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

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Параллельные алгоритмы»
Тема: Группы процессов и коммуникаторы

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2024

Цель работы

Создать программу, где будет создаваться новый коммуникатор и использоваться одна коллективная операция редукции для поиска минимальных значений среди элементов наборов с одним и тем же порядковым номером.

Задание

Вариант 4

В каждом процессе четного ранга (включая главный процесс) дан набор из трех элементов — вещественных чисел. Используя новый коммуникатор и одну коллективную операцию редукции, найти минимальные значения среди элементов исходных наборов с одним и тем же порядковым номером и вывести найденные минимумы в главном процессе. Новый коммуникатор создать с помощью функции MPI_Comm_split.

Указание. При вызове функции MPI_Comm_split в процессах, которые не требуется включать в новый коммуникатор, в качестве параметра color следует указывать константу MPI_UNDEFINED.

Выполнение работы

В ходе лабораторной работы была разработана программа на языке C++ с использованием библиотеки MPI. Функции данной библиотеки позволяют создать новый коммуникатор с помощью функции MPI_Comm_split.

MPI_Comm_split - расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по одной на каждое значение признака подгруппы color:
int MPI_Comm_split(MPI_Comm oldcomm, int color, int key, MPI_Comm *newcomm),

- oldcomm – исходный коммуникатор,
- color – номер коммуникатора, которому должен принадлежать процесс,
- key – порядок ранга процесса в создаваемом коммуникаторе,
- newcomm – создаваемый коммуникатор.

Приведем подробное описание программы.

Программа начинается с инициализации MPI. После этого идет получение ранга текущего процесса и общего количества процессов в коммуникаторе MPI_COMM_WORLD. Затем создается новый коммуникатор even_comm для процессов с четным рангом, а процессы с нечетным рангом не включаются в новый коммуникатор: процессы с четным рангом получают цвет 0, а с нечетным — MPI_UNDEFINED. Создание нового коммуникатора происходит с помощью функции MPI_Comm_split(MPI_COMM_WORLD, color, rank, &even_comm), которая была подробно описана ранее.

Каждый процесс генерирует 3 случайных числа в диапазоне от 0.0 до 100.0. Используется уникальный seed для каждого процесса, чтобы гарантировать различные данные.

После генерации чисел происходит редукция для нахождения минимальных значений среди всех процессов в новом коммуникаторе. “Все – одному” - операция редукции данных.

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm),

- sendbuf - буфер памяти с отправляемым сообщением,
- recvbuf – буфер памяти для результирующего сообщения,
- count - количество элементов в сообщениях,
- type – тип элементов сообщений,
- op - операция, которая должна быть выполнена над данными,
- root - ранг процесса, на котором должен быть получен результат,
- comm - коммуникатор, в рамках которого выполняется операция.

Рассмотрим, как это реализовано в написанной программе:

MPI_Reduce(local_data.data(), global_min.data(), 3, MPI_DOUBLE, MPI_MIN, 0, even_comm).

Таким образом можно увидеть, что редукция находит минимумы и отправляет результат в нулевой процесс, как и было сказано в задании.

Реализованная программа:

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

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //новый коммуникатор для процессов четного ранга
    int color = (rank % 2 == 0) ? 0 : MPI_UNDEFINED;
    MPI_Comm even_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &even_comm);

    if (color == 0) {
        //в коммуникаторе
        int even_rank;
        MPI_Comm_rank(even_comm, &even_rank);
```

```
        //генерация случайных данных
        std::random_device rd;
            std::mt19937 gen(rd() + rank); //уникальный seed для
каждого процесса
        std::uniform_real_distribution<> dis(0.0, 100.0);

        std::vector<double> local_data(3);
        for (auto& val : local_data) {
            val = dis(gen);
        }

        //вывод данных каждого процесса
        std::cout << "Process " << rank << " data: ";
        for (const auto& val : local_data) {
            std::cout << val << " ";
        }
        std::cout << std::endl;

        std::vector<double> global_min(3);

        //редукция для нахождения мин значений
              MPI_Reduce(local_data.data(),  global_min.data(),  3,
MPI_DOUBLE, MPI_MIN, 0, even_comm);

        if (even_rank == 0) {
            std::cout << "Global minimum values: ";
            for (const auto& val : global_min) {
                std::cout << val << " ";
            }
            std::cout << std::endl;
        }
        MPI_Comm_free(&even_comm);
    }

    MPI_Finalize();
    return 0;
}
```

Листинг программы:

```
katya@katya:~/ПА/lb5$ mpirun -np 4 ./lab5
Process 0 data: 3.88365 54.2966 38.3796
Process 2 data: 90.6841 13.9562 56.5127
Global minimum values: 3.88365 13.9562 38.3796
katya@katya:~/ПА/lb5$ mpirun -np 7 ./lab5
Process 0 data: 95.7703 2.29025 84.4675
Process 2 data: 11.0432 15.1506 95.8285
Process 4 data: 63.9715 48.5947 67.5411
Process 6 data: 23.1873 10.8353 79.6846
Global minimum values: 11.0432 2.29025 67.5411
katya@katya:~/ПА/lb5$ mpirun -np 16 ./lab5
Process 0 data: 18.106 18.9302 64.7665
Process 2 data: 61.6477 99.3725 94.5324
Process 8 data: 88.1007 95.5149 88.837
```

```
Process 10 data: 65.4908 27.5506 0.101471
Process 6 data: 6.77857 19.4608 63.4997
Process 12 data: 13.3592 26.4841 39.3829
Process 14 data: 72.373 23.246 82.112
Global minimum values: 6.77857 18.9302 0.101471
Process 4 data: 81.7388 18.943 17.3126
```

Видно, что программа отработала корректно: выведены минимумы из наборов.

Измененная программа для измерения времени:

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

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]);

    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //новый коммуникатор для процессов четного ранга
    int color = (rank % 2 == 0) ? 0 : MPI_UNDEFINED;

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

    MPI_Comm even_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &even_comm);

    if (color == 0) {
        //в коммуникаторе
        int even_rank;
        MPI_Comm_rank(even_comm, &even_rank);

                  std::vector<float>  local_data(data_size,  0.0),
global_min(data_size, 0.0);

        //редукция для нахождения мин значений
                 MPI_Reduce(local_data.data(),  global_min.data(),
data_size, MPI_FLOAT, MPI_MIN, 0, even_comm);

        if (even_rank == 0) {
            end = MPI_Wtime();
            time = end - start;
```

```
                fout << "Data size: " << data_size << " Time: " <<
time << " Processes: " << size << '\n';
            fout.close();
        }
        MPI_Comm_free(&even_comm);
    }

    MPI_Finalize();
    return 0;
}
```

Листинг программы с замерами времени:

```
Data size: 10 Time: 1.5279e-05 Processes: 1
Data size: 10 Time: 1.8105e-05 Processes: 2
Data size: 10 Time: 0.00017335 Processes: 3
Data size: 10 Time: 0.000187457 Processes: 4
Data size: 10 Time: 0.000237633 Processes: 5
Data size: 10 Time: 0.000463493 Processes: 6
Data size: 10 Time: 0.000306183 Processes: 7
Data size: 10 Time: 0.000269243 Processes: 8
Data size: 10 Time: 0.000389211 Processes: 9
Data size: 10 Time: 0.000371056 Processes: 10
Data size: 10 Time: 0.000483811 Processes: 11
Data size: 10 Time: 0.000421753 Processes: 12
Data size: 10 Time: 0.0121547 Processes: 13
Data size: 10 Time: 0.0238372 Processes: 14
Data size: 10 Time: 0.0320848 Processes: 15
Data size: 10 Time: 0.0230383 Processes: 16
Data size: 100 Time: 2.5108e-05 Processes: 1
Data size: 100 Time: 1.9587e-05 Processes: 2
Data size: 100 Time: 0.000266147 Processes: 3
Data size: 100 Time: 0.000175905 Processes: 4
Data size: 100 Time: 0.000268661 Processes: 5
Data size: 100 Time: 0.00022065 Processes: 6
Data size: 100 Time: 0.000445237 Processes: 7
Data size: 100 Time: 0.000137712 Processes: 8
Data size: 100 Time: 0.000343123 Processes: 9
Data size: 100 Time: 0.00036242 Processes: 10
Data size: 100 Time: 0.00037856 Processes: 11
Data size: 100 Time: 0.000443013 Processes: 12
Data size: 100 Time: 0.00724152 Processes: 13
Data size: 100 Time: 0.011426 Processes: 14
Data size: 100 Time: 0.0319629 Processes: 15
Data size: 100 Time: 0.0255522 Processes: 16
Data size: 1000 Time: 1.8365e-05 Processes: 1
Data size: 1000 Time: 2.3374e-05 Processes: 2
Data size: 1000 Time: 0.000214217 Processes: 3
Data size: 1000 Time: 0.000115199 Processes: 4
Data size: 1000 Time: 0.000397356 Processes: 5
Data size: 1000 Time: 0.000288169 Processes: 6
Data size: 1000 Time: 0.000393398 Processes: 7
Data size: 1000 Time: 0.000126271 Processes: 8
Data size: 1000 Time: 0.000431581 Processes: 9
Data size: 1000 Time: 0.000419659 Processes: 10
```

```
Data size: 1000 Time: 0.000618727 Processes: 11
Data size: 1000 Time: 0.000385403 Processes: 12
Data size: 1000 Time: 0.012193 Processes: 13
Data size: 1000 Time: 0.0162218 Processes: 14
Data size: 1000 Time: 0.0196383 Processes: 15
Data size: 1000 Time: 0.0186078 Processes: 16
Data size: 10000 Time: 4.4425e-05 Processes: 1
Data size: 10000 Time: 0.000111762 Processes: 2
Data size: 10000 Time: 0.000316271 Processes: 3
Data size: 10000 Time: 0.000199329 Processes: 4
Data size: 10000 Time: 0.000528205 Processes: 5
Data size: 10000 Time: 0.000537021 Processes: 6
Data size: 10000 Time: 0.000654075 Processes: 7
Data size: 10000 Time: 0.000281646 Processes: 8
Data size: 10000 Time: 0.000738605 Processes: 9
Data size: 10000 Time: 0.000589822 Processes: 10
Data size: 10000 Time: 0.000660957 Processes: 11
Data size: 10000 Time: 0.000607035 Processes: 12
Data size: 10000 Time: 0.0223777 Processes: 13
Data size: 10000 Time: 0.0337282 Processes: 14
Data size: 10000 Time: 0.0312825 Processes: 15
Data size: 10000 Time: 0.0208263 Processes: 16
```

Полученный график отображен на рисунке 1.

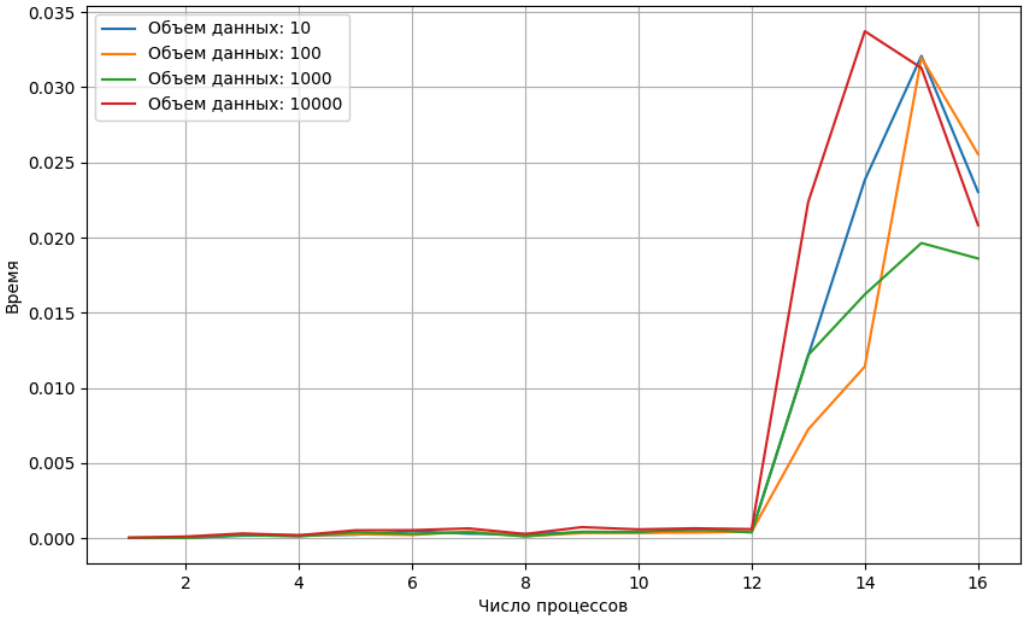

Рисунок 1 – График зависимости времени работы программы от числа запущенных процессов и от объема входных данных

На графике явно видно, что время увеличивается при увеличении количества процессов. Объяснить это можно так: увеличивается

количество процессов, значит увеличивается и количество процессов в группе, и количество наборов данных, которые нужно сравнить. Также видно, что при увеличении объемов данных увеличивается время работы команды. Резкие скачки времени (например, при объеме данных 10 и количестве процессов 15) 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:

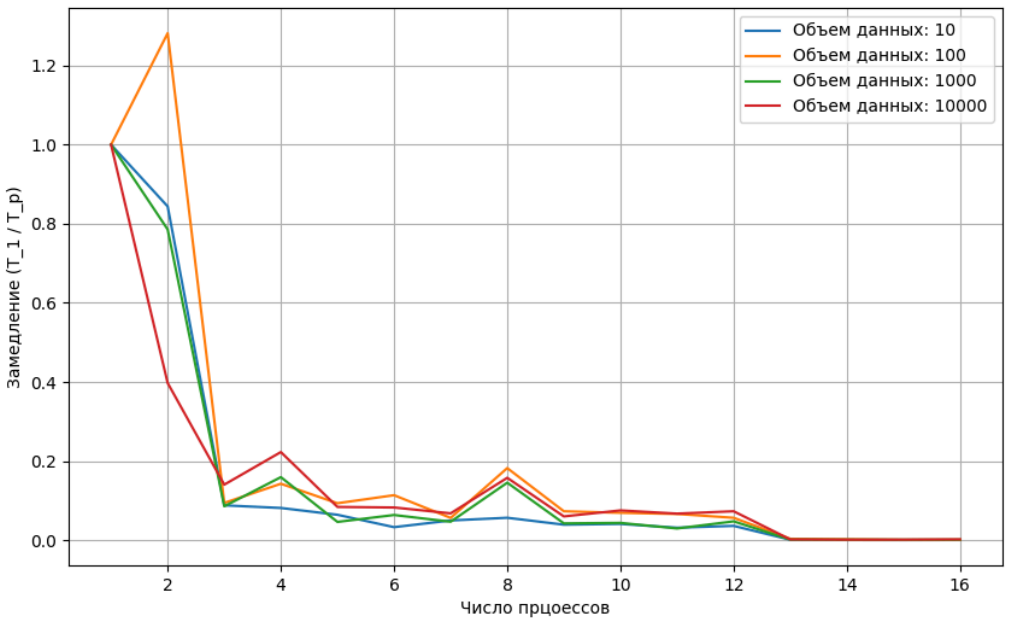

Рисунок 2 – График замедления работы программы в зависимости от количества процессов и объема входных данных

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.

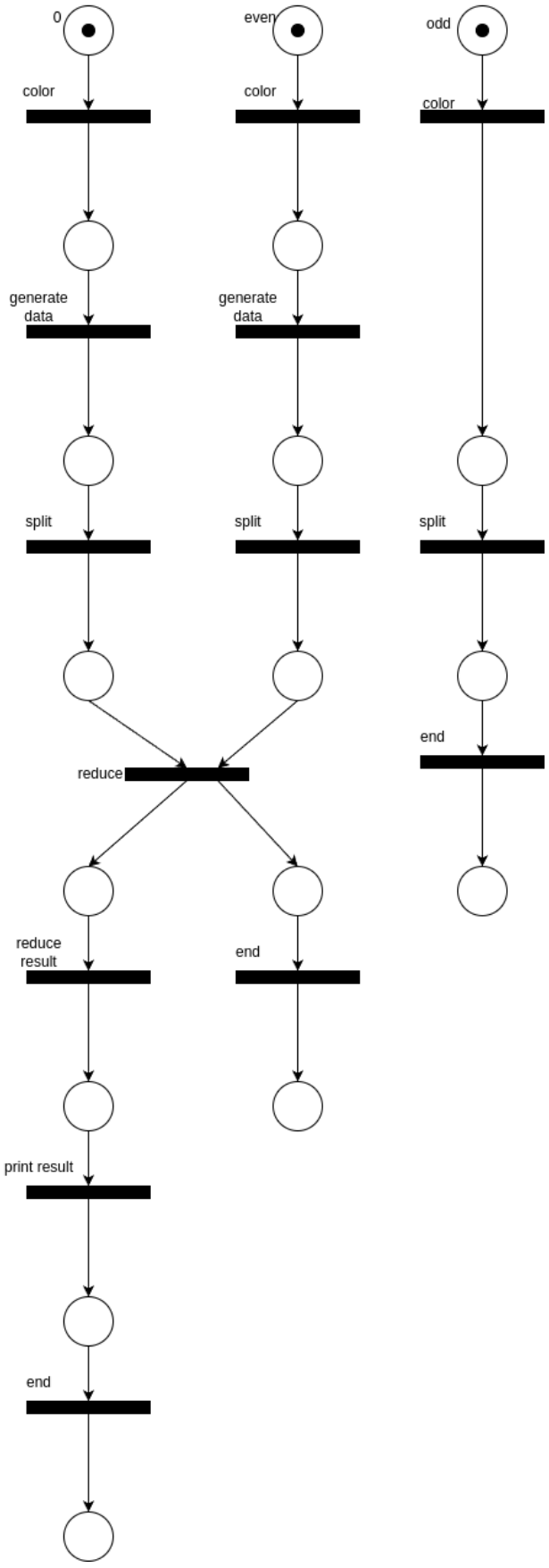

Рисунок 3 – Сети Петри

Вывод

В ходе выполнения лабораторной работы была написана программа, использующая функцию MPI_Comm_split, которая производит расщепление процессов на четные и нечетные и позволяет дальше работать только с четными. Также в написанной программе была использована функция MPI_Reduce, которая позволяет собрать данные с нескольких процессов, произвести над ними какую-то операцию (в нашем случае это нахождение минимума) и отправить результат на определенный процесс (нулевой по условию).

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.