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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Параллельные алгоритмы»
Тема: Коллективные операции

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы

Создать программу для сбора данных с нескольких процессов на один процесс, используя функцию MPI_Gather.

Задание

Вариант 1

В каждом процессе дано вещественное число. Используя функцию MPI_Gather, переслать эти числа в главный процесс и вывести их в порядке возрастания рангов переславших их процессов (первым вывести число, данное в главном процессе).

Выполнение работы

В ходе лабораторной работы была разработана программа на языке C++ с использованием библиотеки MPI. Функции данной библиотеки позволяют организовать сбор данных с нескольких процессов и отправить эти данные на один процесс.

Функция randomFloat генерирует случайное число в диапазоне от 0 до 1. Она использует стандартную функцию rand() и делит результат на RAND_MAX, чтобы получить значение типа float.

Каждый процесс создает сообщение, содержащее свой ранг и случайное число, сгенерированное функцией randomFloat.

MPI_Datatype MPI_Message объявляет новый тип данных MPI. Types[2] и block_lengths[2] определяют типы данных и их количество в структуре Message. Offsets[2] определяет смещения полей в структуре Message. MPI_Type_create_struct создает новый тип данных MPI, который представляет собой структуру Message. MPI_Type_commit фиксирует этот тип данных, чтобы его можно было использовать в MPI-вызовах.

Процесс с рангом 0 создает вектор messages, который будет содержать все сообщения, полученные от других процессов. Размер вектора равен количеству процессов (world_size).

Сбор сообщений происходит с помощью MPI_Gather(&message, 1, MPI_Message, messages.data(), 1, MPI_Message, 0, MPI_COMM_WORLD). &message — адрес данных, которые будут отправлены. Данную функцию лучше описать подробно:

- 1 — количество элементов, которые будут отправлены.
- MPI_Message — тип данных, который будет использоваться для отправки.
- messages.data() — адрес, куда будут сохранены данные на процессе с рангом 0.

- 1 — количество элементов, которые будут получены на каждом процессе.
- MPI_Message — тип данных, который будет использоваться для получения.
- 0 — ранг процесса, который будет собирать данные.
- MPI_COMM_WORLD — коммуникатор, в котором происходит обмен данными.

  После этого нулевой процесс принимает сообщения.

  Реализованная программа:

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

struct Message
{
    int rank;
    float message;
};

float randomFloat()
{
    return (float)(rand()) / (float)(RAND_MAX);
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int world_rank;
    int world_size;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    srand(time(NULL) + world_rank);
    Message message = {world_rank, randomFloat()};

    MPI_Datatype MPI_Message;
    MPI_Datatype types[2] = {MPI_INT, MPI_FLOAT};
    int block_lengths[2] = {1, 1};
    MPI_Aint offsets[2];

    offsets[0] = offsetof(Message, rank);
    offsets[1] = offsetof(Message, message);

        MPI_Type_create_struct(2, block_lengths, offsets, types,
&MPI_Message);
```

```
    MPI_Type_commit(&MPI_Message);

    std::vector<Message> messages;
    if (world_rank == 0)
    {
        messages.resize(world_size);
    }

       MPI_Gather(&message, 1, MPI_Message, messages.data(), 1,
MPI_Message, 0, MPI_COMM_WORLD);

    if (world_rank == 0)
    {
        std::cout << "List of numbers:\n";
        for (int i = 0; i < world_size; i++)
        {
              std::cout << "Message: " << messages[i].message << "
from rank: " << messages[i].rank << std::endl;
        }
    }

    MPI_Type_free(&MPI_Message);
    MPI_Finalize();
    return 0;
}
```

Листинг программы:

```
katya@katya:~/ПА/lb4$ mpirun -np 4 ./main
List of numbers:
Message: 0.490431 from rank: 0
Message: 0.345174 from rank: 1
Message: 0.705015 from rank: 2
Message: 0.0649764 from rank: 3
katya@katya:~/ПА/lb4$ mpirun -np 16 ./main
List of numbers:
Message: 0.880237 from rank: 0
Message: 0.242923 from rank: 1
Message: 0.595456 from rank: 2
Message: 0.456193 from rank: 3
Message: 0.314321 from rank: 4
Message: 0.670445 from rank: 5
Message: 0.035315 from rank: 6
Message: 0.890028 from rank: 7
Message: 0.747914 from rank: 8
Message: 0.605216 from rank: 9
Message: 0.971001 from rank: 10
Message: 0.321837 from rank: 11
Message: 0.181744 from rank: 12
Message: 0.537762 from rank: 13
Message: 0.400089 from rank: 14
Message: 0.252543 from rank: 15
katya@katya:~/ПА/lb4$ mpirun -np 32 ./main
List of numbers:
Message: 0.035315 from rank: 0
Message: 0.890028 from rank: 1
```

```
Message: 0.747914 from rank: 2
Message: 0.605216 from rank: 3
Message: 0.971001 from rank: 4
Message: 0.321837 from rank: 5
Message: 0.181744 from rank: 6
Message: 0.537762 from rank: 7
Message: 0.400089 from rank: 8
Message: 0.252543 from rank: 9
Message: 0.116942 from rank: 10
Message: 0.475271 from rank: 11
Message: 0.333491 from rank: 12
Message: 0.190827 from rank: 13
Message: 0.0475789 from rank: 14
Message: 0.408555 from rank: 15
Message: 0.767391 from rank: 16
Message: 0.126084 from rank: 17
Message: 0.982321 from rank: 18
Message: 0.34259 from rank: 19
Message: 0.198244 from rank: 20
Message: 0.554364 from rank: 21
Message: 0.408367 from rank: 22
Message: 0.279877 from rank: 23
Message: 0.632761 from rank: 24
Message: 0.491612 from rank: 25
Message: 0.350672 from rank: 26
Message: 0.206942 from rank: 27
Message: 0.0676071 from rank: 28
Message: 0.424946 from rank: 29
Message: 0.283541 from rank: 30
Message: 0.643785 from rank: 31
```

Видно, что программа отработала корректно: сообщения выводятся в порядке возрастания рангов.

Измененная программа для измерения времени:

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

struct Message
{
    int rank;
    std::vector<int> message;
};

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

    MPI_Init(&argc, &argv);

    int world_rank;
    int world_size;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

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

    std::vector<int> vec(data_size, world_rank);

    std::vector<int> recv_buffer;
    if (world_rank == 0)
    {
        recv_buffer.resize(world_size * data_size);
    }

    double start_time = MPI_Wtime();

    MPI_Gather(vec.data(), data_size, MPI_INT, recv_buffer.data(),
data_size, MPI_INT, 0, MPI_COMM_WORLD);

    double end_time = MPI_Wtime();
    double time = end_time - start_time;

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

    MPI_Finalize();
    return 0;
}
```

Листинг программы с замерами времени:

```
Data size: 10 Time: 1.094e-05 Processes: 1
Data size: 10 Time: 1.4097e-05 Processes: 2
Data size: 10 Time: 7.9107e-05 Processes: 3
Data size: 10 Time: 1.3615e-05 Processes: 4
Data size: 10 Time: 0.000151952 Processes: 5
Data size: 10 Time: 0.000130763 Processes: 6
Data size: 10 Time: 9.7231e-05 Processes: 7
Data size: 10 Time: 0.000137617 Processes: 8
Data size: 10 Time: 0.000147014 Processes: 9
Data size: 10 Time: 0.000142816 Processes: 10
Data size: 10 Time: 0.000260234 Processes: 11
Data size: 10 Time: 0.000124702 Processes: 12
Data size: 10 Time: 0.00684257 Processes: 13
Data size: 10 Time: 0.000504038 Processes: 14
Data size: 10 Time: 0.000671159 Processes: 15
Data size: 10 Time: 4.9943e-05 Processes: 16
Data size: 100 Time: 1.109e-05 Processes: 1
Data size: 100 Time: 1.1181e-05 Processes: 2
Data size: 100 Time: 9.0388e-05 Processes: 3
Data size: 100 Time: 9.9155e-05 Processes: 4
```

```
Data size: 100 Time: 0.00015554 Processes: 5
Data size: 100 Time: 9.5077e-05 Processes: 6
Data size: 100 Time: 3.4273e-05 Processes: 7
Data size: 100 Time: 0.000117389 Processes: 8
Data size: 100 Time: 0.000478731 Processes: 9
Data size: 100 Time: 0.000559281 Processes: 10
Data size: 100 Time: 0.000466819 Processes: 11
Data size: 100 Time: 0.00112263 Processes: 12
Data size: 100 Time: 0.00048897 Processes: 13
Data size: 100 Time: 0.000482137 Processes: 14
Data size: 100 Time: 0.000724348 Processes: 15
Data size: 100 Time: 0.00351173 Processes: 16
Data size: 1000 Time: 1.3095e-05 Processes: 1
Data size: 1000 Time: 1.644e-05 Processes: 2
Data size: 1000 Time: 0.00010225 Processes: 3
Data size: 1000 Time: 3.1439e-05 Processes: 4
Data size: 1000 Time: 0.000209981 Processes: 5
Data size: 1000 Time: 0.000192628 Processes: 6
Data size: 1000 Time: 0.000407048 Processes: 7
Data size: 1000 Time: 0.000206625 Processes: 8
Data size: 1000 Time: 0.000244956 Processes: 9
Data size: 1000 Time: 0.000148416 Processes: 10
Data size: 1000 Time: 0.000254623 Processes: 11
Data size: 1000 Time: 0.000327549 Processes: 12
Data size: 1000 Time: 0.000746108 Processes: 13
Data size: 1000 Time: 0.000781103 Processes: 14
Data size: 1000 Time: 0.00287103 Processes: 15
Data size: 1000 Time: 0.000387831 Processes: 16
Data size: 10000 Time: 1.1702e-05 Processes: 1
Data size: 10000 Time: 3.4925e-05 Processes: 2
Data size: 10000 Time: 4.6797e-05 Processes: 3
Data size: 10000 Time: 0.000696446 Processes: 4
Data size: 10000 Time: 0.000771305 Processes: 5
Data size: 10000 Time: 0.00150595 Processes: 6
Data size: 10000 Time: 0.00222259 Processes: 7
Data size: 10000 Time: 0.0014772 Processes: 8
Data size: 10000 Time: 0.0014471 Processes: 9
Data size: 10000 Time: 0.00275222 Processes: 10
Data size: 10000 Time: 0.00250282 Processes: 11
Data size: 10000 Time: 0.00277504 Processes: 12
Data size: 10000 Time: 0.00277799 Processes: 13
Data size: 10000 Time: 0.00284169 Processes: 14
Data size: 10000 Time: 0.00659818 Processes: 15
Data size: 10000 Time: 0.00497192 Processes: 16
Data size: 50000 Time: 1.5018e-05 Processes: 1
Data size: 50000 Time: 0.000695794 Processes: 2
Data size: 50000 Time: 0.00166117 Processes: 3
Data size: 50000 Time: 0.0016643 Processes: 4
Data size: 50000 Time: 0.00263554 Processes: 5
Data size: 50000 Time: 0.00288555 Processes: 6
Data size: 50000 Time: 0.00301814 Processes: 7
Data size: 50000 Time: 0.00337405 Processes: 8
Data size: 50000 Time: 0.00414026 Processes: 9
Data size: 50000 Time: 0.00440652 Processes: 10
Data size: 50000 Time: 0.00470942 Processes: 11
Data size: 50000 Time: 0.0051421 Processes: 12
```

```
Data size: 50000 Time: 0.00717914 Processes: 13
Data size: 50000 Time: 0.00756593 Processes: 14
Data size: 50000 Time: 0.0160613 Processes: 15
Data size: 50000 Time: 0.0105178 Processes: 16
```

Полученный график отображен на рисунке 1.

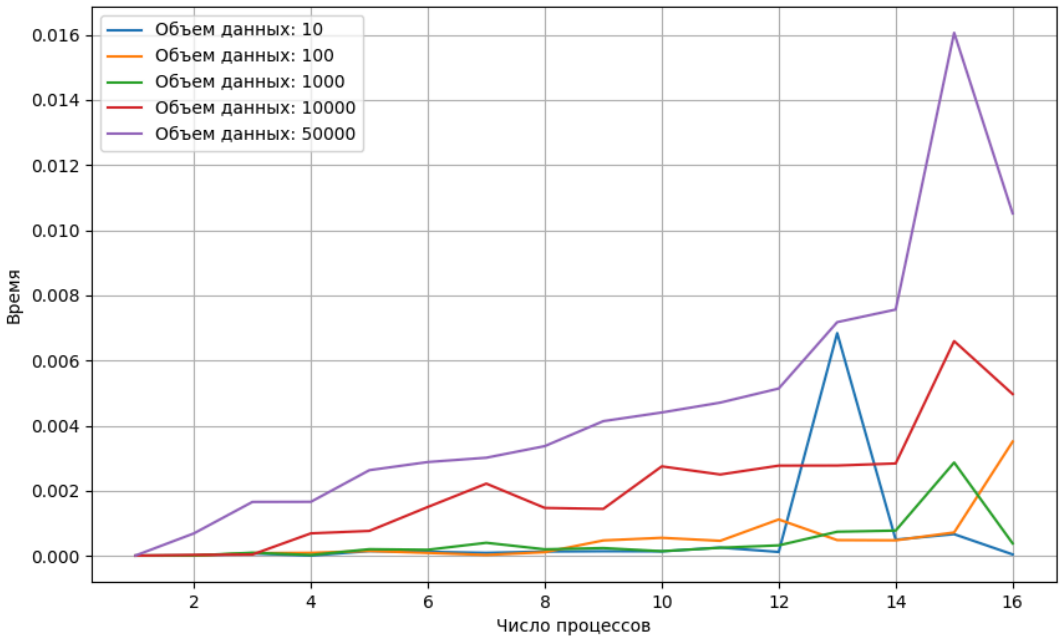

Рисунок 1 – График зависимости времени работы программы от числа запущенных процессов и от объема входных данных

На графике явно видно, что время увеличивается при увеличении количества процессов. Объяснить это можно так: увеличивается количество процессов, значит увеличивается и время принятия сообщений от этих процессов нулевым процессом, а если быть точнее время записи сообщений в буфер нулевого процесса. Также видно, что при увеличении объемов данных увеличивается время работы команды. Резкие скачки времени (например, при объеме данных 10 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 одном процессе (последовательная программа), Tp(n) - время выполнения алгоритма на n процессах.

График замедления программы изображен на рисунке 2:

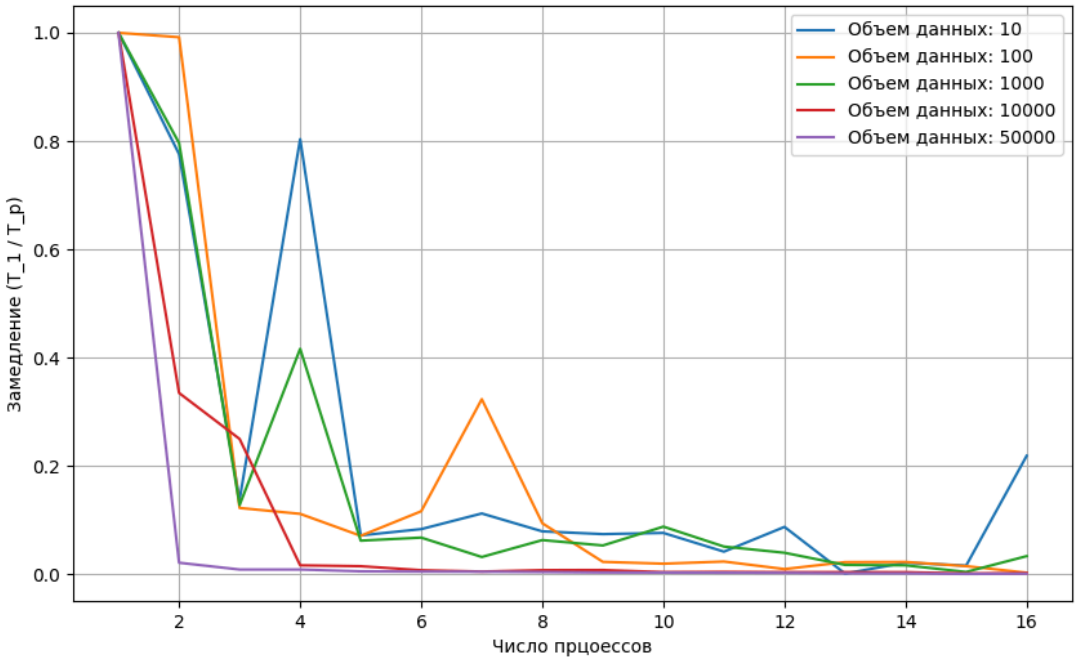

Рисунок 2 – График замедления работы программы в зависимости от количества процессов и объема входных данных

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.

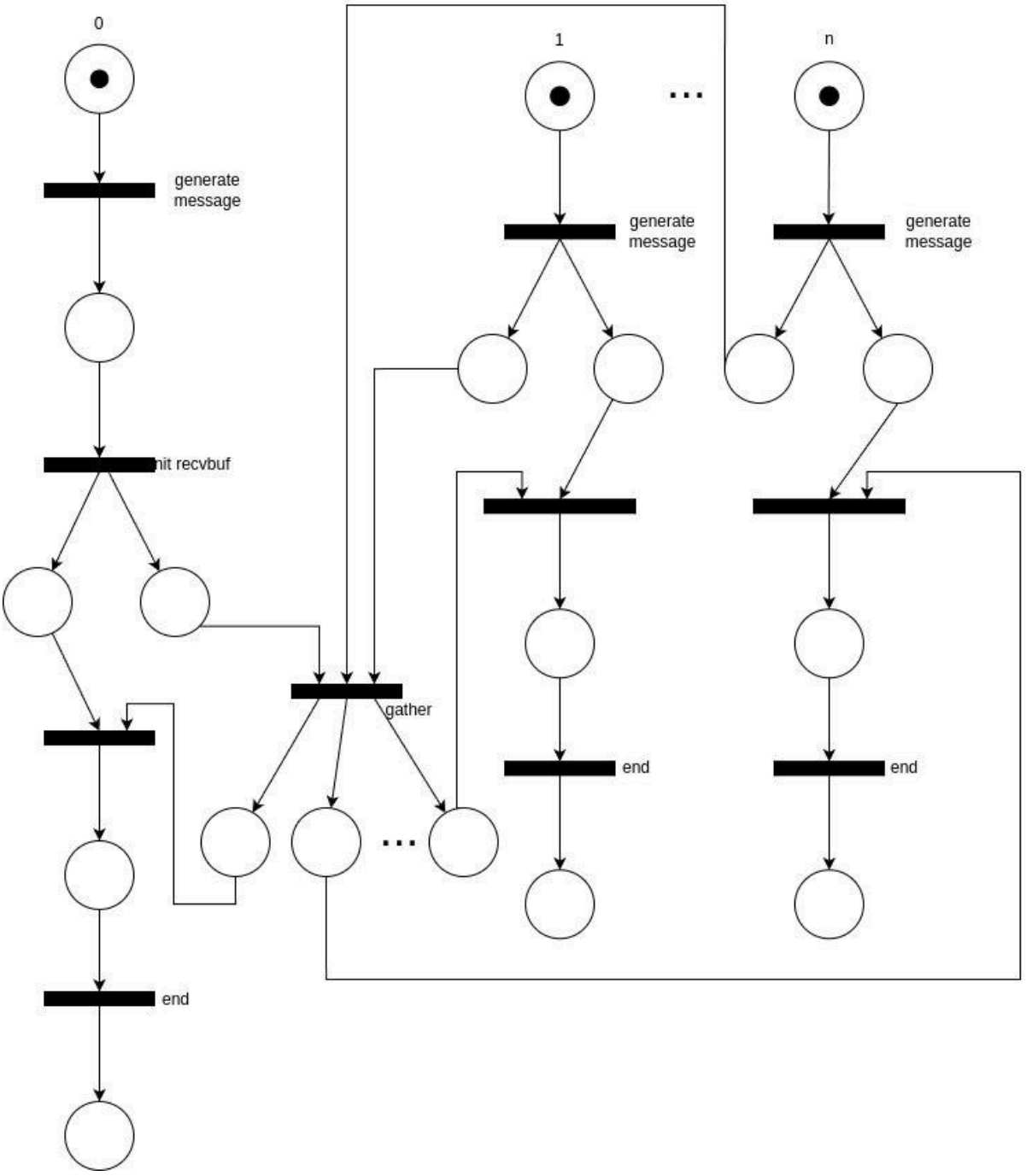

Рисунок 3 – Сети Петри

Вывод

В ходе выполнения лабораторной работы была написана программа, использующая функцию MPI_Gather для пересылки сообщений в главный процесс и выводящая их в порядке возрастания рангов переславших их процессов. Перед этим была изучена функция MPI_Gather, которая как раз и служит для передачи данных от всех процессов одному.

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.