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

ОТЧЕТ

по лабораторной работе №6
по дисциплине «Параллельные алгоритмы»
Тема: Виртуальные топологии

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы

Целью данной работы является демонстрация работы с виртуальными топологиями в MPI и выполнение простого сдвига данных между процессами.

Задание

Вариант 9

В каждом подчиненном процессе дано вещественное число. Определить для всех процессов декартову топологию в виде одномерной решетки и осуществить простой сдвиг исходных данных с шагом −1 (число из каждого подчиненного процесса пересылается в предыдущий процесс). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функций MPI_Send и MPI_Recv. Во всех процессах, получивших данные, вывести эти данные.

Выполнение работы

В ходе лабораторной работы была разработана программа на языке C++ с использованием библиотеки MPI. Функции данной библиотеки позволяют создать декартову топологию и производить в ней некоторые операции.

Приведем подробное описание написанной программы:

Программа начинается с инициализации MPI. После этого идет получение ранга текущего процесса и общего количества процессов в коммуникаторе MPI_COMM_WORLD. Затем идет проверка на количество процессов: программа требует минимум два процесса для работы. Если количество процессов меньше двух, процесс с рангом 0 выводит сообщение об ошибке и программа завершается.

Функция *MPI_Cart_create* создает новый коммуникатор с декартовой топологией на основе существующего коммуникатора. Она имеет следующий прототип:

*int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims, const int *periods, int reorder, MPI_Comm *comm_cart)*

- *comm_old*: Исходный коммуникатор, на основе которого создается новая топология.
- *ndims*: Количество измерений декартовой решетки.
- *dims*: Массив размером ndims, указывающий количество процессов в каждом измерении.
- *periods*: Массив размером ndims, указывающий, являются ли границы решетки периодическими (1 — периодические, 0 — непериодические).
- *reorder*: Флаг, указывающий, можно ли изменять порядок процессов в новом коммуникаторе (1 — можно, 0 — нельзя).

- *comm_cart*: Указатель на новый коммуникатор с декартовой топологией.

После этого происходит получение координат процесса в декартовой топологии с помощью функции *MPI_Cart_coords*, она имеет следующий прототип:

*int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)*

- *comm*: Коммуникатор с декартовой топологией.
- *rank*: Ранг процесса в исходном коммуникаторе.
- *maxdims*: Максимальное количество измерений (размер массива coords).
- *coords*: Массив для хранения координат процесса в декартовой топологии.

После этого идет инициализация данных, которые будут использоваться для отправки. Для удобства отслежки эти данные будут совпадать с рангами процессов.

Затем идет определение соседей процессов в декартовой топологии для отправки с помощью функции *MPI_Cart_shift*. Она позволяет определить, какой процесс является источником и какой процесс является назначением для пересылки данных в заданном направлении и с заданным шагом. Ее прототип выглядит следующим образом:

*int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)*

- *comm*: Коммуникатор с декартовой топологией.
- *direction*: Индекс измерения, в котором выполняется сдвиг (для одномерной решетки это 0).
- *disp*: Шаг сдвига. Положительное значение означает сдвиг вперед, отрицательное — назад.

- *rank_source*: Указатель на переменную, в которую будет записан ранг процесса-источника.
- *rank_dest*: Указатель на переменную, в которую будет записан ранг процесса-назначения.

Дальше *MPI_Send* отправляет данные предыдущему процессу, а *MPI_Recv* принимает данные от следующего процесса. Процесс с рангом 0 сначала отправляет данные, а затем принимает. Все остальные процессы сначала принимают данные, а затем отправляют. Это сделано для того, чтобы не вызвать взаимоблокировку (deadlock), все процессы не могут сначала пытаться отправить данные, а затем принять их. Это может привести к ситуации, когда все процессы ждут приема данных, которые никогда не будут отправлены.

В целом, передача данных может быть выполнена, например, при помощи MPI_Sendrecv, но программа должна была соответствовать условию задания.

Реализованная программа:

```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // программа не работает на одном процессе
    if (size < 2) {
        if (rank == 0) {
              std::cerr << "Для работы программы необходимо минимум
два процесса!" << std::endl;
        }
        MPI_Finalize();
        return 0;
    }

    // размерность декартовой топологии
    int dims = 1; // одномерная решетка
     int periods[] = {1}; // периодичность по осям (1 означает, что
решетка циклична)
```

```
    int reorder = 1; // возможность изменения порядка процессов

    // создание декартовой топологии
    MPI_Comm cart_comm;
    int coords[1]; // координаты процесса в декартовой решетке
     MPI_Cart_create(MPI_COMM_WORLD, dims, &size, periods, reorder,
&cart_comm);

        // получаем координаты текущего процесса в декартовой
топологии
    MPI_Cart_coords(cart_comm, rank, 1, coords);

    double data = rank * 1.0;

    // определяем соседей для пересылки
    int source, dest;
     MPI_Cart_shift(cart_comm, 0, -1, &source, &dest); // сдвиг с
шагом -1 (отправка в предыдущий процесс)

    // отправка и принятие данных
    double received_data;
    MPI_Status status;

    if (rank == 0) { //процесс 0
        // отправляет данные последнему процессу (size-1)
        MPI_Send(&data, 1, MPI_DOUBLE, dest, 0, cart_comm);
        // принимает данные от процесса 1
         MPI_Recv(&received_data, 1, MPI_DOUBLE, (rank + 1) % size,
0, cart_comm, &status);
    } else { //остальные процессы
        // принимают данные от следующего
         MPI_Recv(&received_data, 1, MPI_DOUBLE, (rank + 1) % size,
0, cart_comm, &status);
        // отправляют данные предыдущему
        MPI_Send(&data, 1, MPI_DOUBLE, dest, 0, cart_comm);
    }

    // вывод полученных данных
     std::cout << "Процесс " << rank << " (данные = " << data << ")
получил данные: " << received_data << std::endl;

    MPI_Finalize();

    return 0;
}
```

Листинг программы:

```
katya@katya:~/ПА/lb6$ mpirun -np 2 ./lab6
Процесс 0 (данные = 0) получил данные: 1
Процесс 1 (данные = 1) получил данные: 0
katya@katya:~/ПА/lb6$ mpirun -np 4 ./lab6
Процесс 0 (данные = 0) получил данные: 1
Процесс 1 (данные = 1) получил данные: 2
Процесс 3 (данные = 3) получил данные: 0
```

```
Процесс 2 (данные = 2) получил данные: 3
katya@katya:~/ПА/lb6$ mpirun -np 8 ./lab6
Процесс 0 (данные = 0) получил данные: 1
Процесс 1 (данные = 1) получил данные: 2
Процесс 2 (данные = 2) получил данные: 3
Процесс 3 (данные = 3) получил данные: 4
Процесс 4 (данные = 4) получил данные: 5
Процесс 5 (данные = 5) получил данные: 6
Процесс 6 (данные = 6) получил данные: 7
Процесс 7 (данные = 7) получил данные: 0
katya@katya:~/ПА/lb6$ mpirun -np 16 ./lab6
Процесс 0 (данные = 0) получил данные: 1
Процесс 1 (данные = 1) получил данные: 2
Процесс 2 (данные = 2) получил данные: 3
Процесс 3 (данные = 3) получил данные: 4
Процесс 6 (данные = 6) получил данные: 7
Процесс 4 (данные = 4) получил данные: 5
Процесс 5 (данные = 5) получил данные: 6
Процесс 7 (данные = 7) получил данные: 8
Процесс 8 (данные = 8) получил данные: 9
Процесс 9 (данные = 9) получил данные: 10
Процесс 10 (данные = 10) получил данные: 11
Процесс 12 (данные = 12) получил данные: 13
Процесс 11 (данные = 11) получил данные: 12
Процесс 13 (данные = 13) получил данные: 14
Процесс 14 (данные = 14) получил данные: 15
Процесс 15 (данные = 15) получил данные: 0
```

Видно, что программа отработала корректно: данные из корректно отправились и корректно доставились, получатели тоже были выбраны корректно.

Измененная программа для измерения времени:

```
#include <iostream>
#include <mpi.h>
#include <fstream>
#include <vector>

int main(int argc, char** argv) {
    std::ofstream fout("./data.txt", std::ios::app);

    int data_size;
    if (argc > 1)
        data_size = atoi(argv[1]);

    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // программа не работает на одном процессе
    if (size < 2) {
```

```
        if (rank == 0) {
             std::cerr << "Для работы программы необходимо минимум
два процесса!" << std::endl;
        }
        MPI_Finalize();
        return 0;
    }

    double start, end, time;
    start = MPI_Wtime();

    // размерность декартовой топологии
    int dims = 1; // одномерная решетка
     int periods[] = {1}; // периодичность по осям (1 означает, что
решетка циклична)
    int reorder = 1; // возможность изменения порядка процессов

    // создание декартовой топологии
    MPI_Comm cart_comm;
    int coords[1]; // координаты процесса в декартовой решетке
     MPI_Cart_create(MPI_COMM_WORLD, dims, &size, periods, reorder,
&cart_comm);

        // получаем координаты текущего процесса в декартовой
топологии
    MPI_Cart_coords(cart_comm, rank, 1, coords);

    // Инициализация данных
    std::vector<double> data(data_size, rank * 1.0);
    std::vector<double> received_data(data_size);

    // определяем соседей для пересылки
    int source, dest;
     MPI_Cart_shift(cart_comm, 0, -1, &source, &dest); // сдвиг с
шагом -1 (отправка в предыдущий процесс)

    // отправка и принятие данных
    MPI_Status status;

    if (rank == 0) { //процесс 0
        // отправляет данные последнему процессу (size-1)
            MPI_Send(data.data(), data_size, MPI_DOUBLE, dest, 0,
cart_comm);
        // принимает данные от процесса 1
             MPI_Recv(received_data.data(), data_size, MPI_DOUBLE,
(rank + 1) % size, 0, cart_comm, &status);
    } else { // Остальные процессы
        // аринимают данные от следующего
             MPI_Recv(received_data.data(), data_size, MPI_DOUBLE,
(rank + 1) % size, 0, cart_comm, &status);
        // отправляют данные предыдущему
            MPI_Send(data.data(), data_size, MPI_DOUBLE, dest, 0,
cart_comm);
    }

    // вывод полученных данных
```

```
    //std::cout << "Процесс " << rank << " (данные = " << data <<
") получил данные: " << received_data << std::endl;
    end = MPI_Wtime();
    time = end - start;

    if (rank == 0) {
        fout << "Data size: " << data_size << " Time: " << time <<
" Processes: " << size << '\n';
    }
    fout.close();
    MPI_Finalize();

    return 0;
}
```

Листинг программы с замерами времени:

```
Data size: 10 Time: 0.000125485 Processes: 2
Data size: 10 Time: 0.000326714 Processes: 3
Data size: 10 Time: 0.000413207 Processes: 4
Data size: 10 Time: 0.000456508 Processes: 5
Data size: 10 Time: 0.000341533 Processes: 6
Data size: 10 Time: 0.000569491 Processes: 7
Data size: 10 Time: 0.000578317 Processes: 8
Data size: 10 Time: 0.000781559 Processes: 9
Data size: 10 Time: 0.000672264 Processes: 10
Data size: 10 Time: 0.00085105 Processes: 11
Data size: 10 Time: 0.000786489 Processes: 12
Data size: 10 Time: 0.00877321 Processes: 13
Data size: 10 Time: 0.0316064 Processes: 14
Data size: 10 Time: 0.0350438 Processes: 15
Data size: 10 Time: 0.0306613 Processes: 16
Data size: 100 Time: 0.00019144 Processes: 2
Data size: 100 Time: 0.000331122 Processes: 3
Data size: 100 Time: 0.000438595 Processes: 4
Data size: 100 Time: 0.000398459 Processes: 5
Data size: 100 Time: 0.000440318 Processes: 6
Data size: 100 Time: 0.000580842 Processes: 7
Data size: 100 Time: 0.000591182 Processes: 8
Data size: 100 Time: 0.000789003 Processes: 9
Data size: 100 Time: 0.000727508 Processes: 10
Data size: 100 Time: 0.000785827 Processes: 11
Data size: 100 Time: 0.0185956 Processes: 12
Data size: 100 Time: 0.022416 Processes: 13
Data size: 100 Time: 0.025459 Processes: 14
Data size: 100 Time: 0.0352418 Processes: 15
Data size: 100 Time: 0.0368907 Processes: 16
Data size: 1000 Time: 0.000151996 Processes: 2
Data size: 1000 Time: 0.000436871 Processes: 3
Data size: 1000 Time: 0.000232527 Processes: 4
Data size: 1000 Time: 0.000549313 Processes: 5
Data size: 1000 Time: 0.000481986 Processes: 6
Data size: 1000 Time: 0.000773134 Processes: 7
Data size: 1000 Time: 0.000863774 Processes: 8
Data size: 1000 Time: 0.00112913 Processes: 9
Data size: 1000 Time: 0.000839027 Processes: 10
Data size: 1000 Time: 0.00114154 Processes: 11
```

```
Data size: 1000 Time: 0.0176299 Processes: 12
Data size: 1000 Time: 0.0333223 Processes: 13
Data size: 1000 Time: 0.0224342 Processes: 14
Data size: 1000 Time: 0.0441833 Processes: 15
Data size: 1000 Time: 0.0470086 Processes: 16
Data size: 10000 Time: 0.00320485 Processes: 2
Data size: 10000 Time: 0.00568562 Processes: 3
Data size: 10000 Time: 0.00609883 Processes: 4
Data size: 10000 Time: 0.00726089 Processes: 5
Data size: 10000 Time: 0.0086169 Processes: 6
Data size: 10000 Time: 0.0106334 Processes: 7
Data size: 10000 Time: 0.0149535 Processes: 8
Data size: 10000 Time: 0.0151602 Processes: 9
Data size: 10000 Time: 0.0181752 Processes: 10
Data size: 10000 Time: 0.0194063 Processes: 11
Data size: 10000 Time: 0.0279085 Processes: 12
Data size: 10000 Time: 0.0492459 Processes: 13
Data size: 10000 Time: 0.0394055 Processes: 14
Data size: 10000 Time: 0.0673059 Processes: 15
Data size: 10000 Time: 0.117862 Processes: 16
```

Полученный график отображен на рисунке 1.

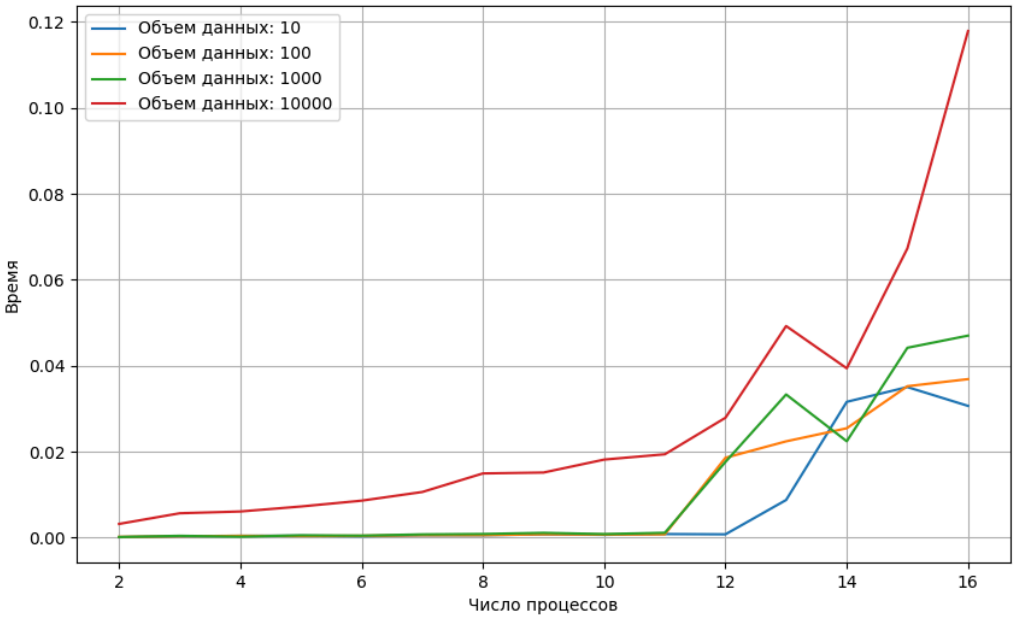

Рисунок 1 – График зависимости времени работы программы от числа запущенных процессов и от объема входных данных

На графике явно видно, что время увеличивается при увеличении количества процессов. Объяснить это можно так: увеличивается количество процессов, значит увеличивается и количество соседей в декартовой решетке, которым нужно отправить данные, а эти соседи

должны еще и принять данные. Также видно, что при увеличении объемов данных увеличивается время работы команды. Резкие скачки времени (например, при объеме данных 1000 и количестве процессов 13) можно обосновать накладными ресурсами. Время работы скрипта также значительно тормозит работу программы.

Теперь можно проанализировать замедление/ускорение работы программы:

Для того, чтобы провести анализ ускорения или замедления работы программы, построим график, используя формулу ускорения: Sp(n) = T1(n)/Tp(n) , где T1(n) – среднее время выполнения алгоритма на двух процессах (последовательная программа), Tp(n) - время выполнения алгоритма на n процессах.

График замедления программы изображен на рисунке 2:

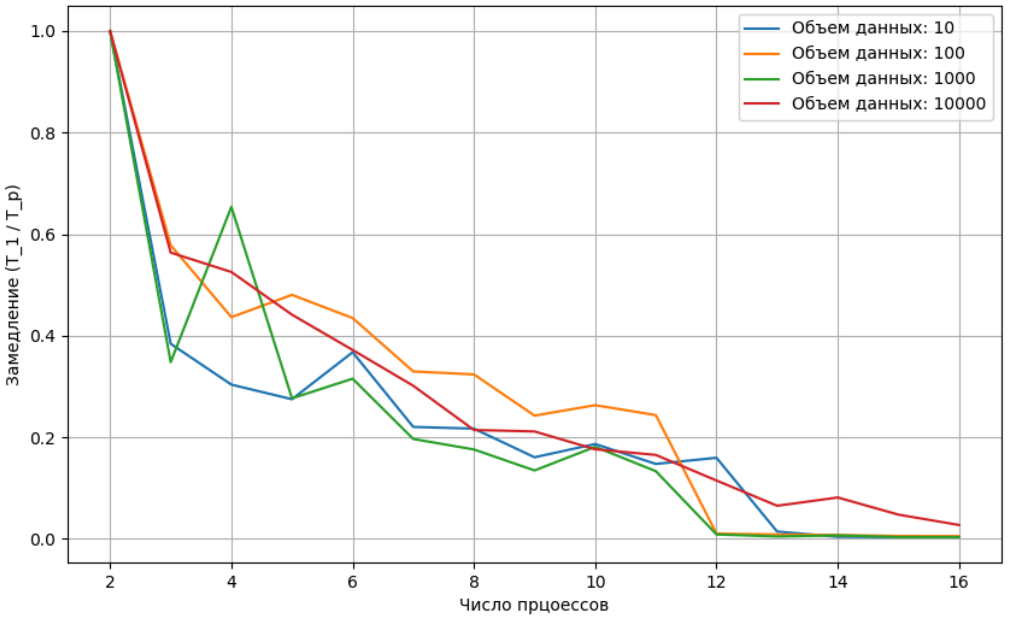

Рисунок 2 – График замедления работы программы в зависимости от количества процессов и объема входных данных

Из графика видно, что при увеличении количества процессов программа замедляется. Это происходит потому, что много времени уходит

на их “взаимодействие”. Глядя на график, можно сказать, что последовательная программа выполняется быстрее. Точки перелома можно обосновать, опять же, накладными расходами.

Сети Петри, отражающие основную логику программы, изображены на рисунке 3.

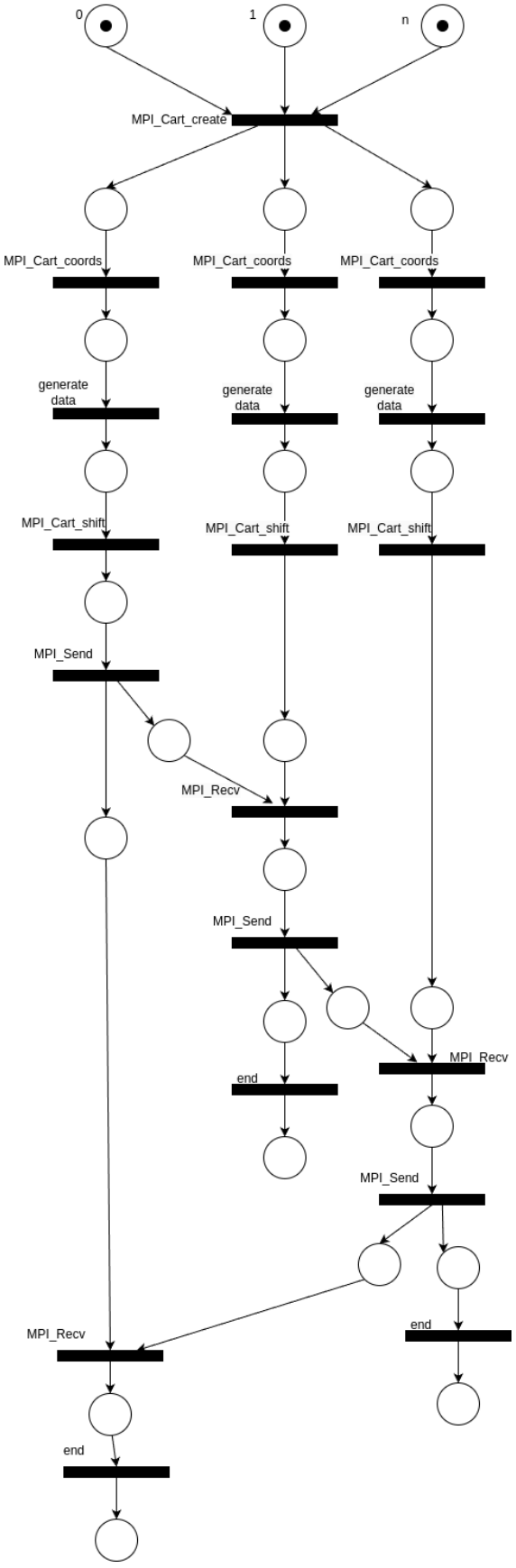

Рисунок 3 – Сети Петри

Вывод

В ходе выполнения лабораторной работы была написана программа, использующая функцию MPI_Cart_create, которая создает новый коммуникатор с декартовой топологией на основе существующего коммуникатора. Также в написанной программе была использована функция *MPI_Cart_shift*. Она позволяет определить, какой процесс является источником и какой процесс является назначением для пересылки данных в заданном направлении и с заданным шагом

В ходе работы был построен график зависимости времени от количества процессов и входных данных, который показывает, что чем больше количество процессов/размер входных данных, тем больше времени нужно на это. Также был построен график замедления, отражающий, что время выполнения параллельной программы больше по сравнению с последовательной программой.