

Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем вычислительных комплексов

Практикум по курсу

«Суперкомпьютеры и параллельная обработка данных»

**Отчет по заданию №2**

Разработка параллельной версии программы для вычисления определенного интеграла методом Симпсона с использованием технологии MPI

Коваленко Анастасия Павловна, 321 группа

Москва, 2020

**Оглавление**

[Постановка задачи 3](#_Toc59235564)

[Текст программы 3](#_Toc59235565)

[Результаты измерений времени выполнения 5](#_Toc59235566)

[Выводы 6](#_Toc59235567)

# **Постановка задачи**

Требуется разработать параллельную версию программы, которая вычисляет определенный интеграл методом Симпсона с использованием технологии MPI, и исследовать масштабируемость полученной параллельной программы, построив график зависимости времени выполнения от числа процессов для различного объема входных данных.

# **Текст программы**

#include <iostream>

#include <sys/time.h>

#include <sstream>

#include <mpi.h>

double fun(double x)

{

return x / (x \* x \* x + x \* x + 5 \* x + 1);

}

double integral(double a, double b, int n, int myrank, int num\_procs)

{

double res = 0;

double h = (b - a) / (2 \* n);

double x1, x2;

for (int i = myrank; i <= n; i += num\_procs) {

x1 = a + (2 \* i - 1) \* h;

x2 = a + 2 \* h \* i;

res += 4 \* fun(x1);

res += 2 \* fun(x2);

}

res += fun(b - h);

res += fun(a) + fun(b);

res \*= (h / 3);

return res;

}

int main(int argc, char \*\*argv)

{

int n, num\_procs, rank, flag;

double a = 0, b = 200;

double result;

std::stringstream s;

s << argv[1];

s >> n;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

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

double start, finish;

if (rank == 0) {

start = MPI\_Wtime();

}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

double cur\_sum = integral(a, b, n, rank, num\_procs);

MPI\_Reduce(&cur\_sum, &result, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

std::cout << "Result is " << result << std::endl;

finish = MPI\_Wtime();

std::cout << "Time passed: " << finish - start << std::endl;

}

MPI\_Finalize();

return 0;

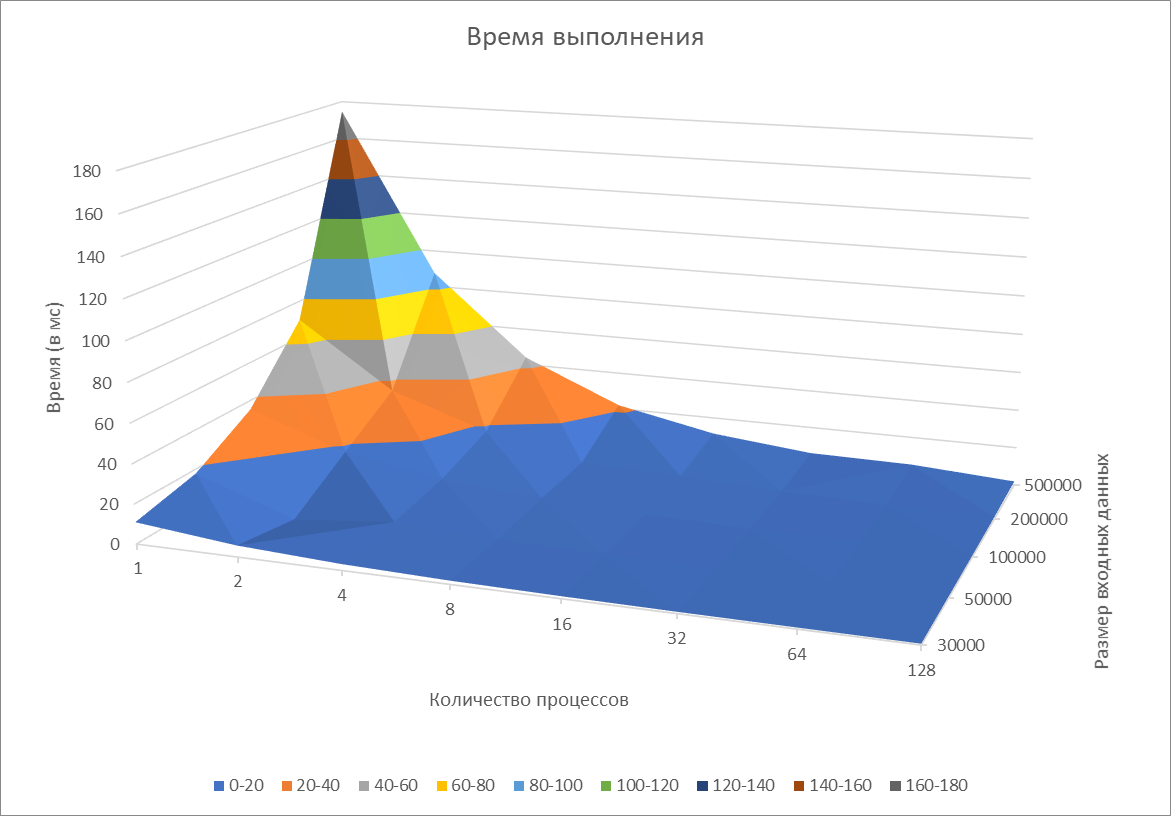
}

# **Результаты измерений времени выполнения**

Ниже приведены результаты измерений времени выполнения полученной программы. Для каждых входных данных программа была запущена несколько раз, в таблице содержатся усредненные результаты без учета случайных выбросов. Время выполнения приведено в формате «секунды.миллисекунды.микросекунды». Зеленым цветом в таблице выделены ячейки, в которых содержится минимальное время выполнения для каждого набора входных данных.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Объем входных данных  Количество процессов | 30000 | 50000 | 100000 | 200000 | 500000 |
| 1 | 0.010.927 | 0.017.812 | 0.035.283 | 0.069.783 | 0.174.567 |
| 2 | 0.005.670 | 0.009.151 | 0.017.855 | 0.035.285 | 0.087.499 |
| 4 | 0.003.079 | 0.004.818 | 0.009.195 | 0.017.872 | 0.044.021 |
| 8 | 0.001.762 | 0.002.636 | 0.004.790 | 0.006.339 | 0.021.940 |
| 16 | 0.001.143 | 0.001.564 | 0.002.661 | 0.004.777 | 0.011.248 |
| 32 | 0.000.805 | 0.001.036 | 0.001.543 | 0.002.641 | 0.005.865 |
| 64 | 0.000.654 | 0.000.752 | 0.001.036 | 0.001.584 | 0.005.221 |
| 128 | 0.000.559 | 0.000.619 | 0.000.755 | 0.001.072 | 0.001.808 |

График зависимости времени исполнения программы от числа процессов и разных входных данных



# **Выводы**

Распараллеливание программы с помощью MPI дало ускорение в десятки раз (примерно 20 раз на небольших входных данных и почти 97 раз на больших объемах данных).

Из полученных результатов можно сделать вывод, что время выполнения программы линейно зависит от числа процессов: чем больше процессов, тем быстрее она выполняется.

Теперь сравним результаты распараллеливания с помощью MPI с результатами использования OpenMP.

Приведена таблица результатов использования OpenMP:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Объем входных данных  Количество  нитей | 30000 | 50000 | 100000 | 200000 | 500000 |
| 1 | 0.1.54 | 0.1.711 | 0.3.374 | 0.6.621 | 0.16.419 |
| 2 | 0.0.607 | 0.0.887 | 0.1.710 | 0.3.619 | 0.8.289 |
| 4 | 0.0.352 | 0.0.510 | 0.0.923 | 0.1.747 | 0.4.210 |
| 8 | 0.0.289 | 0.0.374 | 0.0.652 | 0.1.013 | 0.2.263 |
| 16 | 0.0.408 | 0.0.464 | 0.0.628 | 0.0.854 | 0.1.904 |
| 32 | 0.0.666 | 0.0.711 | 0.0.809 | 0.1.048 | 0.1.605 |
| 64 | 0.2.102 | 0.2.083 | 0.2.193 | 0.2.141 | 0.3.235 |
| 128 | 0.4.179 | 0.4.311 | 0.3.774 | 0.4.039 | 0.4.671 |

Найдем разность значений таблиц (из результатов для MPI вычитаются результаты для OpenMP, время приведено в мс):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Объем входных данных  Количество нитей/процессов | 30000 | 50000 | 100000 | 200000 | 500000 |
| 1 | 9,873 | 16,101 | 31,909 | 63,162 | 158,148 |
| 2 | 5,063 | 8,264 | 16,145 | 31,666 | 79,21 |
| 4 | 2,727 | 4,308 | 8,272 | 16,125 | 39,811 |
| 8 | 1,473 | 2,262 | 4,138 | 5,326 | 19,677 |
| 16 | 0,735 | 1,1 | 2,033 | 3,923 | 9,344 |
| 32 | 0,139 | 0,325 | 0,734 | 1,593 | 4,26 |
| 64 | -1,448 | -1,331 | -1,157 | -0,557 | 1,986 |
| 128 | -3,62 | -3,692 | -3,019 | -2,967 | -2,863 |

Получается, что при большом количестве процессов MPI эффективнее, чем OpenMP, на данных среднего размера и при среднем количестве процессов MPI сравнимо по скорости с OpenMP. Однако для большинства входных данных OpenMP работает быстрее. Скорее всего такая разница в скорости связана с большими накладными расходами на межпроцессное взаимодействие MPI, и использование общей памяти в OpenMP оказывается эффективнее. Для большей эффективности можно использовать комбинацию MPI и OpenMP. В целом параллелизм существенно ускоряет выполнение программы и может быть весьма полезен для решения различных задач.