Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу

«Суперкомпьютеры и Параллельная обработка данных»

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

ОТЧЕТ

о выполненном задании

студента 327 учебной группы факультета ВМК МГУ
Галустова Артемия Львовича

гор. Москва
2021 год

Содержание

Постановка задачи

Рассматривается определенный интеграл на отрезке $[a, b]$ функции одной переменной $f(x)$:

$$\int_a^b f(x)dx$$

В ходе решения задачи требуется:

1. Реализовать алгоритм вычисление определенного интеграла методом трапеций.
2. Используя стандарт OpenMP адаптировать программу для параллельных вычислений.
3. Сравнить скорость вычислений при разном размере входных данных и разном количестве потоков.
4. Установить оптимальное количество потоков для каждого выбранного объема данных и объяснить полученные зависимости.

Описание метода решения

Сущность метода трапеций заключается в замене в каждой ячейки сетки подынтегральной функции на многочлен первой степени, т.е. линейную функцию, проходящую через значения функции в соседних узлах сетки. Тогда площадь под графиком аппроксимируется трапецией. Для отрезка между узлами x_n и x_{n+1} имеет место

$$\int_{x_n}^{x_{n+1}} f(x)dx = \frac{f(x_{n+1}) + f(x_n)}{2}(x_{n+1} - x_n) + E(f)$$

Для всего интервала формула имеет вид:

$$\int_a^b f(x)dx = \frac{f(a)}{2}(x_1 - a) + \sum_{i=1}^{n-1} \frac{f(x_i)}{2}(x_{i+1} - x_{i-1}) + \frac{f(b)}{2}(b - x_{n-1})$$

В данной реализации используется равномерная сетка, потому возможно применить формулу Котеса:

$$\int_a^b f(x)\,dx = h\left(\frac{f_0 + f_n}{2} + \sum_{i=1}^{n-1} f_i\right)$$

Решаемая задача

Для тестирования производительности необходимо предоставить программе достаточно сложную задачу. В данном случае в качестве интегрируемой функции была выбрана функция $f(x) = asin(x) + e^x + H(x)$. Три суммируемые функции были дополнительно разложены в ряды, чтобы исследовать ситуацию, когда функция без аналитического представления:

$$\arcsin x = \sum_{n=0}^{\inf} \frac{(2n)!}{4^n(n!)^2(2n+1)} x^{2n+1}$$

$$H(x) = \sum_{n=0}^{\inf} \frac{2}{n\pi} \sin nx$$

$$e^x = \sum_{n=0}^{\inf} \frac{x^n}{n!}$$

Использовалась равномерные сетки с разбиением на $1000, 10000, 100000$ и 1000000 узлов. Это позволило получить время выполнения алгоритма в пределах от 0.006 секунд до 2 минут.

Параллелизация и оптимизация алгоритма

Основной оптимизацией для программы стало предвычисление коэффициентов ряда. Это управляется семейством опций $\_PREPARE\_COEFFICIENTS$.

Также была добавлена опция для тестирования с распараллеливанием циклов внутри функций $\arcsin x, H(x), e^x$. Использовалась клауза omp parallel for. Управляется семейством опций $\_SERIES\_PARALLEL$. Также был распараллелен основной цикл интегрирования при помощи клаузы omp parallel. Полный код программы доступен в разделе «Полный листинг программы», а также онлайн по адресу `https://github.com/NotLebedev/integralsOpenMP`.

Методика тестирования

Для получения репрезентативных и точных результатов был использован следующий подход для тестирования. Вначале программа выполняет три "прогревочных"раунда вычислений. Их время не учитывается в результате. Затем программа выполняет 5 раундов вычислений и усредняет время их выполнения измеренное при помощи $omp\_get\_wtime()$. Между раундами осуществляется сброс предвычисленных коэффициентов рядов. Данное поведение включается опцией $BENCHMARK$. Количество раундов задаётся опциями $WARMUP\_ROUNDS\_CNT$ и $ROUNDS\_CNT$.

Тестируемые конфигурации и результаты

Первым произведённым тестом было сравнение эффективности программ производимых компиляторами IBM XL C/C++ (xlc++) и GNU Compiler Collection (g++). Эффективность программ сравнивалась на разбиении на 100000 узлов с предвычислением коэффициентов, но без распараллеливания внутренних циклов.

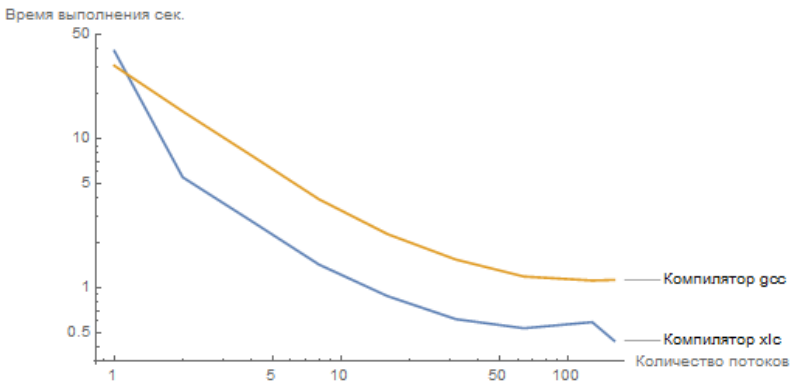

Рис. 1: Сравнение производительности программ собраных xlc и gcc

Данное сравнение показывает существенное превосходство программ генерируемых xlc, потому все дальнейшие тесты были произведены с использование именно его.

Для тестирования производительности были выполнены запуски по описанной выше методике с использованием трёх конфигураций:

- Без оптимизаций
- С предвычислением коэффициентов
- С предвычислением коэффициентов и параллелизацией внутренних циклов

Были полученны следующие результаты:

Optimizations	No	Preopt	Full	No	Preopt	Full
Series	1000			10000		
1	0.458736	0.114494	0.114606	9.12695	1.74852	7.8508
2	0.229843	0.0554668	0.0575435	6.83397	1.65143	1.71756
4	0.116875	0.0285275	0.0294921	5.7009	1.40852	1.47356
8	0.0592613	0.0145033	0.0151491	5.18292	1.31402	1.33771
16	0.0346732	0.00908069	0.00933806	5.15853	1.28186	1.31363
32	0.0247849	0.00663345	0.179549	7.01354	1.99508	1.71551
64	0.0209962	0.00600682	0.57386	11.5531	2.82944	1.91566
128	0.0262914	0.00730744	0.572106	18.6607	5.68873	5.04403
160	0.0423315	0.00730314	0.998173	31.6462	10.1763	10.1221

Optimizations	No	Preopt	Full	No	Preopt	Full
Series	100000			1000000		
1	128.656	38.9753	38.6009		110.277	114.706
2	22.9074	5.53815	5.7552		55.3834	57.5783
4	11.48	2.82549	2.94348	114.855	28.3578	29.4386
8	5.80515	1.43243	1.48801	57.9605	14.3287	14.8846
16	3.33129	0.880636	0.920509	33.3129	8.83835	9.09575
32	2.29806	0.616865	0.647431	22.9805	6.18621	6.39415
64	1.85798	0.535527	0.562976	18.5612	5.38622	5.62526
128	1.95903	0.588818	0.624994	19.9604	6.03441	6.31218
160	1.52373	0.441483	0.475114	15.4235	4.45109	4.81803

Для наглядности на графиках:

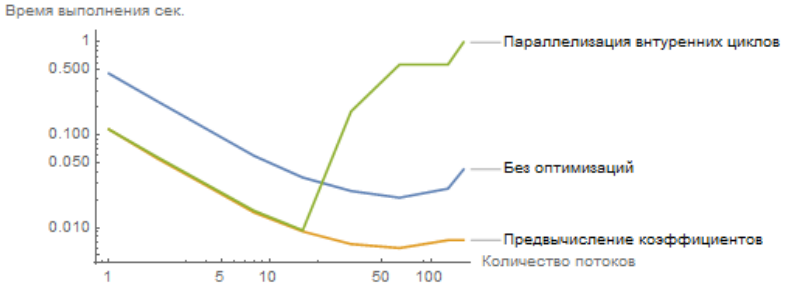

Рис. 2: Время выполнения для 1000 узлов

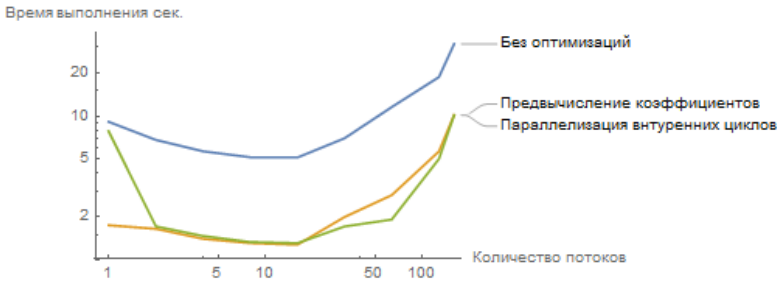

Рис. 3: Время выполнения для 10000 узлов

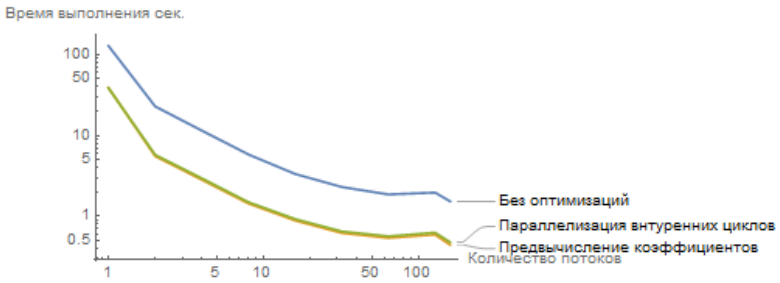

Рис. 4: Время выполнения для 100000 узлов

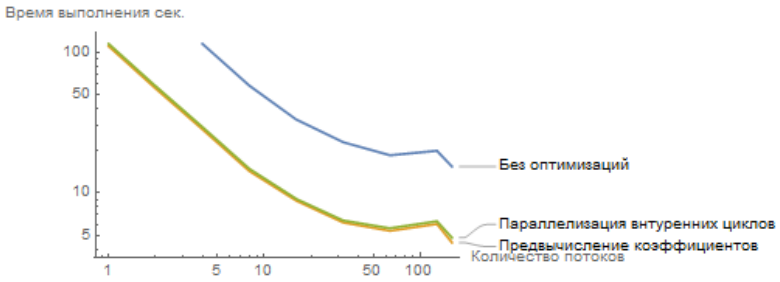

Рис. 5: Время выполнения для 1000000 узлов

Выводы

Распараллеливание даёт существенный выигрыш при больших объемах вычислений. Это заметно как и в неэффективности использования значительного числа потоков для вычисления при малом числе узлов сетки, так и при применении параллелизации внутренних циклов. Предвычисление коэффициентов даёт существенное улучшение производительности чисто программными средствами. При малых объемах вычислений увеличение числа потоков приводит к ухудшению производительности после некоторого порога. Это связанно с тем, что накладные расходы на создание потоков и IPC начинают превышать выгоду от дальнейшего распараллеливания. Однако даже при значительных объемах вычислений производительность с некоторого момента перестаёт увеличиваться пропорционально увеличению кол-ва потоков. Это также связанно с накладными расходами, которые, не совсем превосходят преимущества параллелизации, но всё же частично уменьшают их. Также любопытно резкое замедление при использовании 128 потоков, время выполнения стабильно хуже чем у 64 и 160, что может объясняться особенностями архитектуры комплекса Polus. Полученные результаты демонстрируют, что для разных объемов вычислений целесообразны разные параметры параллелизации. Так для 1000 и 10000 узлов имеет смысл использовать 16-32 ядер, в то время как для 100000 и 1000000 узлов имеет смысл задействовать доступные ресурсы по максимуму.

Полный листинг программы

Файл main.cpp

```
#include <iostream>
#include <omp.h>

#include "types.h"
#include "functions/arcsin.h"
#include "functions/heaviside_step.h"
#include "functions/exp.h"

/**
 * Compound function with heavy calculations
 */
data_t func_big(data_t x) {
    return exp(x) + heaviside_step(x) + arcsin(x);
}

bool invalidate = false;

class Partition {
public:
    Partition(data_t a, data_t b, size_t n_steps) : a_{a}, b_{b},
   n_steps_{n_steps},
                                                    delta_{(b - a)
   / (data_t)n_steps} {}
    data_t operator() (size_t i) const {
        return a_ + ((double) i) * delta_;
    }

    data_t get_delta() const {
        return delta_;
    }

private:
    data_t a_;
    data_t b_;
    size_t n_steps_;
    data_t delta_;
};

class Result {
public:
    void timer_start() {
        time = omp_get_wtime();
    }

    void timer_end() {
        time = omp_get_wtime() - time;
    }

    void set_result(data_t res) noexcept {
        result = res;
    }
```

```
    data_t get_result() const noexcept {
        return result;
    }

    double get_runtime() const noexcept {
        return time;
    }
private:
    double time;
    data_t result;
};

Result run(const size_t n, const data_t a, const data_t b, const
   func_type func) {
    Result res{};
    res.timer_start();
    data_t result = 0.0;

    const Partition x_(a, b, n);
#pragma omp parallel for shared(func, n, x_), reduction(+:result),
   default(none)
    for (size_t i = 1; i < n - 1; i++) {
        result += func(x_(i));
    }

    result += (func(x_(n)) + func(x_(0))) / 2;
    result *= x_.get_delta();

    res.timer_end();
    res.set_result(result);

    return res;
}

void benchmark(const size_t n, const data_t a, const data_t b,
   const func_type func) {

#ifndef WARMUP_ROUNDS_CNT
#define WARMUP_ROUNDS_CNT 3
#endif
    // Warmup round
    for (size_t i = 0; i < WARMUP_ROUNDS_CNT; i++)
        run(n, a, b, func);

#ifndef ROUNDS_CNT
#define ROUNDS_CNT 5
#endif
    // Actual round
    double avg = 0.0;
    for (int i = 0; i < ROUNDS_CNT; i++) {
        // Invalidate coefficient table
        invalidate = true;
        arcsin(0);
        exp(0);
        heaviside_step(0);
```

```
        invalidate = false;

        Result res = run(n, a, b, func);
        avg  = (avg * i + res.get_runtime()) / (i + 1);
    }

    std::cout << "Average runtime: " << avg << " seconds" << std::
    endl;
}

void actual(const size_t n, const data_t a, const data_t b, const
    func_type func) {
    Result res = run(n, a, b, func);

    std::cout << "Runtime: " << res.get_runtime() << " seconds. ";
    std::cout << "Result: " << res.get_result() << std::endl;
}

int main(int argc, char *argv[]) {
    size_t n = 1000000;
    if (argc == 2) {
        char *end;
        size_t cnt = strtoull(argv[1], &end, 10);
        if (end != argv[1])
            n = cnt;
    }
    const data_t a = -1.0;
    const data_t b = 1.0;
    const func_type func = func_big;

#ifdef BENCHMARK
    benchmark(n, a, b, func);
#else
    actual(n, a, b, func);
#endif

}
```

Файл functions/arcsin.cpp

```
#include <cmath>

#include "arcsin.h"

extern bool invalidate;

/**
 * Arc sine via series expansion
 * Integral of arcsin(x) is sqrt(1 - x^2) + x * arcsin(x)
 * Integral from 0 to 1 is (pi - 2) / 2 = 0.570796
 *
 * Defines:
 * SI_SERIES_PARALLEL -- define to enable parallel calculation of
    series
 * SI_SERIES_SIZE -- defines number of terms to use in series
    expansion, default 800
```

```
 * SI_PREPARE_COEFFICIENTS -- define to prepare array with
   precalculated coefficients
 **/
data_t arcsin(data_t x) {
#ifndef SI_SERIES_SIZE
#define SI_SERIES_SIZE 80
#endif

#ifdef SI_PREPARE_COEFFICIENTS
    static data_t coeffs[SI_SERIES_SIZE] = {0};
    static bool coeffs_ready = false;
#pragma omp threadprivate(coeffs)
#pragma omp threadprivate(coeffs_ready)
    if (invalidate)
        coeffs_ready = false;
    if (!coeffs_ready) {
        for (int n = 0; n < SI_SERIES_SIZE; n++) {
            coeffs[n] = tgammal(2.0 * n + 1) / (powl(4.0, n) * powl
   (tgammal(n + 1), 2.0) * (2.0 * n + 1));
        }
        coeffs_ready = true;
    }
#endif

    data_t result = 0.0;
#ifdef SI_SERIES_PARALLEL
#pragma omp parallel for shared(x), reduction(+:result), default(
   none)
#endif
    for (int n = 0; n < SI_SERIES_SIZE; n++) {
#ifdef SI_PREPARE_COEFFICIENTS
        result += powl(x, 2 * n + 1) * coeffs[n];
#else
        result += powl(x, 2 * n + 1) * tgammal(2.0 * n + 1) / (powl
   (4.0, n) * powl(tgammal(n + 1), 2.0) * (2.0 * n + 1));
#endif
    }

    return result;
}
```

Файл *functions/heaviside_step.cpp*

```
#include <cmath>
#include "heaviside_step.h"

extern bool invalidate;

/**
 * Heaviside step function, or the unit step function. 0 for x <=
   0, 1 otherwise.
 * This representation calculates it using Fourier series
 *
 * Defines:
 * HS_SERIES_PARALLEL -- define to enable parallel calculation of
   series
```

```
 * HS_SERIES_SIZE -- defines number of terms to use in series
   expansion, default 100
 * HS_PREPARE_COEFFICIENTS -- define to prepare array with
   precalculated coefficients
 **/
data_t heaviside_step(data_t x) {
#ifndef HS_SERIES_SIZE
#define HS_SERIES_SIZE 100
#endif

#ifdef HS_PREPARE_COEFFICIENTS
    static data_t coeffs[HS_SERIES_SIZE] = {0};
    static bool coeffs_ready = false;
#pragma omp threadprivate(coeffs)
#pragma omp threadprivate(coeffs_ready)
    if (invalidate)
        coeffs_ready = false;
    if (!coeffs_ready) {
        for (int i = 0; i < HS_SERIES_SIZE; i++) {
            coeffs[i] = 2 / ((2 * i + 1) * M_PI);
        }
        coeffs_ready = true;
    }
#endif

    data_t result = 0.5;
#ifdef HS_SERIES_PARALLEL
#pragma omp parallel for shared(x), reduction(+:result), default(
   none)
#endif
    for (int i = 0; i < HS_SERIES_SIZE; i++) {
        int n = 2 * i + 1;
#ifdef HS_PREPARE_COEFFICIENTS
        result += coeffs[i] * sinl(n * x);
#else
        result += 2 / (n * M_PI) * sinl(n * x);
#endif
    }

    return result;
}
```

Файл functions/exp.cpp

```
#include <cmath>
#include "exp.h"

extern bool invalidate;

/**
 * Exponent function represented by taylor series
 *
 * Defines:
 * EXP_SERIES_PARALLEL -- define to enable parallel calculation of
   series
 * EXP_SERIES_SIZE -- defines number of terms to use in series
```

```
   expansion, default 100
 * EXP_PREPARE_COEFFICIENTS -- define to prepare array with
   precalculated coefficients
 **/
data_t exp(data_t x) {
#ifndef EXP_SERIES_SIZE
#define EXP_SERIES_SIZE 100
#endif

#ifdef EXP_PREPARE_COEFFICIENTS
    static data_t coeffs[EXP_SERIES_SIZE] = {0};
    static bool coeffs_ready = false;
#pragma omp threadprivate(coeffs)
#pragma omp threadprivate(coeffs_ready)
    if (invalidate)
        coeffs_ready = false;
    if (!coeffs_ready) {
        for (int n = 0; n < EXP_SERIES_SIZE; n++) {
            coeffs[n] = 1 / tgammal(n + 1);
        }
        coeffs_ready = true;
    }
#endif

    data_t result = 0.0;
#ifdef EXP_SERIES_PARALLEL
#pragma omp parallel for shared(x), reduction(+:result), default(
   none)
#endif
    for (int n = 0; n < EXP_SERIES_SIZE; n++) {
#ifdef EXP_PREPARE_COEFFICIENTS
        result += coeffs[n] * powl(x, n);
#else
        result += powl(x, n) / tgammal(n + 1);
#endif
    }

    return result;
}
```