МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по курсу

"Суперкомпьютеры и параллельная обработка данных"

Разработка параллельной версии программы для вычисления определенного интеграла с использованием метода Симпсона

ОТЧЕТ

о выполненном задании

студента 320 учебной группы факультета ВМК МГУ

Швецова Федора Алексеевича

Москва, 2018 г.

Оглавление

1 Постановка задачи

Требуется:

1. Разработка параллельной версии программы для вычисления определенного интеграла с использованием метода Симпсона с помощью технологий параллельного программирования OpenMP и MPI.
2. Сравнить их эффективность.
3. Исследовать масштабируемость полученных программ и построить графики зависимости времени выполнения программ от числа используемых ядер и объёма входных данных.

2 Описание вычисления определенного интеграла методом Симпсона

2.1 Математическая основа

Задана функция вида $y = f(x)$, имеющая непрерывность на интервале $[a;\ b]$, необходимо произвести вычисление определенного интеграла $\int_a^b f\left(x\right)dx$

Необходимо разбить отрезок $[a;\ b]$ на n отрезков вида $[x_{2i-2}; x_{2i}],\ i = 1,\ 2, \ldots,\ n$ с длиной $2h = \frac{b-a}{n}$ и точками $a = x_0 < x_2 < x_4 < \ldots < x_{2\pi-2} < x_{2\pi} = b$. Тогда точки $x_{2i-1},\ i = 1,\ 2, \ldots,\ n$ считаются серединами отрезков $[x_{2i-2};\ x_{2i}],\ i = 1,\ 2, \ldots,\ n$. Данный случай показывает, что определение узлов производится через $x_i = a + i \cdot h,\ i = 0,\ 1, \ldots,\ 2n$.

Формула метода Симпсона имеет вид

$$\int_a^b f\left(x\right)dx \approx \frac{h}{3}\left(f\left(x_0\right) + 4\sum_{i=1}^{n} f\left(x_{2i-1}\right) + 2\sum_{i=1}^{n-1} f\left(x_{2i}\right) + f\left(x_{2n}\right)\right).$$

2.2 Последовательный алгоритм

Для реализации данного алгоритма достаточно использовать всего лишь один цикл for. Вычисления надо производить в соответствии с представленной выше формулой.

```
double simpson_integral(pointFunc f, double a, double b, int n) {
    const double h = (b - a) / n;
    double k1 = 0, k2 = 0;

    for (int i = 1; i < n; i += 2) {
        k1 += f(a + i * h);
        k2 += f(a + (i + 1) * h);
    }
    return h / 3 * (f(a) + 4 * k1 + 2 * k2 + f(b));
}
```

2.3 Параллельный алгоритм

2.3.1 OpenMP

Для использования OpenMP модуля достаточно добавить всего одну клаузу #pragma, так как в программе не приходится использовать общую память, а надо распараллелить один цикл for. Каждому потоку придётся создать локальные переменный k1, k2, i. И по завершению работы мастер-потоку надо будет просуммировать значения k1, k2, всех остальных потоков

```
#pragma omp parallel for reduction(+:k1, k2)
```

2.3.2 MPI

В разработке

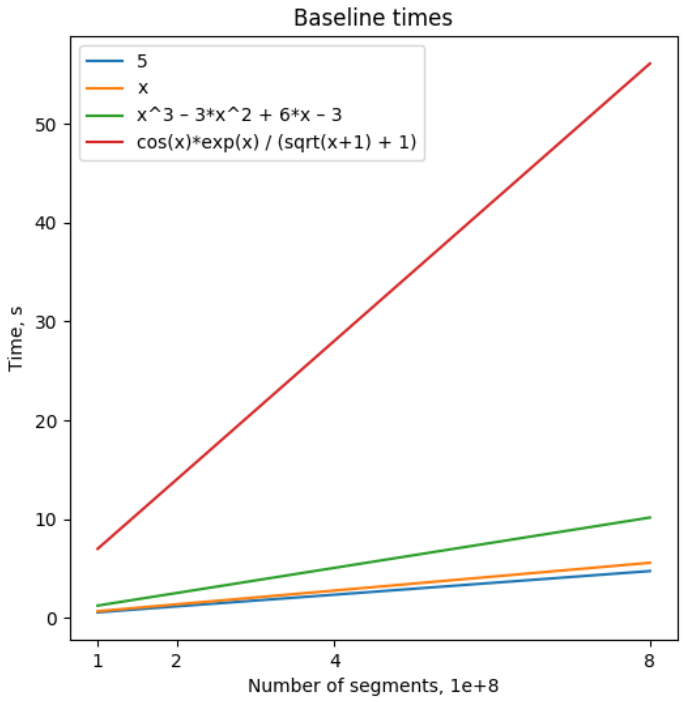
3 Результаты замеров времени выполнения

В качестве разных входных данных использовались функции разной сложности.

1) f(x) = 5;
2) f(x) = x;
3) f(x) = $x^3 - 3*x^2 + 6*x - 3$
4) f(x) = cos(x)*exp(x) / (sqrt(x+1) + 1)

Интегрирование производилось на отрезке [0; 1].
Количество отрезков, на которые мы разбиваем наш сегмент: $n*10^8$, n = 1, 2, 4, 8.
Замеры времени производились с помощью функции omp_get_wtime();
Для каждого набора входных данных происходило три итерации просчёта с последующим усреднением времени.

3.1 Последовательная версия на Polus

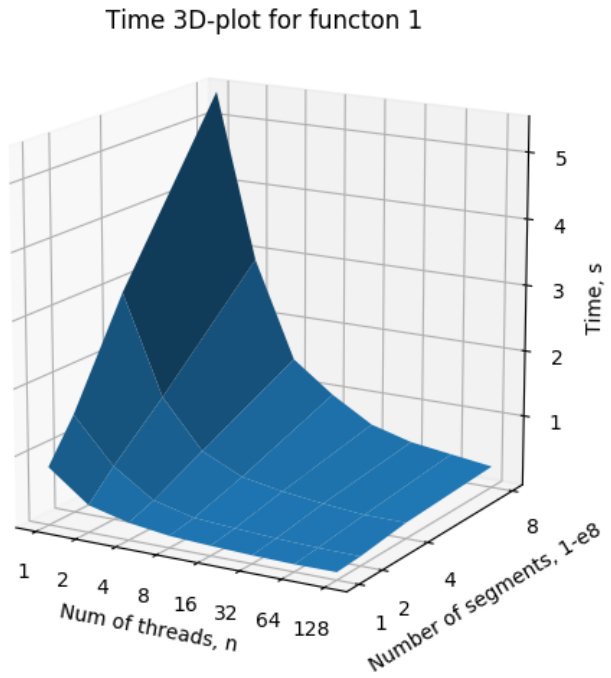


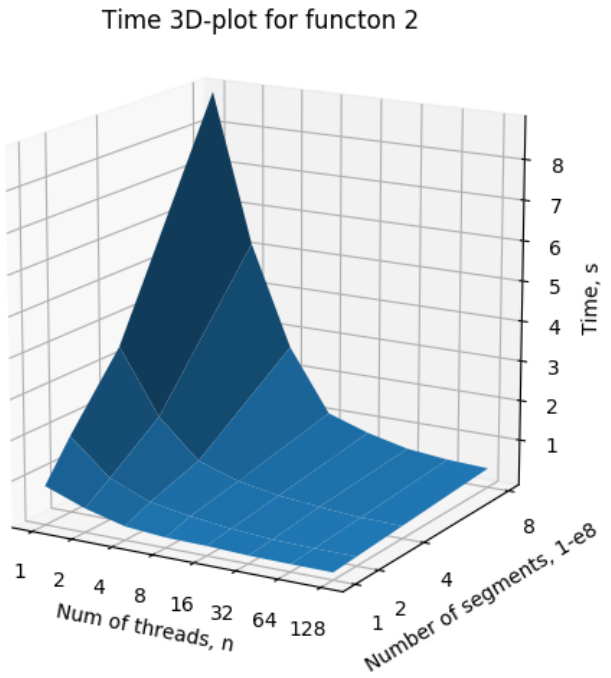
На графике можно наблюдать зависимость времени работы алгоритма для разных функций и для разного количества отрезков. Ожидаемо зависимость от кол-ва отрезков линейная.

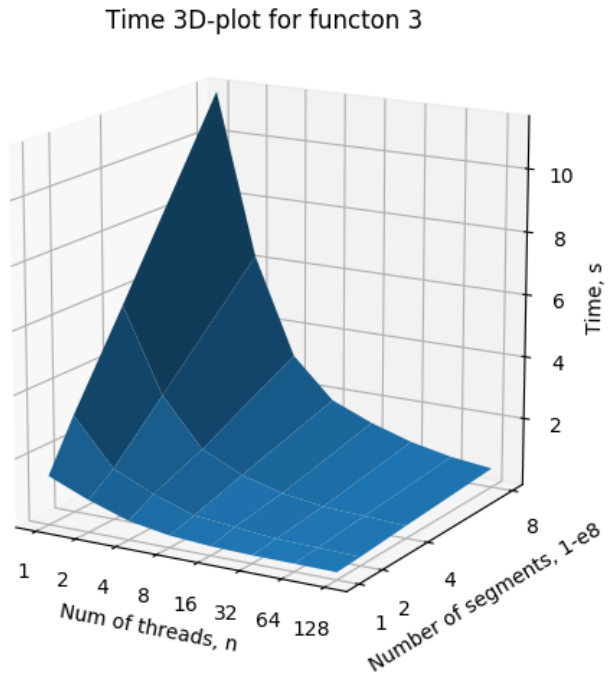
3.2 OpenMP на Polus

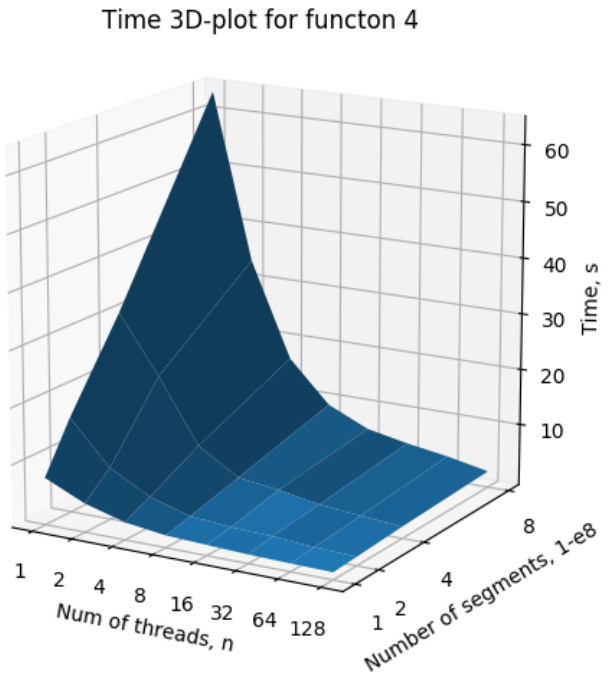
Я использовал 32 процессора и количество потоков 2^k, k = 0, …, 7.

3D Графики зависимости времени выполнения программы от количества потоков и количество сегментов разбиения. По оси количества потоков использована логарифмическая шкала для наглядности.

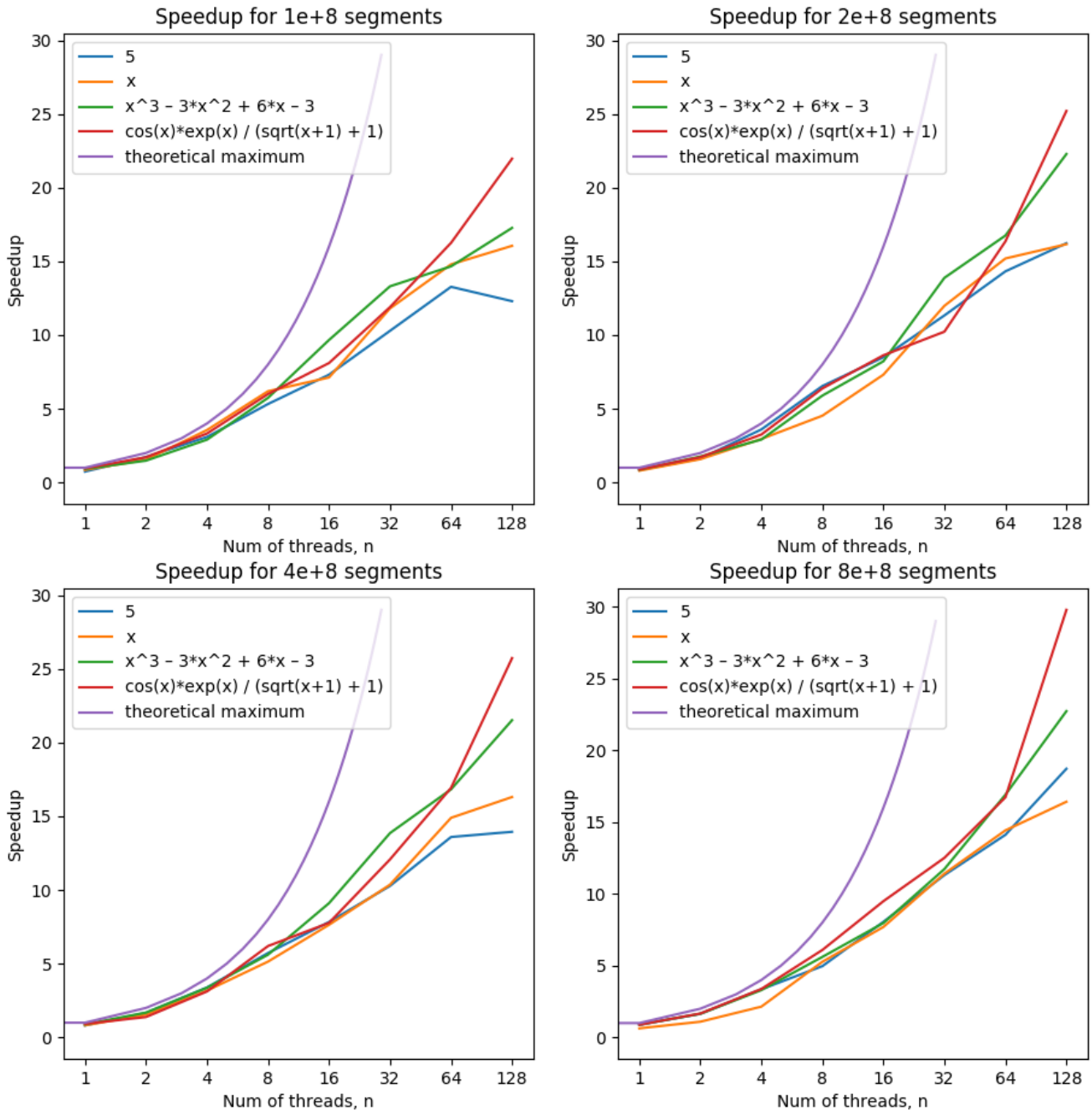








Графики ускорения работы программ. По оси количества потоков использована логарифмическая шкала для наглядности.



3.3 MPI на Polus

В разработке

4 Анализ результатов

4.1 OpenMP

По результатам тестов видно, что задача хорошо параллелизуется. Получалось достигать ускорения работы вплоть до 30 раз. OpenMP хорошо подходит под данную задачу, так как здесь не требуется общение между потоками или использование общей памяти. Так же на основе тестов можно сделать вывод о том, что выбор функции практически не влияет на степень параллелизации, а вот увеличение количество отрезков, на которые мы разбиваем наш интервал позволяет лучше параллелизировать задачу, увеличив максимальное ускорение с (около) 20 до (около) 30 раз.

4.2 MPI

В разработке.

4.3 Сравнение

В разработке.

5 Выводы

В разработке.