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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «ПА»

Тема: ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ ОБМЕНА ДАННЫМИ «ТОЧКА - ТОЧКА» В БИБЛИОТЕКЕ MPI

Студент гр. 8304 _____ Сергеев А.Д.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2020

Задание.

Процесс 0 генерирует массив и раздает его другим процессам для вычисления локальных сумм, после чего вычисляет общую сумму.

Описание алгоритма.

Процесс с рангом 0 создаёт массив чисел с плавающей точкой, делит его на равные части между остальными процессами и рассылает им указатели на эти части указав в поле тэг свой ранг. После этого он создаёт массив локальных сумм и ожидает окончания работы остальных процессов для его заполнения, при чем, как только очередной процесс завершается, высчитанная им сумма заносится в элемент массива, номер которого равен рангу этого процесса. Сумма элементов массива локальных сумм и будет общей суммой элементов массива.

Каждый из процессов с рангом, отличным от нуля, создаёт массив, длина которого равна частному количества элементов массива, созданного нулевым процессом, и количеству ненулевых процессов. После этого он ожидает сообщение от нулевого процесса, заполняющего массив, вычисляет сумму его элементов и посылает сумму нулевому процессу, указывая в поле тэг свой ранг.

Листинг программы.

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

#define array_size 10000000
#define array_upper_limit 1000.0

void populate_array(double* arr, unsigned int size) {
srand(time(NULL));
printf("Array (length %d): [ ", size);
for (unsigned int i = 0; i < size; i++) {
arr[i] = (array_upper_limit / RAND_MAX) * rand();
//printf("%lf ", arr[i]);
}
printf("]\n");
}
```

```
double sum_array(const double* arr, unsigned int size) {
if (size == 0) return -1;
double max = arr[0];
for (unsigned int i = 0; i < size; i++) if (arr[i] > max) max = arr[i];
return max;
}

int main(int argc, char* argv[]) {
int ProcNum, ProcRank;
MPI_Status Status;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

int work_nodes = ProcNum - 1;
int node_quote = array_size / work_nodes;

if (ProcRank == 0) {
double start_time = MPI_Wtime();
printf ("Root process started at %lf\n", start_time);

double* array = (double*) malloc(array_size * sizeof(double));
populate_array(array, array_size);

for (int i = 1; i < ProcNum; i++)
MPI_Send(array + (i - 1) * node_quote, node_quote, MPI_DOUBLE, i,
ProcRank, MPI_COMM_WORLD);

double* local_sums = (double*) malloc(work_nodes * sizeof(double));
printf ("Local maxes received (one of each %d elements): [ ",
node_quote);
for (int i = 1; i < ProcNum; i++) {
MPI_Recv(&(local_sums[i - 1]), 1, MPI_DOUBLE, MPI_ANY_SOURCE, i,
MPI_COMM_WORLD, &Status);
printf("%lf ", local_sums[i - 1]);
}
printf("]\n");

free(array);
double max = sum_array(local_sums, ProcNum);
free(local_sums);
printf("Global array max: %lf\n", max);

double end_time = MPI_Wtime();
printf("Root process ended at %lf\n", end_time);
printf("All operations took %lf\n", end_time - start_time);

} else {
double* arr = (double*) malloc(node_quote * sizeof(double));
```

```
MPI_Recv(arr, node_quote, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
&Status);
double local_sum = sum_array(arr, node_quote);
MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, ProcRank, MPI_COMM_WORLD);
free(arr);
}

MPI_Finalize();
return 0;
}
```

Результаты работы.

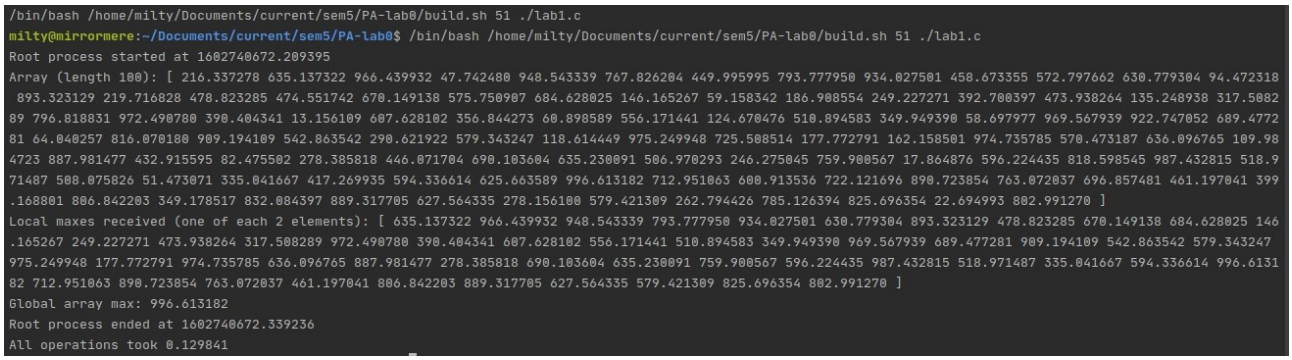

Рисунок 1 — Результат запуска программы для 51 процесса

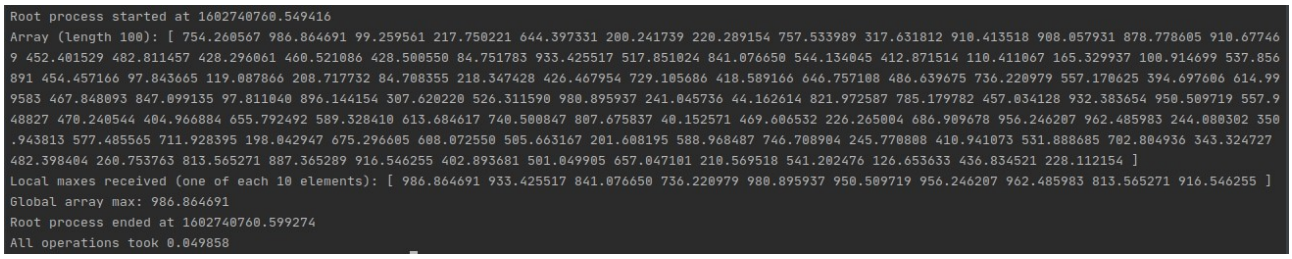

Рисунок 2 — Результат запуска программы для 11 процессов

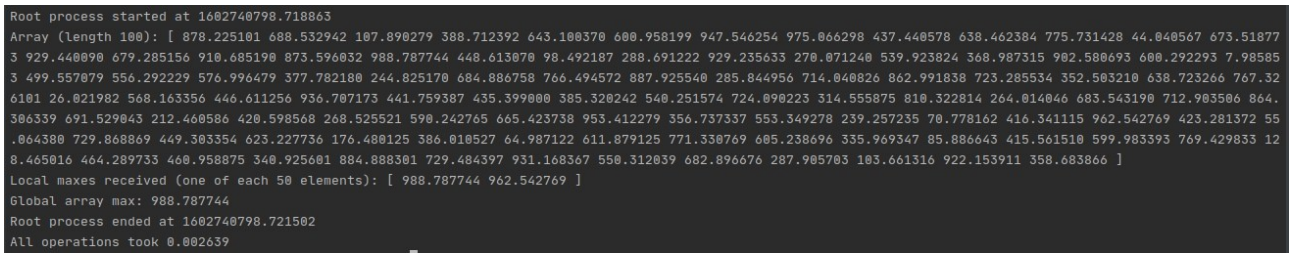

Рисунок 3 — Результат запуска программы для 3 процессов

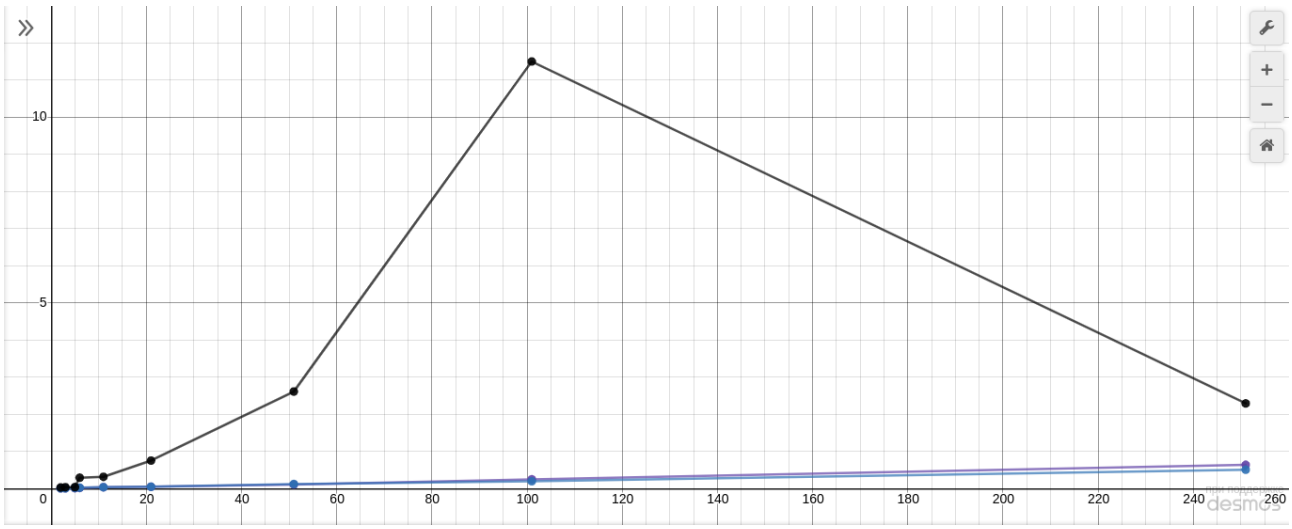

Рисунок 4 — Графики зависимости времени работы от количества процессов

Синий график — массив из 100 элементов.

Фиолетовый график — массив из 10000 элементов.

Черный график — массив из 1000000 элементов.

Выводы.

Была написана, в которой использовались параллельные вычисления MPI. Различия во времени выполнения программы для разного количества процессов обусловлены тем, что задача поиска суммы элементов массива сама по себе выполняется гораздо быстрее, чем передача сообщений между узлами. Разница во времени выполнения программы между случаями с одинаковым количеством процессом, но разными размерами массива всё же демонстрируют зависимость времени выполнения от количества элементов массива. Пик, наблюдаемый при выполнении на 100 процессах, по моему мнению, появляется из-за различных вариантов оптимизации передачи сообщений используемых для разных количеств процессов.