Status;

        MPI_Init(&argc, &argv);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
        MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);

        starttime = MPI_Wtime();
        if (ProcRank != 0) {
                MPI_Send(&ProcRank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        else if(ProcRank == 0){
                printf("Hello from process %3d\n", ProcRank);
                for(int i=1; i<ProcNum; i++){
                        MPI_Recv(&RecvRank, 1, MPI_INT, i,
MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
                        printf("Hello from process %3d\n", RecvRank);
                }
        }

        endtime = MPI_Wtime();
        MPI_Finalize();

        printf("That took %.4f milliseconds, process number %d\n",
(endtime-starttime)*1000, ProcRank);

        return 0;
}
```

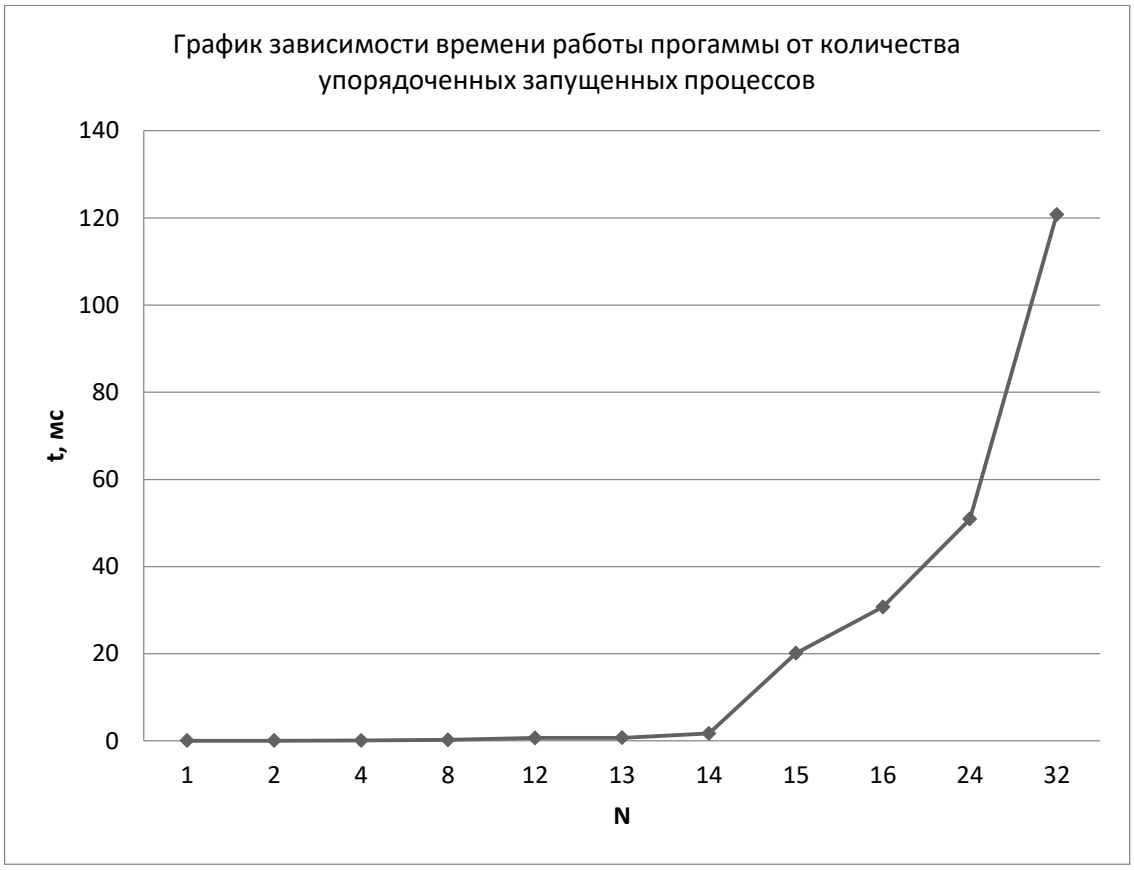
Полученный вывод:

```
Hello from process   0
Hello from process   1
Hello from process   2
Hello from process   3
Hello from process   4
Hello from process   5
Hello from process   6
Hello from process   7
Hello from process   8
Hello from process   9
That took 0.3088 milliseconds, process number 0
That took 0.0130 milliseconds, process number 1
That took 0.0173 milliseconds, process number 2
That took 0.0193 milliseconds, process number 3
That took 0.0156 milliseconds, process number 5
That took 0.0113 milliseconds, process number 6
That took 0.0122 milliseconds, process number 4
That took 0.0148 milliseconds, process number 7
That took 0.0134 milliseconds, process number 8
That took 0.0181 milliseconds, process number 9
```

Количество процессов (шт)	Среднее затрачиваемое время (мс)
1	0.0407
2	0.044
4	0.077
8	0.227
12	0.681

13	0,73
14	1.718
15	20.1132
16	30.7
24	50.87
32	120.74

Табл. 3 – Результаты времени работы программы на разном количестве упорядоченных процессов.



Вывод

В ходе выполнения лабораторной работы были изучены основные функции и устройство библиотеки MPI. Был проведен анализ скорости работы программы и ее ускорения для разного количества процессов: время выполнения с увеличением числа процессов растет т.к. нулевой процесс

последовательно принимает сообщения от остальных процессов, вследствие чего возникают дедлоки. Соответственно ускорение программы с увеличением числа процессов уменьшается.

Время выполнения программы значительно увеличивается после прохождения отметки в 12 процессов. Это происходит потому, что компьютер имеет 6 ядерный процесс с 12-ю потоками. Получается, происходит конкуренция процессов: некоторые процессы (отправляющие) ожидают, пока система выделит для них квант времени (освободится поток) и они сделают отправку сообщения.

Что касается сравнения времени работы упорядоченного получения и случайного получения сообщений, то временная разница между двумя подходами минимальна. Время выполнения кода может зависеть от множества факторов, таких как размер сообщений, количество процессов и алгоритмы реализации MPI. В нашем случае размер сообщения мал и их обработка происходит достаточно быстро. Поэтому экспериментально выявить разницу в скорости выполнения обработки запросов достаточно сложно.