МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ   
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ   
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Отчёт по лабораторной работе №6  
по дисциплине «Параллельные вычисления»

**Основы MPI**

Выполнил: студент группы ФИб-4301-51-00     / К.О. Дёмин /

Проверил: ст.преподаватель каф. ПМИ     / А.В. Торбеева /

Киров 2019

Оглавление

[Цель работы 3](#_Toc26722481)

[Задание 3](#_Toc26722482)

[Листинг 5](#_Toc26722483)

[Полученные результаты 15](#_Toc26722484)

[Выводы по лабораторной работе 23](#_Toc26722485)

# Цель работы

Получить навыки разработки параллельных программ с использованием технологии MPI.

# Задание

1. Написать параллельную программу, вычисляющую ln 2 по формуле  (*N* ≤ 5∙109).   
   Реализовать а) с использованием парных операций, б) с использованием коллективных операций.

Замерить среднее время выполнения программы для *n* = 108, 5∙108, 109 (на каждое – не менее 3 запусков) для 1, 2, 4 и 8 процессов. Вычислить среднее ускорение для 2, 4 и 8 процессов. Рассчитать эффективность параллельного алгоритма. Построить графики ускорения и эффективности.

Сравнить результаты использования парных и коллективных операций.

1. Имеются два процесса. Первый процесс последовательно читает массивы из файла и отправляет данные массивы второму процессу. Каждый массив записан в файле в отдельной строке, элементы – целые числа, разделенные пробелами. Второй процесс получает массив, вычисляет сумму его элементов и отправляет первому полученный результат.   
   Написать параллельную программу, реализующую работу этих процессов. Суммы элементов массивов должны быть занесены в итоговый файл, каждый элемент – в отдельной строке.
2. ***Концепция master-slave***. Имеется *P* + 1 процесс (*P* задается пользователем). У 0 процесса имеется массив размера *N* (число *N* задается пользователем, *N* ≤ 5∙108, элементы массива – числа от *N* до 1). Ему нужно вычислить сумму элементов массива. Нулевой процесс может раздавать задания другим процессам (остальные процессы – рабочие). Написать параллельную программу, реализующую работу этих процессов.   
   Способы раздачи:   
   1) всем поровну;   
   2) всем одинаковые порции, потом по мере освобождения;   
   3) сначала всем одинаковые порции, потом размер порции каждый раз уменьшается в два раза.
3. Написать параллельную программу, проверяющую, является ли заданный массив размера *N* (*N* ≤ 5∙108) упорядоченным.

Замерить среднее время выполнения программы для *N* = 108, 5∙108 (на каждое – не менее 3 запусков) для 1, 2, 4 и 8 процессов. Вычислить среднее ускорение для 2, 4 и 8 процессов. Рассчитать эффективность параллельного алгоритма. Построить графики ускорения и эффективности.

1. Написать параллельную программу, вычисляющую сумму двух векторов размера *N* (*N* ≤ 5∙108).

Замерить среднее время выполнения программы для *N* = 108, 5∙108 (на каждое – не менее 3 запусков) для 1, 2, 4 и 8 процессов. Вычислить среднее ускорение для 2, 4 и 8 процессов. Рассчитать эффективность параллельного алгоритма. Построить графики ускорения и эффективности.

# Листинг

Задание 1

#include <iostream>

#include <math.h>

#include <mpi.h>

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

long long N;

double sum = 0;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

double start, end;

if (proc\_rank == 0)

{

//  вводим n и рассылаем всем процессам

std::cout << "input N: ";

std::cin >> N;

start = MPI\_Wtime();

for (int i = 1; i < proc\_num; ++i)

MPI\_Send(&N, 1, MPI\_LONG\_LONG, i, 0, MPI\_COMM\_WORLD);

}

else

{

MPI\_Recv(&N, 1, MPI\_LONG\_LONG, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &st);

}

//  считаем значение

for (long long k = proc\_rank + 1; k <= N; k += proc\_num)

sum += (((k - 1) & 1) == 1 ? -1.0 : 1.0) / k;

//  отправляем главному процессу

if (proc\_rank == 0)

{

double tmpsum;

//  собираем значения других процессов и вычисляем результат

for (int i = 1; i < proc\_num; ++i)

{

MPI\_Recv(&tmpsum, 1, MPI\_LONG\_LONG, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &st);

sum += tmpsum;

}

end = MPI\_Wtime();

std::cout << "time to sum = " << end - start << "s\n";

printf("Result: %.10f", sum);

}

else

{

MPI\_Send(&sum, 1, MPI\_LONG\_LONG, 0, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <chrono>

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

long long N;

double sum = 0;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

double start, end;

if (proc\_rank == 0)

{

//  вводим n

std::cout << "input N: ";

std::cin >> N;

start = MPI\_Wtime();

}

//  рассылаем всем N

MPI\_Bcast(&N, 1, MPI\_LONG\_LONG, 0, MPI\_COMM\_WORLD);

//  считаем сумму

double tmpsum = 0;

for (long long k = proc\_rank + 1; k <= N; k += proc\_num)

tmpsum += (((k - 1) & 1) == 1 ? -1.0 : 1.0) / k;

//  собираем значения в сумме и выводим результаты

MPI\_Reduce(&tmpsum, &sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (proc\_rank == 0)

{

end = MPI\_Wtime();

std::cout << "time to sum = " << end - start << std::endl;

printf("Result: %.10f", sum);

}

MPI\_Finalize();

return 0;

}

Задание 2

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <fstream>

#include <string>

#include <vector>

#include <iterator>

#include <sstream>

#include <algorithm>

#include "task2.h"

const int firstTag = 1;

const int secondTag = 2;

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

long long N;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

if (proc\_rank == 0)

{

std::string line;

std::ifstream in("D:/YandexDisk/Универ/4 Курс/Параллельное программирование/lab 6/x64/Debug/input.txt"); // окрываем файл для чтения

std::ofstream out("D:/YandexDisk/Универ/4 Курс/Параллельное программирование/lab 6/x64/Debug/output.txt");  //  открывает файл для записи

if (in.is\_open())

{

//  читаем строку

while (getline(in, line))

{

//  выводим считанные данные

std::cout << line << std::endl;

std::istringstream is(line);

std::vector<std::string> tokens;

//  делим строку

std::copy(std::istream\_iterator<std::string>(is),

std::istream\_iterator<std::string>(),

std::back\_inserter(tokens));

//  парсим числа в массив

int\* arr = new int[tokens.size()];

for (int i = 0; i < tokens.size(); ++i)

arr[i] = std::atoi(tokens.at(i).c\_str());

//  отправляем числа второму процессу

int size = tokens.size();

MPI\_Send(arr, size, MPI\_INT, 1, firstTag, MPI\_COMM\_WORLD);

//  ожидаем ответ от второго процесса

MPI\_Probe(1, secondTag, MPI\_COMM\_WORLD, &st);

MPI\_Recv(&N, 1, MPI\_LONG\_LONG, 1, secondTag, MPI\_COMM\_WORLD, &st);

//  записываем ответ в файл

out << N << "\n";

delete[] arr;

}

}

in.close();     // закрываем файлы

out.close();

int end = INT\_MAX;

//  говорим о завершении

MPI\_Send(&end, 1, MPI\_INT, 1, firstTag, MPI\_COMM\_WORLD);

}

else if (proc\_rank == 1)

{

//  ожидаем массивы

while (true)

{

int s = MPI\_Probe(0, firstTag, MPI\_COMM\_WORLD, &st);

int size;

//  парсим размер

MPI\_Get\_count(&st, MPI\_INT, &size);

//  std::cout << size << std::endl;

int\* arr = new int[size];

MPI\_Recv(arr, size, MPI\_INT, 0, firstTag, MPI\_COMM\_WORLD, &st);

// std::cout << "ok" << std::endl;

//  если сигнал конца, то завершаем процесс

if (size == 1 && arr[0] == INT\_MAX) break;

long long res = 0;

//  считаем сумму

for (int i = 0; i < size; ++i)

res += arr[i];

delete[] arr;

//  отправляем результат

MPI\_Send(&res, 1, MPI\_LONG\_LONG, 0, secondTag, MPI\_COMM\_WORLD);

}

}

MPI\_Finalize();

return 0;

}

Задание 3

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <string>

#include <algorithm>

const int answerTag = 0;

const int feedbackTag = 1;

//  копирование части массива

int\* getPart(int\* arr, int start, int count)

{

int\* result = new int[count];

for (int i = 0; i < count; ++i)

result[i] = arr[start + i];

return result;

}

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

int plan;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

if (proc\_rank == 0)  //  master

{

//  читаем n

int n;

std::cout << "Input N" << std::endl;

std::cin >> n;

std::cout << "Choose your destiny:" << std::endl;

//  считываем план работы и отправляем его рабам

std::cin >> plan;

for (int i = 1; i < proc\_num; ++i)

MPI\_Send(&plan, 1, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

int\* arr = new int[n];

for (int i = 0; i < n; ++i)

arr[i] = i + 1;

//  если всем поровну, то делим всем и отправляем

if (plan == 1)

{

int remainder = n;

int start = 0;

for (int i = 1; i < proc\_num; ++i)

{

int s = ceil(remainder / (double)(proc\_num - i));

remainder -= s;

int\* part = getPart(arr, start, s);

MPI\_Send(part, s, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

delete[] part;

start += s;

}

//  считываем результаты

long long result = 0;

for (int i = 1; i < proc\_num; ++i)

{

long long temp;

MPI\_Recv(&temp, 1, MPI\_LONG\_LONG, i, feedbackTag, MPI\_COMM\_WORLD, &st);

result += temp;

}

std::cout << "Master: " << result << std::endl;

}

else

{

//  если всем одинаковые порции, а затем по мере освобождения

int perProcess = (n / (proc\_num - 1));

std::cout << "PER PROCESS: " << perProcess << std::endl;

for (int i = 1; i < proc\_num; ++i)

{

int start = (i - 1) \* perProcess;

int\* part = getPart(arr, start, perProcess);

MPI\_Send(part, perProcess, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

delete[] part;

}

long long result = 0;

int remainder = n - (proc\_num - 1) \* perProcess;

int start = (proc\_num - 1) \* perProcess;

//  если всем поровну, то делим остаток на все процессы. Если каждый раз уменьшать в 2 раза, то делим остаток на половину

if (plan == 3 && remainder > 2)

perProcess = remainder / 2;

else

perProcess = ceil(remainder / (double)(proc\_num - 1));

int dop = 0;

std::cout << "Remainder: " << remainder << std::endl;

//  получаем значения от других процессов и отправляем свободным доп задания, пока остаток > 0

for (int i = 0; i < proc\_num - 1 + dop; ++i)

{

long long temp;

MPI\_Recv(&temp, 1, MPI\_LONG\_LONG, MPI\_ANY\_SOURCE, feedbackTag, MPI\_COMM\_WORLD, &st);

result += temp;

if (remainder > 0)

{

std::cout << "PP: " << perProcess << std::endl;

int s = remainder > perProcess ? perProcess : remainder;

remainder -= s;

int\* bfg = getPart(arr, start, s);

MPI\_Send(bfg, s, MPI\_INT, st.MPI\_SOURCE, answerTag, MPI\_COMM\_WORLD);

start += s;

delete[] bfg;

++dop;

if (plan == 3 && perProcess > 1)

perProcess /= 2;

}

}

std::cout << "Master: " << result << std::endl;

int exit = INT\_MAX;

for (int i = 1; i < proc\_num; ++i)

{

MPI\_Send(&exit, 1, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

}

}

delete[] arr;

}

else  //  slaves

{

//  получаем план

MPI\_Recv(&plan, 1, MPI\_INT, 0, answerTag, MPI\_COMM\_WORLD, &st);

//  если всем поровну, то просто получаем результат, вычисляем и отправляем результат. Иначе ждём сигнал конца

do

{

MPI\_Probe(0, answerTag, MPI\_COMM\_WORLD, &st);

int size;

MPI\_Get\_count(&st, MPI\_INT, &size);

int\* buff = new int[size];

MPI\_Recv(buff, size, MPI\_INT, 0, answerTag, MPI\_COMM\_WORLD, &st);

if (size == 1 && buff[0] == INT\_MAX) break;

std::cout << "Slave " << proc\_rank << " data input" << std::endl;

long long result = 0;

for (int i = 0; i < size; ++i)

result += buff[i];

delete[] buff;

MPI\_Send(&result, 1, MPI\_LONG\_LONG, 0, feedbackTag, MPI\_COMM\_WORLD);

std::cout << "Slave " << proc\_rank << " data output" << std::endl;

} while (plan != 1);

}

MPI\_Finalize();

return 0;

}

Задание 4

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <string>

#include <algorithm>

const int answerTag = 0;

const int feedbackTag = 1;

//  1 - по возрастанию, -1 - по убыванию, 0 - массив равен, INT\_MAX - разные

int checkArray(int\* arr, int size)

{

int result = 0;

for (int i = 1; i < size; ++i)

{

int temp = 0;

if (arr[i - 1] < arr[i])

temp = 1;

else if (arr[i - 1] > arr[i])

temp = -1;

if (result == 0)

result = temp;

else if (result != temp && temp != 0)

{

result = INT\_MAX;

break;

}

}

return result;

}

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

double start, end;

int n;

int\* arr = NULL;

if (proc\_rank == 0)

{

std::cout << "Input N" << std::endl;

std::cin >> n;

std::cout << "Choose your destiny:\n1. Random\n2. Ascending\n3. Descending" << std::endl;

int plan;

std::cin >> plan;

if (plan < 1 || plan > 3)

return 0;

//  заполняем массив так, как нам надо

arr = new int[n];

for (int i = 0; i < n; ++i)

{

if (plan == 1)

arr[i] = rand();

else if (plan == 2)

arr[i] = i;

else arr[i] = n - i;

}

start = MPI\_Wtime();  //  записываем время начала рассчета

for (int i = 1; i < proc\_num; ++i)  //  отправляем размер всем

MPI\_Send(&n, 1, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

}

else

{

//  принимаем размер n

MPI\_Recv(&n, 1, MPI\_INT, 0, answerTag, MPI\_COMM\_WORLD, &st);

}

int\* sendcounts = new int[proc\_num];  //  кол-во элементов

int\* displs = new int[proc\_num];  //  смещение

int remainder = n;  //  остаток

for (int i = 0; i < proc\_num; ++i)

{

displs[i] = i > 0 ? displs[i - 1] + sendcounts[i - 1] - 1 : 0;  //  смещение = смещение старого значения + кол-во элементов у пред. элемента - 1

sendcounts[i] = ceil(remainder / (double)(proc\_num - i)) + (i != proc\_num - 1 ? 1 : 0);  //  кол-во элементов = остаток / кол-во оставшихся процессов + (если процесс не последний, то добавляем границу следующего)

remainder = remainder - (sendcounts[i] - 1);  //  остаток = остаток - (кол-во элементов - 1)

}

//  буфер для приёма

int\* buff = new int[sendcounts[proc\_rank]];

//  распределяем между процессами

MPI\_Scatterv(arr, sendcounts, displs, MPI\_INT, buff, sendcounts[proc\_rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

//  получаем результат

int res = checkArray(buff, sendcounts[proc\_rank]);

//  если главный процесс и упорядочено, то получаем значения других процессов и смотрим результат

if (proc\_rank == 0)

{

if (res != INT\_MAX)

{

for (int i = 1; i < proc\_num; ++i)

{

int temp;

MPI\_Recv(&temp, 1, MPI\_INT, MPI\_ANY\_SOURCE, feedbackTag, MPI\_COMM\_WORLD, &st);

if (res == 0)

res = temp;

else if (temp != res && temp != 0)  //  если значения не совпадают, значит массив не упорядочен

{

res = INT\_MAX;

break;

}

}

}

end = MPI\_Wtime();

std::cout << "Time: " << end - start << std::endl;

std::cout << "Master: " << (res != INT\_MAX ? (res == 1 ? "Ascending" : "Descending") : "FALSE") << std::endl;

}

else

{

//  отправляем результат главному процессу

MPI\_Send(&res, 1, MPI\_INT, 0, feedbackTag, MPI\_COMM\_WORLD);

}

//  чистим память

delete[] sendcounts;

delete[] displs;

delete[] arr;

delete[] buff;

MPI\_Finalize();

return 0;

}

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <string>

#include <algorithm>

const int answerTag = 0;

const int feedbackTag = 1;

//  генерация массива

int\* genArray(int size)

{

int\* result = new int[size];

for (int i = 0; i < size; ++i)

result[i] = rand() % 100;

return result;

}

//  вывод массива

void print(int\* arr, int size)

{

if (size > 10) return;

for (int i = 0; i < size; ++i)

std::cout << arr[i] << "\t";

std::cout << std::endl;

}

int main(int argc, char\* argv[])

{

int proc\_rank, proc\_num;

MPI\_Status st;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &proc\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &proc\_num);

int\* a = NULL;

int\* b = NULL;

int\* c = NULL;

int n;

double start, end;

if (proc\_rank == 0)

{

//  читаем n, генерируем массивы, рассылаем остальным процессам n

std::cout << "Input N" << std::endl;

std::cin >> n;

a = genArray(n);

b = genArray(n);

c = new int[n];

start = MPI\_Wtime();

for (int i = 1; i < proc\_num; ++i)

MPI\_Send(&n, 1, MPI\_INT, i, answerTag, MPI\_COMM\_WORLD);

print(a, n);

print(b, n);

}

else

{

//  считываем n

MPI\_Recv(&n, 1, MPI\_INT, 0, answerTag, MPI\_COMM\_WORLD, &st);

}

//  вычисляем размеры массивов и смещения

int\* sendcounts = new int[proc\_num];

int\* displs = new int[proc\_num];

int remainder = n;

for (int i = 0; i < proc\_num; ++i)

{

displs[i] = i > 0 ? displs[i - 1] + sendcounts[i - 1] : 0;

sendcounts[i] = ceil(remainder / (double)(proc\_num - i));

remainder -= sendcounts[i];

}

int\* aBuff = new int[sendcounts[proc\_rank]];

int\* bBuff = new int[sendcounts[proc\_rank]];

//  рассылаем a и b

MPI\_Scatterv(a, sendcounts, displs, MPI\_INT, aBuff, sendcounts[proc\_rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatterv(b, sendcounts, displs, MPI\_INT, bBuff, sendcounts[proc\_rank], MPI\_INT, 0, MPI\_COMM\_WORLD);

//  вычисляем сумму векторов

int\* cBuff = new int[sendcounts[proc\_rank]];

for (int i = 0; i < sendcounts[proc\_rank]; ++i)

cBuff[i] = aBuff[i] + bBuff[i];

//  собираем всё

MPI\_Gatherv(cBuff, sendcounts[proc\_rank], MPI\_INT, c, sendcounts, displs, MPI\_INT, 0, MPI\_COMM\_WORLD);

//  если главный процесс, то запоминаем время и выводим

if (proc\_rank == 0)

{

end = MPI\_Wtime();

print(c, n);

std::cout << "Time: " << end - start << std::endl;

}

//  чистим память

delete[] sendcounts;

delete[] displs;

delete[] a;

delete[] b;

delete[] c;

delete[] aBuff;

delete[] bBuff;

delete[] cBuff;

MPI\_Finalize();

return 0;

}

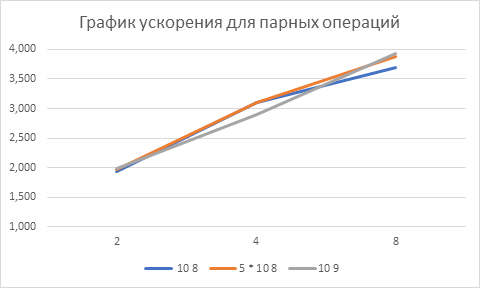
# Полученные результаты

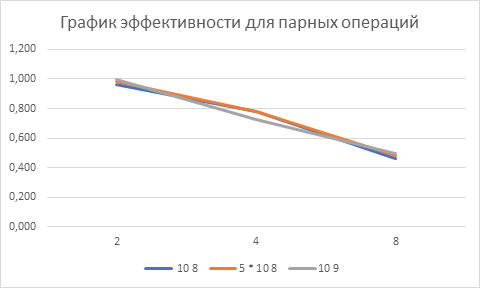
Тестирование проводилось на ОС Windows 10 x64 с установленным MSMPI и с системой на процессоре Intel Core i7 4770 с базовой частотой 3.4 ГГц (турбобуст до 3.9 на ядро, до 3.7 на всех ядрах), четырьмя ядрами и восьмью потоками, 16 Гб ОЗУ с частотой 1600 МГц.

Задание 1

Парные операции:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер, n | Число процессов | | | |
| 1 | 2 | 4 | 8 |
| 10 8 | 0,377 | 0,196 | 0,121 | 0,102 |
| 5 \* 10 8 | 1,894 | 0,964 | 0,610 | 0,489 |
| 10 9 | 3,844 | 1,935 | 1,328 | 0,977 |
| Таблица 1. Время работы, сек. | | | | |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 1,924 | 3,106 | 3,699 |
| 5 \* 10 8 | 1,964 | 3,106 | 3,873 |
| 10 9 | 1,987 | 2,894 | 3,936 |
| Таблица 2. Ускорение для парных операций. | | | |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 0,962 | 0,776 | 0,462 |
| 5 \* 10 8 | 0,982 | 0,776 | 0,484 |
| 10 9 | 0,993 | 0,723 | 0,492 |
| Таблица 3. Эффективность для парных операций. | | | |





Коллективные операции:

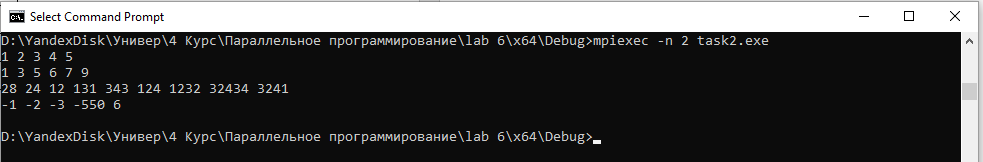
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер, n | Число процессов | | | |
| 1 | 2 | 4 | 8 |
| 10 8 | 0,382 | 0,196 | 0,114 | 0,106 |
| 5 \* 10 8 | 1,916 | 0,961 | 0,597 | 0,486 |
| 10 9 | 3,792 | 1,933 | 1,165 | 0,983 |
| Таблица 4. Время работы при коллективных операциях. | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 1,949 | 3,343 | 3,618 |
| 5 \* 10 8 | 1,993 | 3,208 | 3,939 |
| 10 9 | 1,962 | 3,254 | 3,858 |
| Таблица 5. Ускорение при коллективных операциях. | | | |

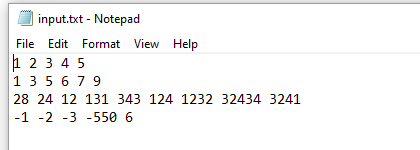
|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 0,975 | 0,836 | 0,452 |
| 5 \* 10 8 | 0,996 | 0,802 | 0,492 |
| 10 9 | 0,981 | 0,813 | 0,482 |
| Таблица 6. Эффективность при коллективных операциях. | | | |

Как видно из таблиц и графиков, разница между парными операциями и коллективными на уровне погрешности из–за того, что в ОС запущены прочие процессы. Ускорение на 8 процессах приближается к 4, а эффективность приближается к 0.5.

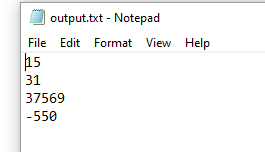
Задание 2



Входной файл:



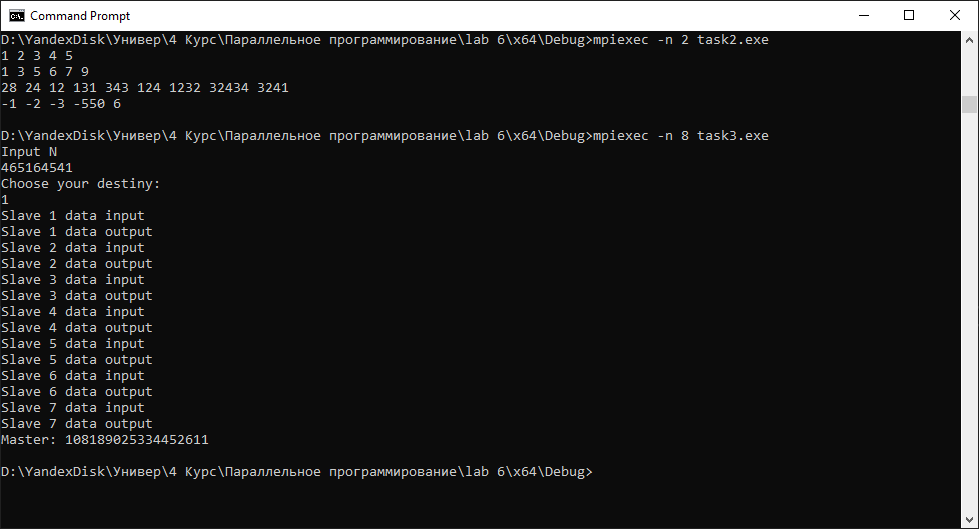
Выходной файл:



Первый процесс передавал массив чисел второму процессу, а второй процесс вычислял сумму чисел и возвращал значение первому. Первый процесс записывал числа в файл.

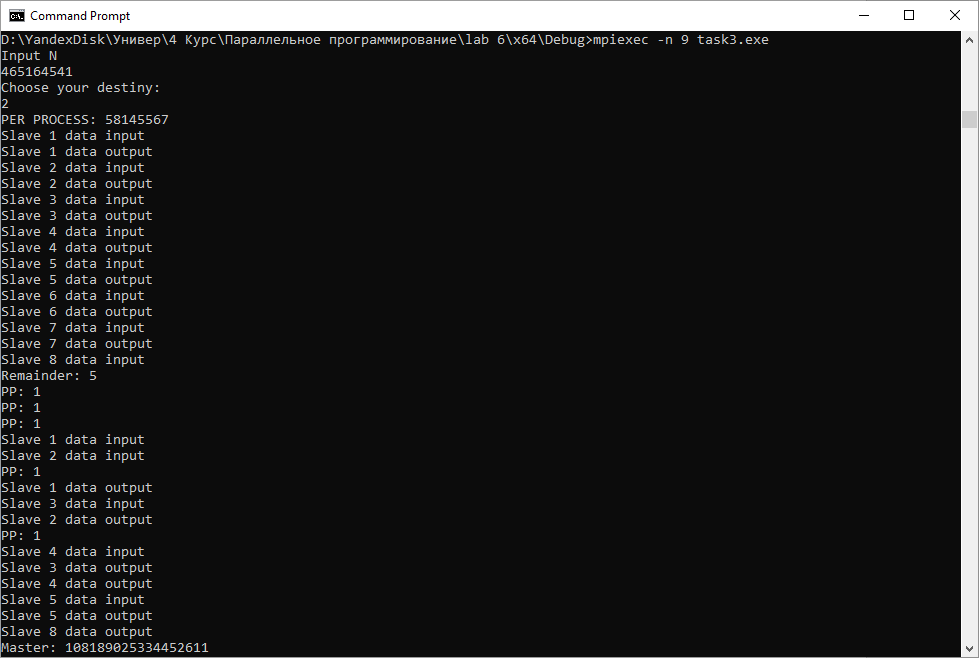
Задание 3

Всем поровну:



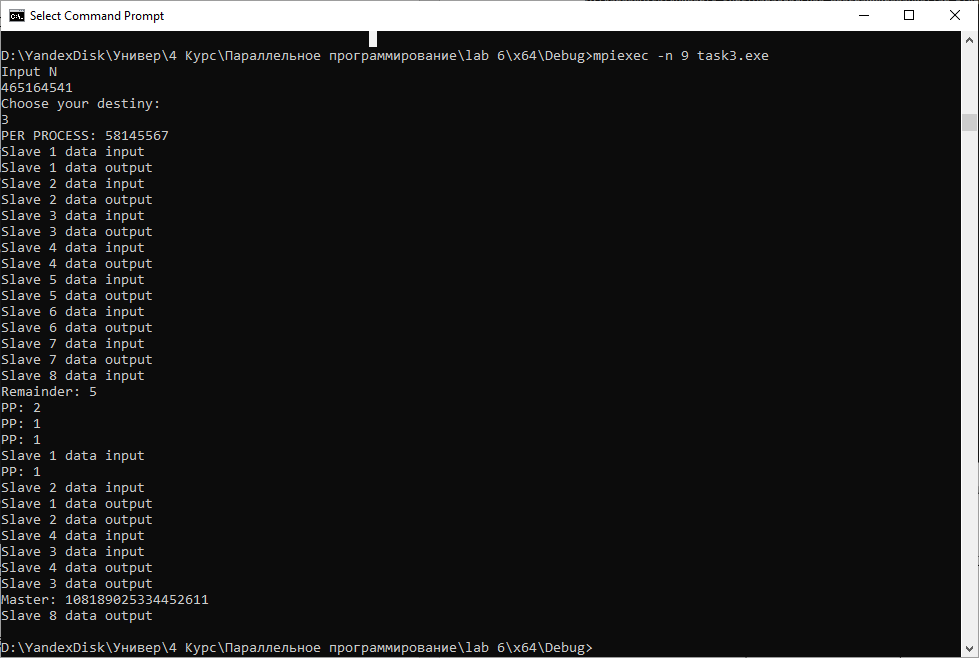
Все процессы получают одинаковое кол-во для вычисления, вычисляют и возвращают главному процессу.

Сначала всем поровну, потом по мере освобождения:



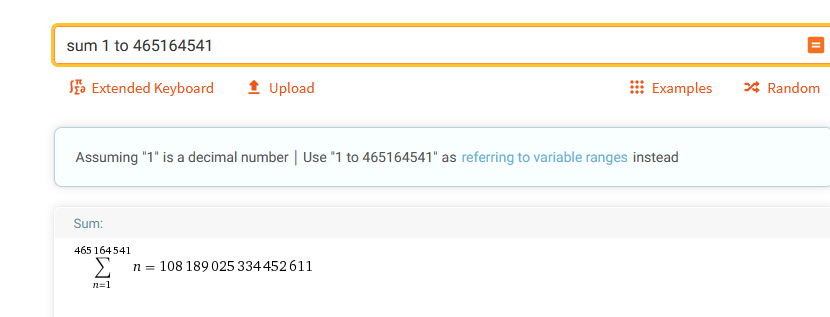
Сначала все получают одинаковую часть, а затем остаток делится между всеми процессами.

Сначала всем одинаковые порции, потом размер порции каждый раз уменьшается в два раза:



Сначала все процессы получают одинаковое кол-во, затем остаток каждый раз делится на два и раздаётся остальным процессам.

Значение в вольфраме:



Задание 4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер, n | Число процессов | | | |
| 1 | 2 | 4 | 8 |
| 10 8 | 0,419 | 0,257 | 0,196 | 0,174 |
| 5 \* 10 8 | 2,189 | 1,291 | 0,916 | 0,832 |
| Таблица 7. Время работы алгоритма, сек. | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 1,628 | 2,141 | 2,411 |
| 5 \* 10 8 | 1,696 | 2,391 | 2,632 |
| Таблица 8. Ускорение от распараллеливания. | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 0,814 | 0,535 | 0,301 |
| 5 \* 10 8 | 0,848 | 0,598 | 0,329 |
| Таблица 9. Эффективность распараллеливания. | | | |

Наибольшие ускорение 2,632 достигается при размере на восьми процессах, эффективность при этом составляет 0,329.

Задание 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер, n | Число процессов | | | |
| 1 | 2 | 4 | 8 |
| 10 8 | 0,767 | 0,544 | 0,471 | 0,444 |
| 5 \* 10 8 | 3,979 | 2,792 | 2,393 | 2,332 |
| Таблица 10. Время работы алгоритма. | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 1,409 | 1,629 | 1,727 |
| 5 \* 10 8 | 1,425 | 1,662 | 1,706 |
| Таблица 11. Ускорение параллельного алгоритма. | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Размер, n | Число процессов | | |
| 2 | 4 | 8 |
| 10 8 | 0,705 | 0,407 | 0,216 |
| 5 \* 10 8 | 0,713 | 0,416 | 0,213 |
| Таблица 12. Эффективность параллельного алгоритма. | | | |

По таблицам и графикам видно, что ускорение достигает всего 1,7 для 8 процессов с эффективностью 0,213. Это связано с тем, что приходится передавать огромное количество данных между процессами (максимальная потребляемая память составила 12 гигабайт ОЗУ). Производительность упирается в скорость обмена информацией между процессами, пропускную способность ОЗУ и мощность процессора. Разница между 4 и 8 процессами незначительна, так как центральный процессор не имеет восьми физических ядер, а использует многопоточность.

# Выводы по лабораторной работе

При выполнении лабораторной работы были получены навыки разработки на С++ с применением MPI. Данная технология позволяет распределить выполнение между несколькими процессами с помощью интерфейса обмена сообщений. С помощью неё удалось получить ускорение почти в 4 раза при вычислении логарифма. Несмотря на то, что каждый процесс имеет собственную выделенную память, при выполнении на одном компьютере удалось достигнуть ускорения в 1,7 раз при использовании 12 гигабайт озу из 16.