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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Параллельные алгоритмы»
Тема: Использование аргументов-джокеров

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы

Изучение параллельных взаимодействий процессов на примере функций библиотеки MPI. Написание программы с использованием аргументов-джокеров согласно поставленной задаче. Анализ зависимости времени работы программы от числа процессов при различном объеме данных.

Задание

Вариант 3

Имитация топологии «звезда» (процесс с номером 0 реализует функцию центрального узла). Процессы в случайном порядке генерируют пакеты, состоящие из адресной и информационной части и передают их в процесс 0. Маршрутная часть пакета содержит номер процесса-адресата. Процесс 0 переадресовывает пакет адресату. Адресат отчитывается перед процессом 0 в получении. Процесс 0 информирует процесс-источник об успешной доставке.

Выполнение работы

В ходе лабораторной работы была разработана программа на языке C++ с использованием библиотеки MPI. Функции данной библиотеки позволяют организовать посылки между процессами с использованием аргументов-джокеров. В данной лабораторной работе была реализована имитация топологии сети «звезда».

В этой топологии один процесс выполняет роль центрального узла (процесс с номером 0), а остальные процессы генерируют и передают пакеты через центральный узел. Рассмотрим механизм работы подробнее.

Процесс с номером 0 выполняет роль центрального узла. Остальные процессы (с номерами от 1 до size-1) генерируют пакеты и передают их центральному узлу. Пакет состоит из маршрутной части (номер процесса-источника и номер процесса-адресата), важно, что номер адресата генерируется произвольно, причем гарантируется, что номер адресата не будет совпадать со своим и с нулевым, и информационной части (произвольные данные).

Сгенерированные данные процессы отправляются центральному узлу. Принимая с помощью функции MPI_Recv, где в качестве отправителя может быть любой процесс, благодаря аргументу джокеру, центральный процесс записывает данные в буферный пакет, который будет зафиксирован в таблице.

После этого центральный узел переадресовывает пакеты процессам-адресатам согласно маршрутной части.

Принятие процессами сообщений, которые были переадресованы им, происходит исходя из того, что в один процесс могут быть отправлены несколько пакетов, так как адрес генерируется случайно. Данная особенность реализована при помощи цикла, который организован так, чтобы были получены все пакеты. Это необходимо, чтобы избежать

случая, когда процесс не дождался посылки, хотя она была отправлена. При помощи MPI_Iprobe будет устанавливаться флаг о том, есть ли входящие посылки.

После получения пакетов центральному процессу будет отправлено подтверждение. В ходе работы была обнаружена следующая особенность: некоторые процессы получали не те сообщения подтверждения от нулевого процесса, или не дождались вовсе. Во избежание этой ошибки было использовано sleep(2), чтобы создать искусственную синхронизацию.

Центральный узел информирует процессы-источники об успешной доставке пакетов после получения отчета от адресатов.

Код реализованной программы:

```
#include <mpi.h>
#include <stdio.h>
#include <map>
#include <vector>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

#define TAG 0

typedef struct
{
    int source;
    int destination;
    int data;
} Packet;

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);

    if (size < 2)
    {
        printf("This program requires at least 2 processes.\n");
        MPI_Finalize();
        return 1;
    }

    srand(time(0) + rank); // Инициализация генератора случайных чисел

    if (rank == 0)
    {
        std::map<int, Packet> packets_map;
```

```
        // Получаем пакеты от всех процессов
        for (int i = 1; i < size; i++)
        {
            Packet buf;
               MPI_Recv(&buf, sizeof(Packet), MPI_BYTE, MPI_ANY_SOURCE, TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            packets_map[buf.source] = buf;
        }

        // Переадресовываем пакеты процессам-адресатам
        for (int i = 1; i < size; i++)
        {
                        MPI_Send(&packets_map[i],  sizeof(Packet),  MPI_BYTE,
packets_map[i].destination, TAG, MPI_COMM_WORLD);
        }

        // Получаем подтверждения от адресатов
        for (int i = 1; i < size; i++)
        {
            bool delivery_confirmation;
                    MPI_Recv(&delivery_confirmation,  sizeof(bool),  MPI_BYTE,
MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        sleep(2);

        // Информируем источники об успешной доставке
        for (int i = 1; i < size; i++)
        {
            bool delivery_confirmation = true;
                    MPI_Send(&delivery_confirmation,  sizeof(bool),  MPI_BYTE,
packets_map[i].source, TAG, MPI_COMM_WORLD);
        }
    }
    else
    {
        // Генерация пакета с случайным адресатом (кроме самого себя и 0)
        Packet packet;
        packet.source = rank;
        do
        {
            packet.destination = rand() % size;
        } while (packet.destination == rank || packet.destination == 0);
        packet.data = rank;

        // Отправляем пакет процессу 0
        MPI_Send(&packet, sizeof(Packet), MPI_BYTE, 0, TAG, MPI_COMM_WORLD);
          std::cout << "Process " << rank << " sent a packet to process " <<
packet.destination << std::endl;

        // Ожидание получения переадресованного пакета
        std::map<int, Packet> packets_map;
        MPI_Status status;
        int attempts = 0;

        while (attempts < 100000) // Необходимо, чтобы избежать случая, когда
процесс не дождался посылки, хотя она была отправлена
        {
            int flag = 0;
            MPI_Iprobe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
            if (flag)
```

```
            {
                Packet received_packet;
                 MPI_Recv(&received_packet, sizeof(Packet), MPI_BYTE, 0, TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                  std::cout << "Process " << rank << " received a packet from
process " << received_packet.source << std::endl;
                packets_map[received_packet.source] = received_packet;
            }
            else
            {

                attempts++;
            }
        }

        for (int i = 0; i < packets_map.size(); i++)
        {
            bool delivery_confirmation = true;
             MPI_Send(&delivery_confirmation, sizeof(bool), MPI_BYTE, 0, TAG,
MPI_COMM_WORLD);
        }

        // Получение подтверждения успешной доставки для исходного пакета
        bool success;
          MPI_Recv(&success, sizeof(bool), MPI_BYTE, 0, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        if (success)
        {
                  std::cout << "Process " << rank << " received delivery
confirmation for its packet." << std::endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```

Измененная программа для измерения времени:

```
#include <mpi.h>
#include <stdio.h>
#include <map>
#include <vector>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

#define TAG 0

typedef struct
{
    int source;
    int destination;
    int* data;
} Packet;
```

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

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

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

    if (size < 2)
    {
        MPI_Finalize();
        return 1;
    }
    double start, end, diff;

     srand(time(0) + rank); // Инициализация генератора случайных
чисел

    if (rank == 0)
    {
        start = MPI_Wtime();
        std::map<int, Packet> packets_map;

        // Получаем пакеты от всех процессов
        for (int i = 1; i < size; i++)
        {
            Packet buf;
                        MPI_Recv(&buf,  sizeof(Packet),  MPI_BYTE,
MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            packets_map[buf.source] = buf;
        }

        // Переадресовываем пакеты процессам-адресатам
        for (int i = 1; i < size; i++)
        {
               MPI_Send(&packets_map[i], sizeof(Packet), MPI_BYTE,
packets_map[i].destination, TAG, MPI_COMM_WORLD);
        }

        // Получаем подтверждения от адресатов
        for (int i = 1; i < size; i++)
        {
            bool delivery_confirmation;
                    MPI_Recv(&delivery_confirmation,  sizeof(bool),
MPI_BYTE, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        sleep(2);
```

```
        // Информируем источники об успешной доставке
        for (int i = 1; i < size; i++)
        {
            bool delivery_confirmation = true;
                    MPI_Send(&delivery_confirmation,  sizeof(bool),
MPI_BYTE, packets_map[i].source, TAG, MPI_COMM_WORLD);
        }
        end = MPI_Wtime();
        diff = end - start;
         fout << "Data size: " << data_size << " Time: " << diff <<
" Processes: " << size << std::endl;
        fout.close();
    }
    else
    {
        start = MPI_Wtime();
          // Генерация пакета с случайным адресатом (кроме самого
себя и 0)
        Packet packet;
        packet.source = rank;
        do
        {
            packet.destination = rand() % size;
         } while (packet.destination == rank || packet.destination
== 0);
        packet.data = new int[data_size];

        // Отправляем пакет процессу 0
             MPI_Send(&packet,  sizeof(Packet),  MPI_BYTE,  0,  TAG,
MPI_COMM_WORLD);

        // Ожидание получения переадресованного пакета
        std::map<int, Packet> packets_map;
        MPI_Status status;
        int attempts = 0;

          while (attempts < 100000) // Необходимо, чтобы избежать
случая,  когда  процесс  не  дождался  посылки,  хотя  она  была
отправлена
        {
            int flag = 0;
                 MPI_Iprobe(0,  MPI_ANY_TAG,  MPI_COMM_WORLD,  &flag,
&status);
            if (flag)
            {
                Packet received_packet;
                        MPI_Recv(&received_packet,  sizeof(Packet),
MPI_BYTE, 0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                             packets_map[received_packet.source]  =
received_packet;
            }
            else
            {

                attempts++;
            }
```

```
        }

        for (int i = 0; i < packets_map.size(); i++)
        {
            bool delivery_confirmation = true;
                    MPI_Send(&delivery_confirmation,  sizeof(bool),
MPI_BYTE, 0, TAG, MPI_COMM_WORLD);
        }

         // Получение подтверждения успешной доставки для исходного
пакета
        bool success;
              MPI_Recv(&success,  sizeof(bool),  MPI_BYTE,  0,  TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}
```

Скрипт, запускающий программу на разных процессах с разными входными данными:

```
#!/bin/bash

> data/data.txt

mpic++ -o ./time time.cpp

sizes=(10 100 1000 10000)

for size in "${sizes[@]}"
do
    for nprocs in {3..16}
    do
        mpiexec -n $nprocs ./time $size
    done
done

python3 graphics.py
```

Программа, отрисовывающая графики по данным из файла, где хранится количество процессов, размер входных данных и время:

```
import matplotlib.pyplot as plt
from datetime import datetime

data = {}

with open('./data/data.txt', 'r') as file:
    for line in file:
        parts = line.split()
```

```
        size = int(parts[2])
        time = float(parts[4])
        procs = int(parts[6])

        if size not in data:
            data[size] = {}
        if procs not in data[size]:
            data[size][procs] = []
        data[size][procs].append(time)

plt.figure(figsize=(10, 6))
for size in data:
    processes = sorted(data[size].keys())
    times = [sum(data[size][p]) for p in processes]

    plt.plot(processes, times, label=f'Объем данных: {size}')

plt.xlabel('Число процессов')
plt.ylabel('Время')
plt.legend()
plt.grid(True)
plt.show()
```

Листинг запущенной программы:

```
Data size: 10 Time: 2.0235 Processes: 3
Data size: 10 Time: 2.01306 Processes: 4
Data size: 10 Time: 2.02237 Processes: 5
Data size: 10 Time: 2.01299 Processes: 6
Data size: 10 Time: 2.02221 Processes: 7
Data size: 10 Time: 2.02232 Processes: 8
Data size: 10 Time: 2.02236 Processes: 9
Data size: 10 Time: 2.02216 Processes: 10
Data size: 10 Time: 2.02264 Processes: 11
Data size: 10 Time: 2.0284 Processes: 12
Data size: 10 Time: 2.05465 Processes: 13
Data size: 10 Time: 2.04564 Processes: 14
Data size: 10 Time: 2.04594 Processes: 15
Data size: 10 Time: 2.04705 Processes: 16
Data size: 100 Time: 2.01494 Processes: 3
Data size: 100 Time: 2.01959 Processes: 4
Data size: 100 Time: 2.01029 Processes: 5
Data size: 100 Time: 2.01788 Processes: 6
Data size: 100 Time: 2.02061 Processes: 7
Data size: 100 Time: 2.02082 Processes: 8
Data size: 100 Time: 2.0228 Processes: 9
Data size: 100 Time: 2.02573 Processes: 10
Data size: 100 Time: 2.02326 Processes: 11
Data size: 100 Time: 2.0243 Processes: 12
Data size: 100 Time: 2.03473 Processes: 13
Data size: 100 Time: 2.05749 Processes: 14
Data size: 100 Time: 2.05269 Processes: 15
Data size: 100 Time: 2.04003 Processes: 16
Data size: 1000 Time: 2.01186 Processes: 3
Data size: 1000 Time: 2.0133 Processes: 4
Data size: 1000 Time: 2.01601 Processes: 5
```

```
Data size: 1000 Time: 2.01102 Processes: 6
Data size: 1000 Time: 2.0223 Processes: 7
Data size: 1000 Time: 2.03707 Processes: 8
Data size: 1000 Time: 2.02044 Processes: 9
Data size: 1000 Time: 2.02223 Processes: 10
Data size: 1000 Time: 2.02313 Processes: 11
Data size: 1000 Time: 2.02145 Processes: 12
Data size: 1000 Time: 2.04165 Processes: 13
Data size: 1000 Time: 2.04562 Processes: 14
Data size: 1000 Time: 2.04997 Processes: 15
Data size: 1000 Time: 2.04993 Processes: 16
Data size: 10000 Time: 2.01504 Processes: 3
Data size: 10000 Time: 2.01121 Processes: 4
Data size: 10000 Time: 2.02043 Processes: 5
Data size: 10000 Time: 2.02239 Processes: 6
Data size: 10000 Time: 2.02004 Processes: 7
Data size: 10000 Time: 2.02227 Processes: 8
Data size: 10000 Time: 2.02209 Processes: 9
Data size: 10000 Time: 2.02228 Processes: 10
Data size: 10000 Time: 2.02266 Processes: 11
Data size: 10000 Time: 2.02421 Processes: 12
Data size: 10000 Time: 2.03387 Processes: 13
Data size: 10000 Time: 2.06529 Processes: 14
Data size: 10000 Time: 2.05797 Processes: 15
Data size: 10000 Time: 2.05112 Processes: 16
```

Полученный график представлен на рисунке 1:

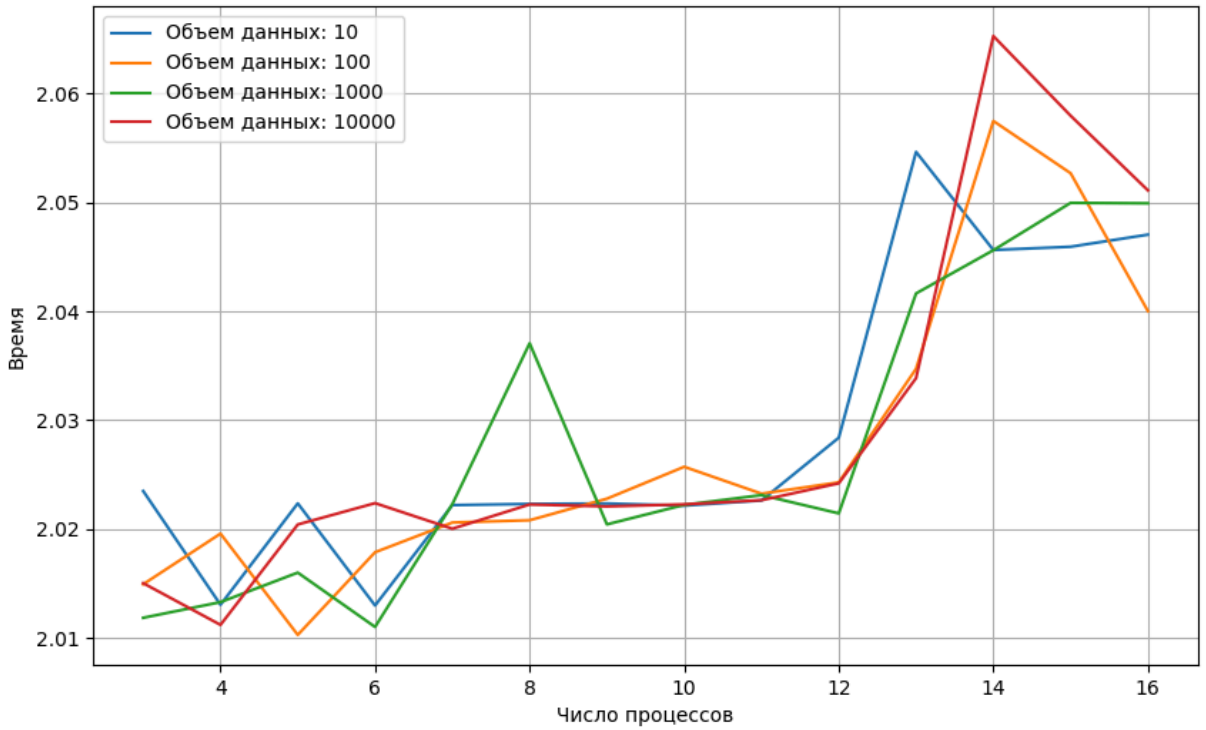


Рисунок 1 – График зависимости времени от количества процессов и объема данных

На графике явно видно, что время увеличивается при увеличении количества процессов. Объяснить это можно так: увеличивается количество процессов, значит увеличивается и время взаимодействия между ними, больше происходит отправлений, больше происходит принятий. Также видно, что при увеличении объемов данных увеличивается время работы команды. Резкие скачки времени (например, при объеме данных 1000 и количестве процессов 8) можно обосновать накладными ресурсами. Известно, что в физике такие точки принято откидывать при проведении экспериментов, называя их ошибочными. Время работы скрипта также значительно тормозит работу программы. В целом, для такой программы нормальной практикой считается “притормаживать”.

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 трех процессах (последовательная программа), $Tp(n)$ - время выполнения алгоритма на n процессах.

График замедления программы изображен на рисунке 2:

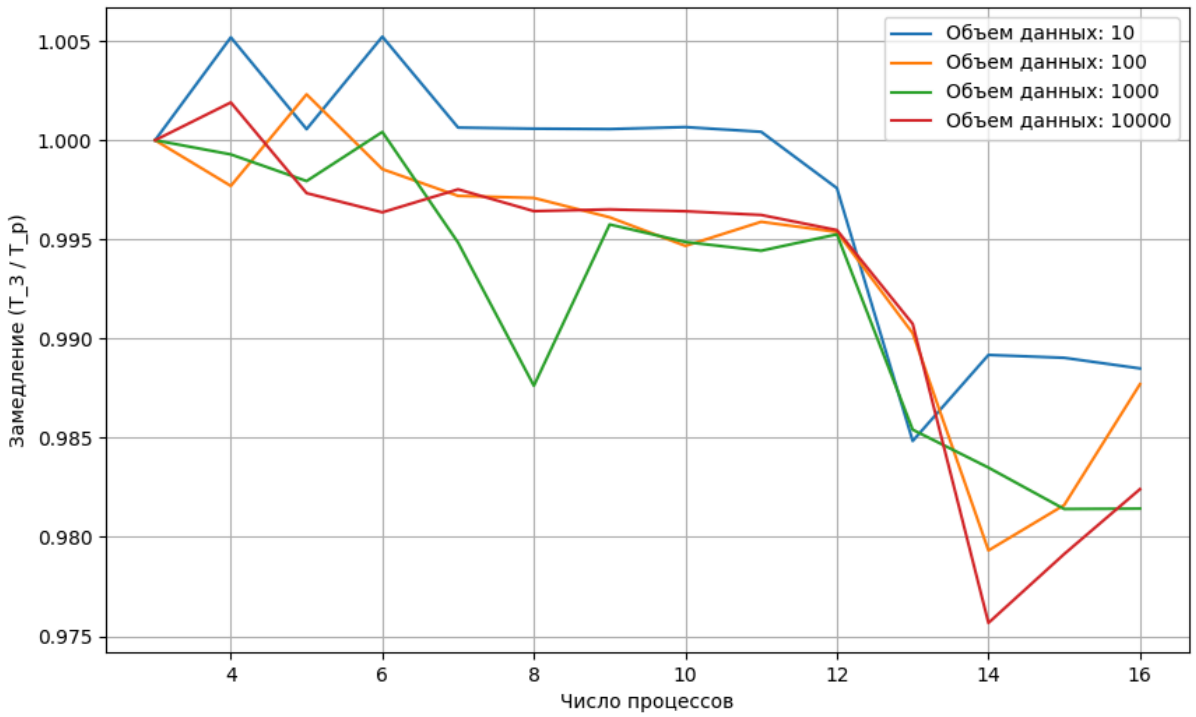
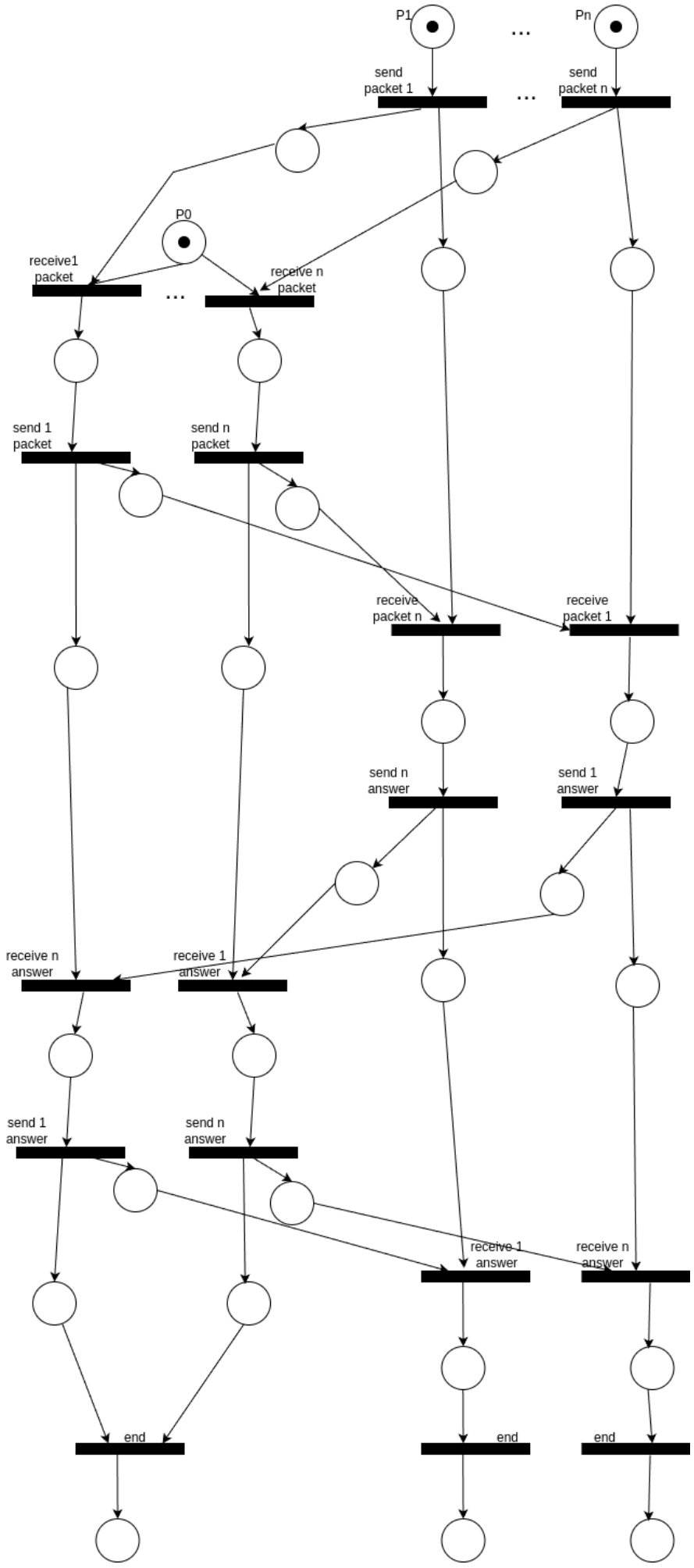


Рисунок 2 – Замедление работы программы в зависимости от количества процессов и объема данных

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. Видно, что в первую очередь идет отправка пакетов из n-ых процессов. Нулевой процесс их принимает, затем отправляет адресатам. Адресаты принимают и отправляют подтверждение нулевому процессу. После этого нулевой процесс отправляет эти подтверждения процессам-источникам.


P1
...
Pn
send
packet 1
...
send
packet n
P0
receive1
packet
...
receive n
packet
send 1
packet
send n
packet
receive
packet n
receive
packet 1
send n
answer
send 1
answer
receive n
answer
receive 1
answer
send 1
answer
send n
answer
receive 1
answer
receive n
answer
end
end
end

Вывод

В ходе выполнения лабораторной работы была написана программа, имитирующая топологию “звезда”, где нулевой процесс реализует функцию центрального узла. В программе было задействовано использование специальных параметров: джокеров. Джокер – родовой термин, имеющий значение "что-либо, отвечающее очень общему множеству характеристик". MPI_ANY_SOURCE позволяет получателю получить сообщения от любого отправителя, а MPI_ANY_TAG позволяет получателю получить любой тип сообщения от отправителя. 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.