**Курс "Суперкомпьютеры и параллельная обработка данных"**

**Разработка параллельной версии программы для вычисления определенного интеграла с использованием метода трапеций с помощью технологий OpenMP и MPI**

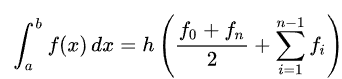
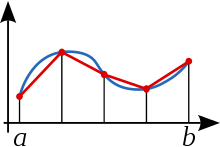
**Отчет о выполненном задании**

**студента 323 учебной группы факультета ВМК МГУ**

**Чернышова Николая Дмитриевича**

1. **Реализованный алгоритм**

В полученном задании необходимо было реализовать параллельную версию программы для вычисления определенного интеграла с использованием метода трапеций. Мною был реализован метод трапеций с равномерной сеткой, т.е. точки, в которых брались значения функции, находились на равном расстоянии друг от друга (b - a) / n, где n – число разбиений, являющееся параметром.

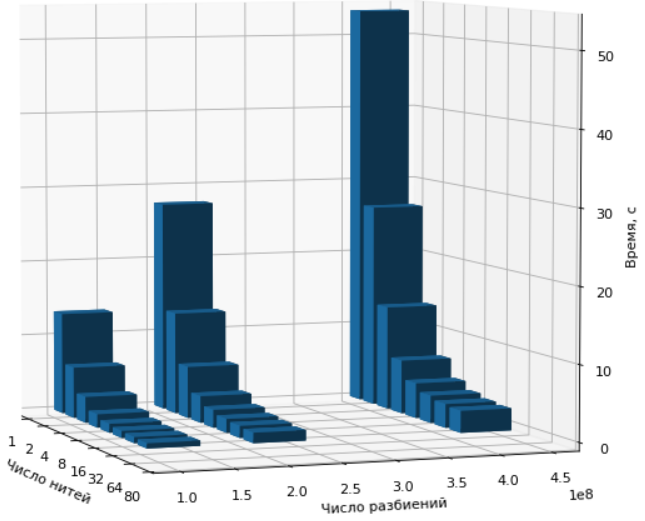


1. **Эффективность полученного решения**

Для того, чтобы проанализировать масштабируемость решения по данным, я тестировал его в трех случаях: 100 миллионов разбиений, 200 миллионов и 400 миллионов. Интегрировались две разных функции f1 и f2 на отрезке [1, 5].

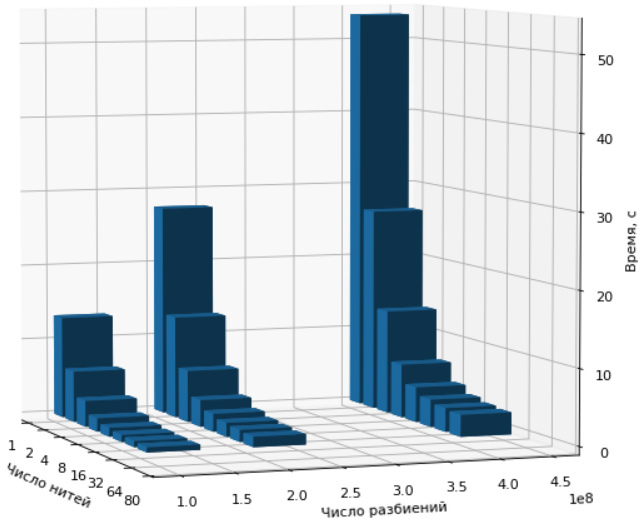
Также замечу, что решение с использованием OpenMP тестировалось на вычислительной машине Polus с параметром количество используемых нитей равным 1; 2; 4; 8; 16; 32; 64; 80, а решение с использованием MPI тестировалось на вычислительной машине Bluegene/P с параметром количество используемых процессов равным 1; 2; 4; 8; 16; 32; 64; 80; 128.

Графики зависимости времени выполнения программы с использованием OpenMP от количества разбиений и количества нитей:



Результаты для функции

f1 = sin(2 \* x) \* sqrt(x)



Результаты для функции

f2 = 1 + x3 \* log(x)

Исходя из полученных результатов для OpenMP можно сделать выводы:

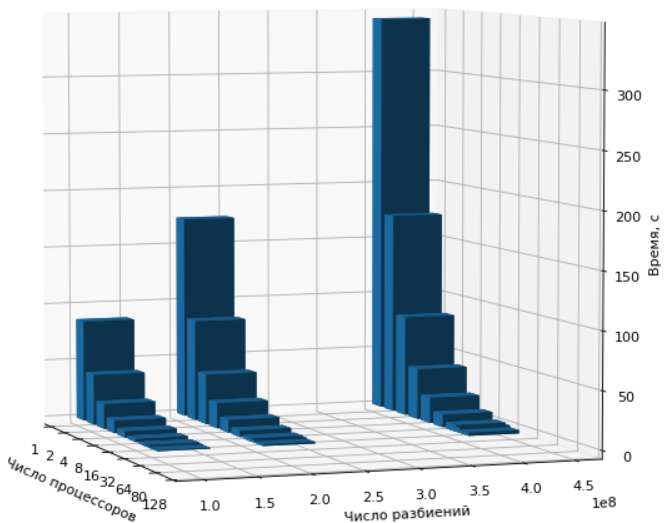
1. Для полученного решения ускорение почти линейно зависит от числа нитей на небольшом числе нитей, но эта зависимость значительно ослабляется при дальнейшем увеличении числа нитей за счет увеличения числа вспомогательных вычислений(например, затраты времени на создание OMP нитей)

Например, для f1 на 400 млн разбиений переход с 1 нити на 80 нитей вызвал ускорение работы примерно в 56.080656 / 2.864795 = 19.574 раза, а переход с 1 нити на 8 нитей вызвал ускорение работы примерно в 56.080656 / 7.378039 = 7.601 раза

1. Решение хорошо масштабируется относительно числа разбиений(почти линейная зависимость времени работы от числа разбиений)

Например, для f1 на 80 нитях увеличение работы программы при переходе с 100 млн на 400 млн разбиений составило примерно 2.864795 / 0.721326 = 3.972 раза

Графики зависимости времени выполнения программы с использованием MPI от количества разбиений и количества нитей:

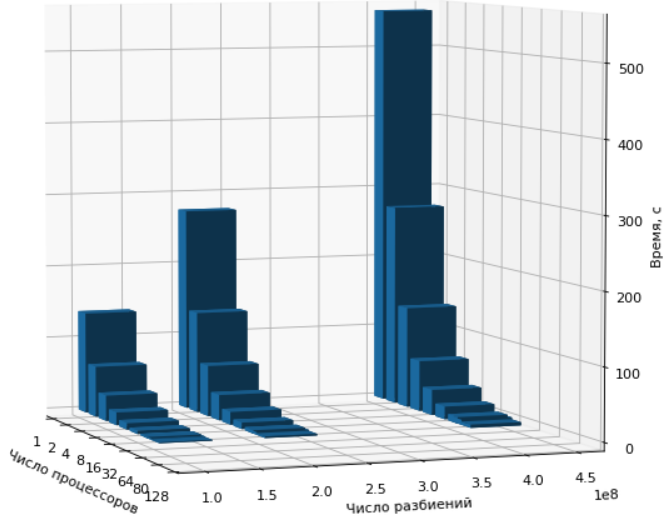


Результаты для функции

f1 = sin(2 \* x) \* sqrt(x)

Результаты для функции

f2 = 1 + x3 \* log(x)



Исходя из полученных результатов можно сделать выводы:

1. Для полученного решения ускорение почти линейно зависит от числа нитей на любом из исследованных количеств используемых процессов

Например, для f1 на 400 млн разбиений переход с 1 процесса на 128 процессов вызвал ускорение работы примерно в 348.756451 / 2.795036 = 124.78 раза

1. Решение отлично масштабируется относительно числа разбиений(опять же почти линейная зависимость времени от числа разбиений)

Например, для f1 на 128 процессах увеличение работы программы при переходе с 100 млн на 400 млн разбиений составило примерно 2.795036 / 0.698918 = 3.998 раза

Таким образом, при сравнении полученных решений на MPI и OpenMP можно сделать выводы, что масштабируемость относительно кол-ва разбиений на OpenMP лишь немного уступает масштабируемости на MPI, но при этом ускорение с увеличением числа нитей на OpenMP **значительно** менее эффективно, чем ускорение с увеличением числа процессов на MPI.

Также интересно, что отношение времени работы программы по интегрированию функции f1 к времени работы программы по интегрированию функцииf2 значительно отличается в процессе исследования двух рассмотренных решений. В частности,

Tomp+polus(f2) / Tomp+polus(f1) составляет около 1.9 на 1 нити

Tmpi+bluegene(f2) / Tmpi+bluegene(f1) составляет около 1.6 на 1 процессе

1. **Текст программы на Си с использованием OMP**

Обе программы можно найти по ссылке https://github.com/nickcherr/skpod2021

Написанный мной текст программы выглядит следующим образом:

#include <stdio.h>

#include <omp.h>

#include <math.h>

// Интегрируемая функция 1

double function1(double x) {

return sin(2 \* x) \* sqrt(x);

}

// Интегрируемая функция 2

double function2(double x) {

return (1.0 + pow(x, 3) \* log(x));

}

int main() {

double start\_time, end\_time;

long double sum = 0, res, w;

int num\_threads;

//Пределы интегрирования

double a = 1.0, b = 5.0;

//Число разбиений

long n = 400000000;

start\_time = omp\_get\_wtime();

w = (b - a) / (double)n;

#pragma omp parallel shared(w) reduction(+:sum)

{

num\_threads = omp\_get\_num\_threads();

#pragma omp for schedule(static)

for(long i=1; i < n - 1; i++)

//sum += function1(a + w \* i);

sum += function2(a + w \* i);

}

//sum += (function1(a) + function1(b)) / 2;

sum += (function2(a) + function2(b)) / 2;

res = w \* sum;

end\_time = omp\_get\_wtime();

printf("N=%ld, Nproc=%d, res=%Lf, Time=%lf \n", n, num\_threads, res, end\_time - start\_time);

return 0;

}

Программа запускалась на Polus с использованием скрипта для упрощения тестирования программы с разным количеством используемых нитей.

1. **Текст программы на Си с использованием MPI**

#include <mpi.h>

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

// Интегрируемая функция 1

long double function1(long double x) {

return sin(2 \* x) \* sqrt(x);

}

// Интегрируемая функция 2

long double function2(long double x) {

return (1.0 + pow(x, 3) \* log(x));

}

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

double start\_time, end\_time;

long double sum = 0.0, all\_sum, res, w;

int error\_code = MPI\_Init(&argc, &argv);

if (error\_code != 0)

return error\_code;

//Пределы интегрирования

long double a = 1.0, b = 5.0;

//Число разбиений

long n = 100000000;

//Переменные для группового взаимодействия процессов в MPI

int myrank, ranksize, i;

//Определяем свой номер в группе:

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

//Определяем размер группы:

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ranksize);

start\_time = MPI\_Wtime();

w = (b - a) / (long double)n;

long double cur\_a, cur\_b, cur\_w;

long num\_per\_p;

long double \*sbuf = NULL;

if (!myrank) { //Процесс-Master

//Определяем размер диапазона для каждого процесса:

num\_per\_p = n / ranksize;

sbuf = (long double\*)calloc(ranksize\*3, sizeof(long double));

if (sbuf == NULL) return 1;

cur\_a = a;

for (i = 0; i < n - ranksize \* num\_per\_p; i++) {

cur\_b = cur\_a + (num\_per\_p + 1) \* w;

sbuf[i \* 3] = cur\_a;

sbuf[i \* 3 + 1] = cur\_b;

sbuf[i \* 3 + 2] = w;

cur\_a = cur\_b;

}

for (i = n - ranksize \* num\_per\_p; i < ranksize; i++) {

cur\_b = cur\_a + num\_per\_p \* w;

sbuf[i \* 3] = cur\_a;

sbuf[i \* 3 + 1] = cur\_b;

sbuf[i \* 3 + 2] = w;

cur\_a = cur\_b;

}

}

long double rbuf[3];

//Рассылка всем процессам, включая процесс-мастер

//начальных данных для расчета:

MPI\_Scatter(sbuf, 3, MPI\_LONG\_DOUBLE, rbuf, 3, MPI\_LONG\_DOUBLE, 0, MPI\_COMM\_WORLD);

if (!myrank)

if(sbuf)

free(sbuf);

cur\_a = rbuf[0];

cur\_b = rbuf[1];

cur\_w = rbuf[2];

sum = 0.0;

//Расчет интеграла в своем диапазоне, выполняют все процессы:

// sum += (function1(cur\_a) + function1(cur\_b)) / 2.0 ;

sum += (function2(cur\_a) + function2(cur\_b)) / 2.0 ;

cur\_a += cur\_w;

for( ; cur\_a < cur\_b; cur\_a += cur\_w)

// sum += function1(cur\_a);

sum += function2(cur\_a);

//Редуцируем значения интегралов от процессов:

MPI\_Reduce(&sum, &all\_sum, 1, MPI\_LONG\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (!myrank) { //Процесс-Master

res = w \* all\_sum;

end\_time = MPI\_Wtime();

printf("N=%ld, Nproc=%d, res=%Lf, Time=%lf \n", n, ranksize, res, end\_time - start\_time);

}

//Завершение работы с MPI

MPI\_Finalize();

return 0;

}

Программа запускалась на Bluegene/P с использованием скрипта mpisubmit.bg с разным количеством используемых процессов.