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: Коллективные операции

Студент гр. 3381 _____ Иванов А.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург
2025

Цель работы.

Изучить основы коллективных операций в OpenMPI и написать программу в соответствии с заданием.

Задание.

Вариант 9

В каждом процессе дан набор из K + 5 целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_MAXLOC, найти максимальное значение среди элементов данных наборов с одним и тем же порядковым номером и ранг процесса, содержащего это максимальное значение. Вывести в главном процессе вначале все максимумы, а затем — ранги содержащих их процессов.

Выполнение работы.

Код программы можно условно разделить на следующие функциональные блоки:

1. Получение ранга процесса с помощью MPI_Comm_Rank
2. Получение количества параллельных процессов в коммуникаторе с помощью MPI_Comm_size
3. Заполнения массива array: динамически выделяется память для массива array с размером array_size = world_size + 5, который далее заполняется случайными знаечениями с помощью функции fill_array. Эта функция в качестве зерна для генерации случайного значения использует значение текущего времени time(NULL) с добавлением ранга процесса. Это позволяет получать различные сгенерированные значения в каждом из процессов. Внутри функции fill_array также выполняется деление по модулю 1000, чтобы ограничить диапазон значений и сделать вывод более удобочитаемым.

4. Блок условной компиляции: после заполнения массива его значения могут быть выведены в блоке условной компиляции, если была определена директива DEBUG и не определена директива MEASURE_TIME.
5. Создание и заполнение буфера для рассылки массива: создаётся буфер sendbuf размера array_size * 2, который содержит пары значение массива — ранг процесса, содержащего этот элемент массива.
6. Создание массива recvbuf для получения «сокращенных» с помощью MPI_Reduce значений: сначала определяется указатель recvbuf со значением NULL. Только в главном процессе в этот указатель записывается выделенный кусок памяти. Эта область памяти после выполнения MPI_Reduce будет содержать пары: максимальное значение — ранга процесса, содержащего это значение.
7. Вызов MPI_Reduce.
8. Вывод записанных в recvbuf значений: сначала выводятся максимальные значения среди одних и тех же номеров элементов в массивах каждого процесса, а после них сооветствующие ранги процессов. Данный блок отрабатывает только если определена директива прекомпилятора MEASURE_TIME.
9. Блок очистки памяти: метка содержащая вызовы функции free для каждого указателя, содержащего выделенную на куче область памяти. Переход к этой метке осуществляется при каждой возникающей ошибке в блоке обработки ошибок для функции malloc.

Сеть Петри, примеры запусков и листинг программы представлены в приложениях.

Тестирование.

Написанная программа была запущена на 2, 4, 8, 16, 32 процессах. Количество запусков для каждого количества процессов равно 50, для того чтобы статистически усреднить результат. Результаты приведены в таблице 1.

Таблица 1 — Результаты замеров

Количество процессов	Среднее время (100 запусков)	Ускорение
1	0.000016	-0.000016
2	0.000052	-0.000039
4	0.000091	-0.000039
8	0.000189	-0.000098
16	0.000205	-.0.000016
32	0.000291	-0.000086
64	0.000681	-0.000390

График ускорения представлен в таблице 1.

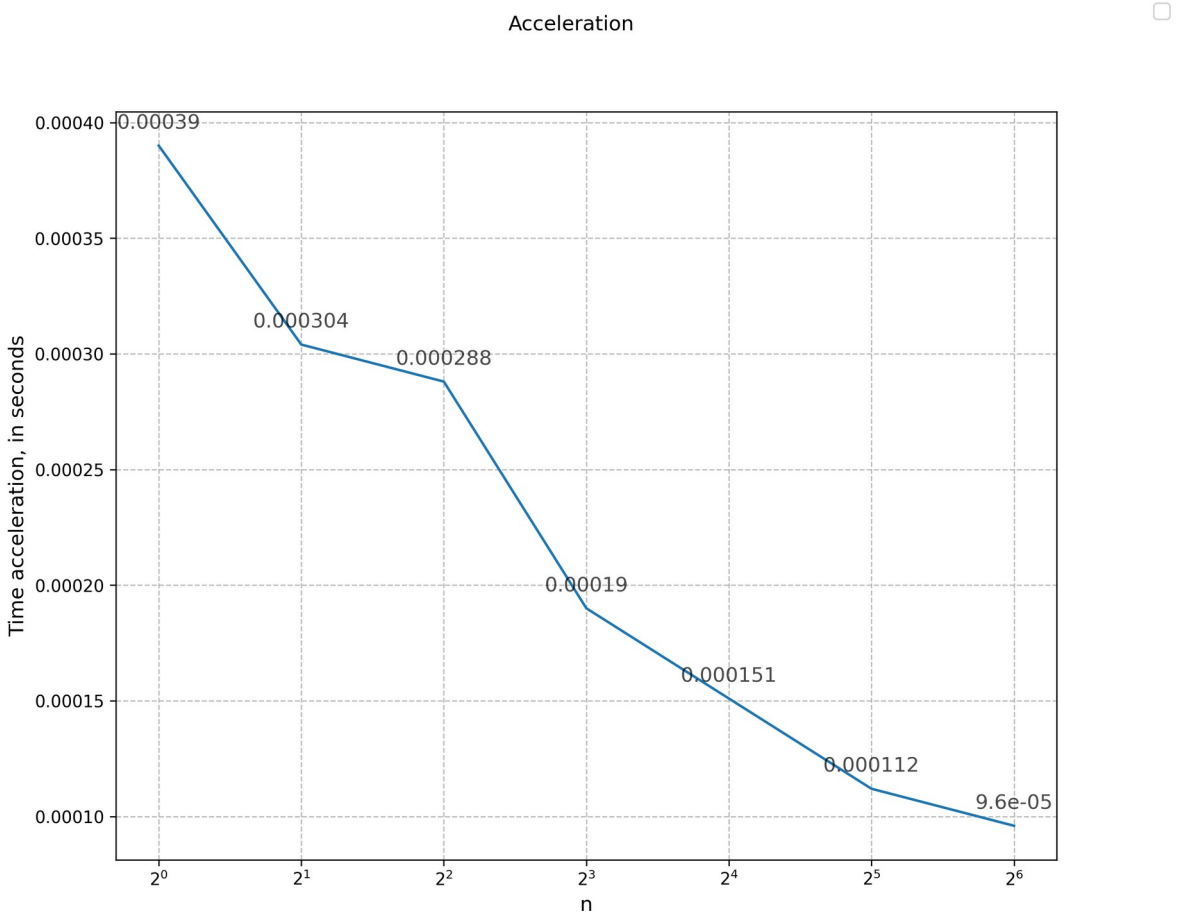

Рисунок 1 — График ускорения

График зависимости времени от количества процессов представлен на рисунке 2.

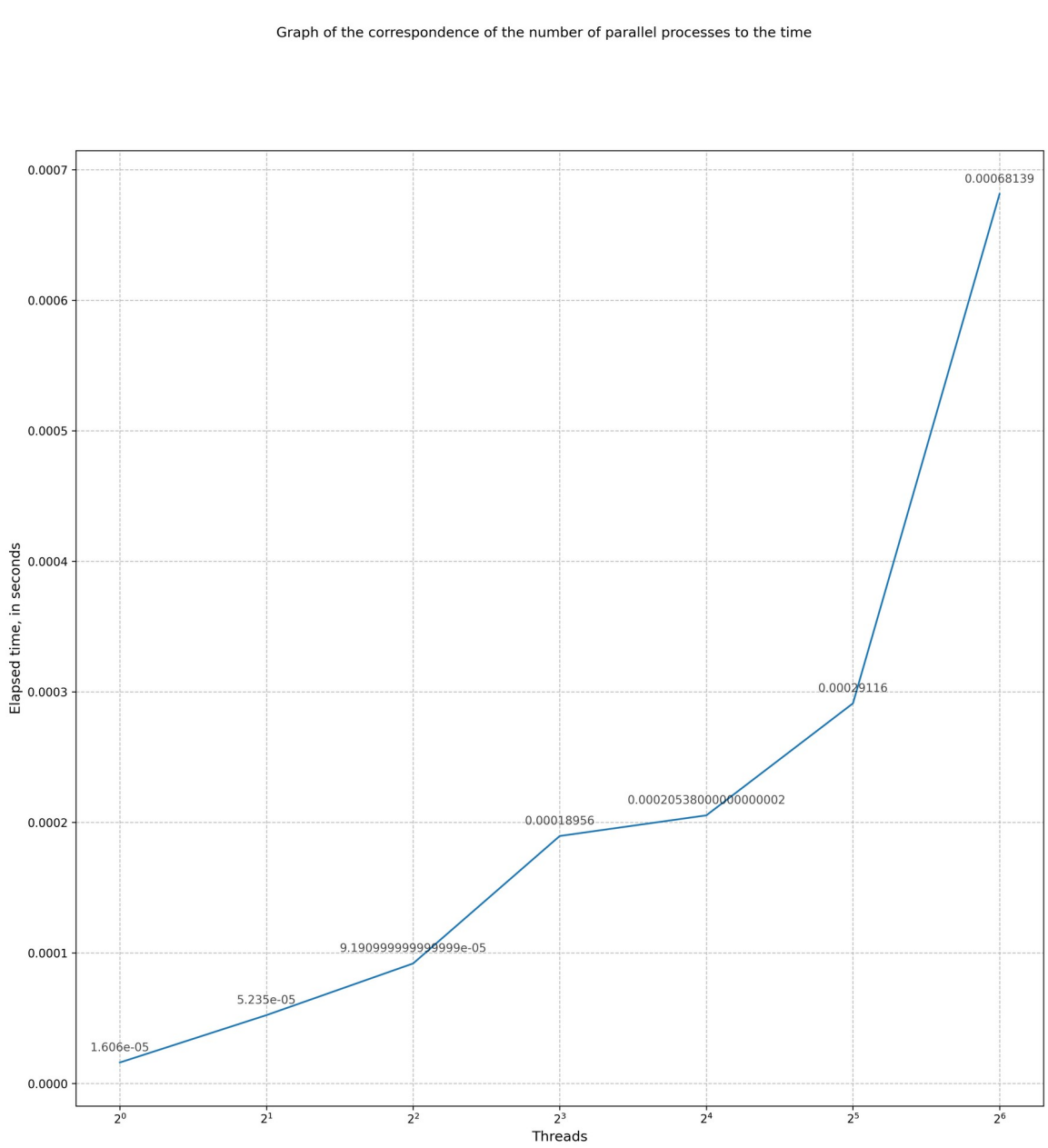

Рисунок 2 — График зависимости времени от количества параллельных процессов

Такой почти линейный график (от 32 до 64 взлёт кажется существенным, но это потому что разрыв в количестве процессов становится довольно существенен) времени ожидаем, так как в данной задаче не происходит никакого никакого распараллеливания единой задачи. Количество элементов массива линейно зависит от количества запущенных параллельных процессов.

Больше процессов — больше размер массива и соответственно большему числу процессов надо выполниться. То есть каждый раз время должно возрастать на условную величину $\Delta x \Delta y$, где Δx — изменение среднего времени выполнения отдельного процесса, Δy — изменение количества процессов. Значение Δx константно, так как размер массива растёт линейно в зависимости от общего количества процессов. То есть в идеальных условиях зависимость должна выглядеть примерно так: $y = kx$, где x — количество процессов. Но опять же стоит учитывать накладные расходы ОС на смену контекста — с увеличением количества процессов они тоже будут возрастать и не факт что линейно.

Выводы.

В ходе лабораторной работы был получен практический опыт работы с коллективными операциями в OpenMPI. Была разработана программа, которая генерирует массивы со случайными значениями в каждом процессе и находит максимумы по соответствующим элементам этих массивов с помощью коллективной операции MPI_Reduce.

ПРИЛОЖЕНИЕ А
СЕТЬ ПЕТРИ

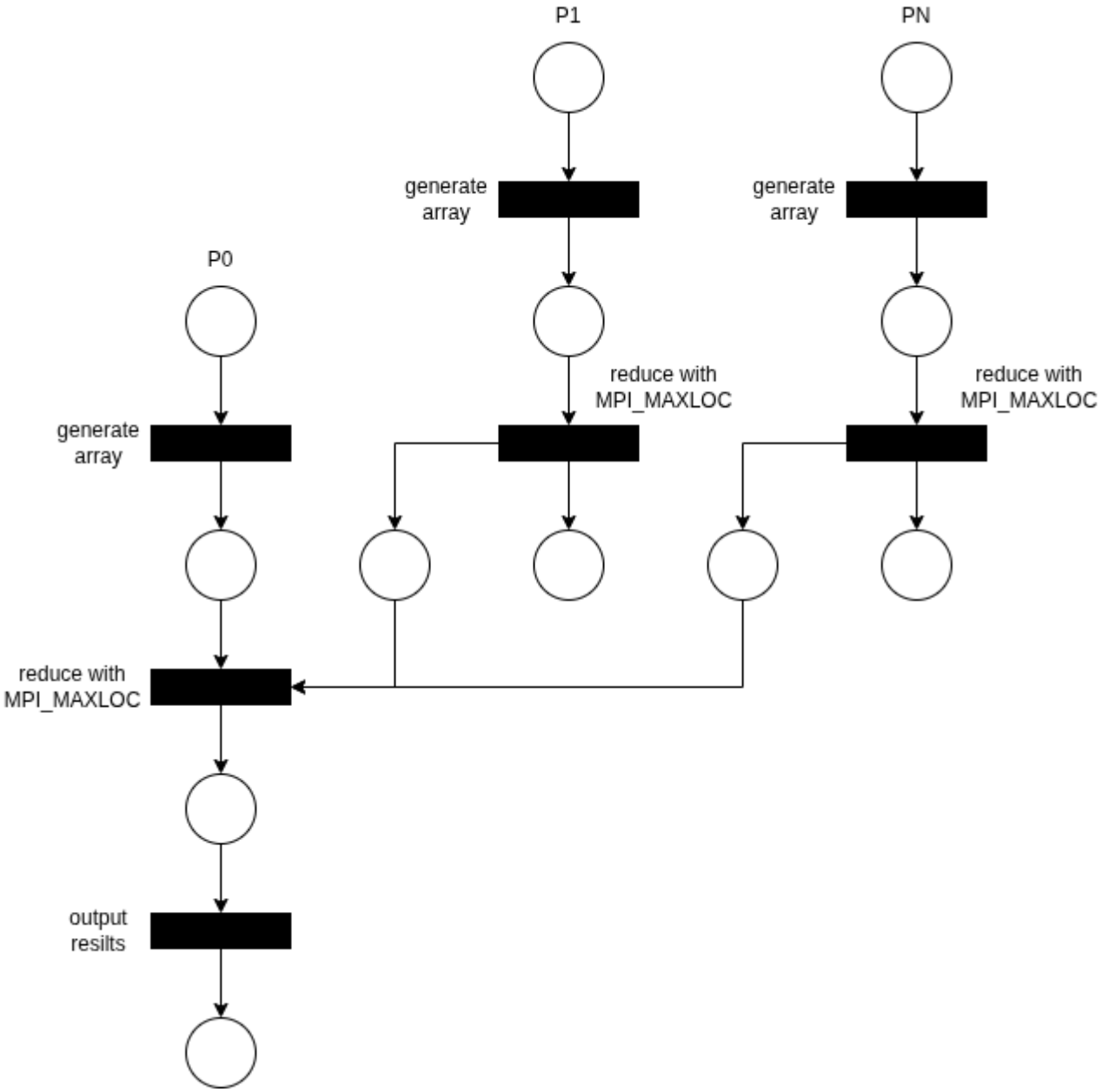

Рисунок А.1 — Сеть Петри

ПРИЛОЖЕНИЕ В
ПРИМЕРЫ РАБОТЫ АЛГОРИТМА

```
# Compiled with DEBUG flag
→ mpicc main.c -o main && valgrind mpirun -n 8 ./main
==186448== Memcheck, a memory error detector
==186448== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==186448== Using Valgrind-3.25.1 and LibVEX; rerun with -h for copyright info
==186448== Command: mpirun -n 8 ./main
==186448==
Rank 0 array: 559 455 952 769 813 167 999 779 228  53 568 275 351
Rank 3 array: 294 738  46 554 463 665 960 777 165 267 696 713 594
Rank 6 array: 649 790 674 512 492  72 148 178 969 704 622 541 804
Rank 1 array: 829 546  63  24 719 872 448 480 402 482  32 496 177
Rank 2 array: 397 148 331 902 206 974  14 556 364  83  39  39 928
Rank 7 array: 125 560 368 859 857 510  63  12 768 505 314 273 218
Rank 4 array: 538 551 265 877 640 245 313 649 977 903 518 760 331
Rank 5 array: 649 425 970 842 639 890 312 523 519  68 216 245 750
     MAX: 829 790 970 902 857 974 999 779 977 903 696 760 928
    RANK:   1   6   5   2   7   2   0   0   4   4   3   4   2

→ mpicc main.c -o main && mpirun -n 1 ./main
     MAX:  46 154 627 850 854 498
    RANK:   0   0   0   0   0   0

→ mpicc main.c -o main && mpirun -n 2 ./main
     MAX: 886 548 935 926 755  46 591
    RANK:   0   1   1   1   0   0   0

→ mpicc main.c -o main && mpirun -n 3 ./main
     MAX: 918 896 671 414 665 820 985 891
    RANK:   0   2   2   1   2   1   1   2

# Comiled with MEASURE_TIME flag
→ mpirun -n 8 ./main
0.000057
```

ПРИЛОЖЕНИЕ С
ЛИСТИНГ

Название файла: timer.c

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

#include "task.h"

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

    // Get rank
    int world_rank = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get amount of processes
    int world_size = 0;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int root_rank = 0;

    #ifdef MEASURE_TIME
    MPI_Barrier();
    double start_time, end_time;
    if (world_rank == 0) {
        start_time = MPI_Wtime();
    }
    #endif

    if (task(world_size, world_rank, root_rank)) {
        perror("Unable to recover from error in main task");
        goto finalize;
    }

    #ifdef MEASURE_TIME
    if (world_rank == 0) {
        end_time = MPI_Wtime();
        printf("%lf\n", end_time - start_time);
    }
    #endif

finalize:
    return MPI_Finalize();
}
```

Название файла: task.h

```
#ifndef TASK_H
#define TASK_H

int task(int world_size, int world_rank, int root_rank);

#endif
```

Название файла: task.c

```
#include <mpi.h>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


void fill_array(int* array, size_t array_size, unsigned int seed)
{
    srand(seed);

    const int upper_bound = 1000;
    for (size_t i = 0; i < array_size; ++i) {
        array[i] = rand() % upper_bound;
    }
}

int task(int world_size, int world_rank, int root_rank)
{
    int *array = NULL, *sendbuf = NULL, *recvbuf = NULL;

    // Create and fill array with random numbers
    const size_t array_size = world_size + 5;
    array = (int*)malloc(array_size * sizeof(int));
    if (!array) {
        perror("Can't allocate memory for array");
        goto free_memory;
    }
    fill_array(array, array_size, time(NULL) + world_rank);

    #if defined(DEBUG) && !defined(MEASURE_TIME)
    printf("Rank %d array: ", world_rank);
    for (size_t i = 0; i < array_size; ++i) {
        printf("%3d ", array[i]);
    }
    putchar('\n');
    #endif

    const size_t sendbuf_size = array_size * 2;
    sendbuf = (int*)malloc(sendbuf_size * sizeof(int));
    if (!sendbuf) {
        perror("Can't allocate memory for sendbuf");
        goto free_memory;
    }
    for (size_t i = 0; i < array_size; ++i) {
        sendbuf[i*2] = array[i];
        sendbuf[i*2 + 1] = world_rank;
    }

    // To get return value in form: result1, index1, result2, index2, ...
    const size_t recvbuf_size = sendbuf_size;
    if (world_rank == root_rank) {
        recvbuf = (int*)malloc(recvbuf_size * sizeof(int));
        if (!recvbuf) {
            perror("Can't allocate memory for recvbuf");
            goto free_memory;
        }
    }

    MPI_Reduce(sendbuf, recvbuf, array_size, MPI_2INT, MPI_MAXLOC, root_rank,
MPI_COMM_WORLD);
```

```
    #ifndef MEASURE_TIME
    if (world_rank == root_rank) {
        // Output all max values
        printf("%8s: ", "MAX");
        for (size_t i = 0; i < recvbuf_size; i+=2) {
            printf("%3d ", recvbuf[i]);
        }
        putchar('\n');

        // Output ranks of processes containing those max values
        printf("%8s: ", "RANK");
        for (size_t i = 1; i < recvbuf_size; i+=2) {
            printf("%3d ", recvbuf[i]);
        }
        putchar('\n');
    }
    #endif

free_memory:
    free(array);
    free(sendbuf);
    free(recvbuf);

    return 0;
}
```