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 MPI 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

За основу была взята версия программы, использующая стандарт OpenMP. Была сохранена оптимизация предвычислением коэффициентов ряда. Это управляется семейством опций $\_PREPARE\_COEFFICIENTS$. Оптимизация распараллеливание внутренних циклов была удалена, т.к. она не улучшала или

даже ухудшала производительность. Основная работа заключалась в переработке основного цикла вычисления интеграла и его распараллеливание с использованием функций MPI. В каждый slave-процесс отправляется параметры части разбиения для которых slave-процесс вычисляет $\sum_{i=a_k}^{b_k} f_i$. Разбиение подбирается так, чтобы количнство узлов сетки у различных процессов отличалось максимум на 1. Рассылка осуществляется при помощи MPI_Send, MPI_Recv т.к. использование неблокирующих опреаций не улучшает производительность в силу отсутствия каких либо вычислений между отправкой и приёмом, но требует доп память. Затем после вычисления частичных сум используется $MPI\_Allreduce$ для получения суммы $\sum_{i=1}^{n-1} f_i$. Master-процесс выполняет оставшиеся 4 операции и получает ответ. Также полный код программы доступен в разделе «Полный листинг программы», а также онлайн по адресу `https://github.com/NotLebedev/integralsOpenMP/tree/mpi-impl`.

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 $MPI\_Wtime()$. Между раундами осуществляется сброс предвычисленных коэффициентов рядов. Данное поведение включается опцией $BENCHMARK$. Количество раундов задаётся опциями $ROUNDS\_CNT$ и $WARMUP\_ROUNDS\_CNT$.

Тестируемые конфигурации и результаты

Первым произведённым тестом было сравнение эффективности программ производимых компиляторами IBM XL C/C++ (mpixlcxx++) с и без оптимизаций и GNU Compiler Collection (mpicxx++). Эффективность программ сравнивалась на разбиении на 1000000 узлов с предвычислением коэффициентов.

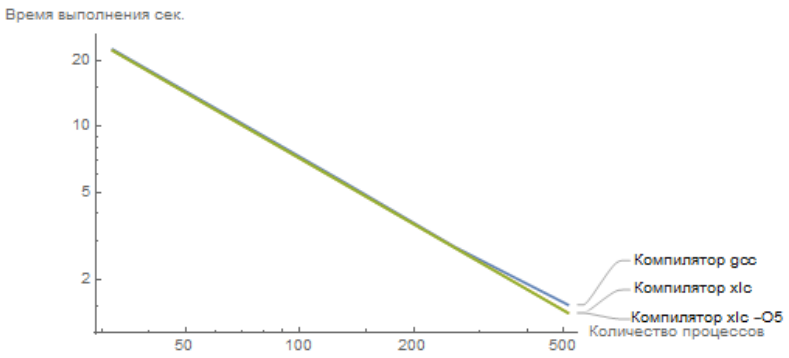

Рис. 1: Сравнение производительности программ собраных xlc и gcc

В отличии от OpenMP программы на машине Polus разница между gcc и xlc незначительна и больше всего проявляется на максимальном числе процессов. Использование оптимизаций для xlc даёт ещё меньший, практически несущественный выигрыш. Для чистоты эксперимента были использованы исполняемы файлы полученные компиляторм xlc с максимальным уровнем оптимизаций.

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

- Без оптимизаций
- С предвычислением коэффициентов

Были полученны следующие результаты:

Optimization	No	Full	No	Full
Series	1000		10000	
1	6.2305	2.76137	62.2856	27.6681
2	3.14104	1.40206	31.3597	14.0333
4	1.57998	0.705443	15.7431	7.06001
8	0.794901	0.355019	7.87509	3.53876
16	0.405254	0.179976	3.94487	1.77335
32	0.0596141	0.0248764	0.508452	0.227737
64	0.0360565	0.0152341	0.262685	0.117083
128	0.0263095	0.0124611	0.13843	0.0621525
256			0.0816035	0.0400828
512			0.0614147	0.0374633

Optimization	No	Full	No	Full
Series	100000		1000000	
8		35.3529		
16	39.3525	17.6991		
32	4.93753	2.22145		22.1799
64	2.47605	1.11448	22.846	11.12
128	1.24735	0.561337	11.6273	5.57871
256	0.630394	0.283789	5.8362	2.79079
512	0.332448	0.155836	3.523	1.39956

Для наглядности на графиках:

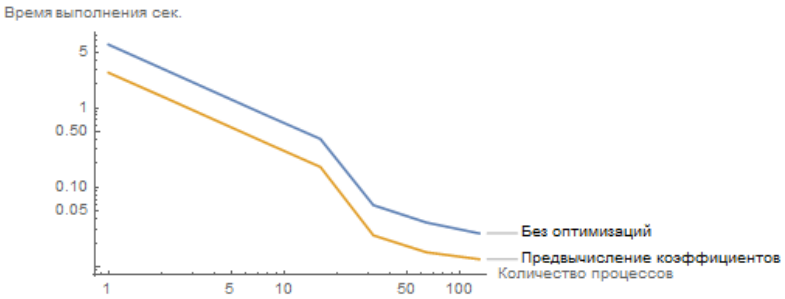

Рис. 2: Время выполнения для 1000 узлов

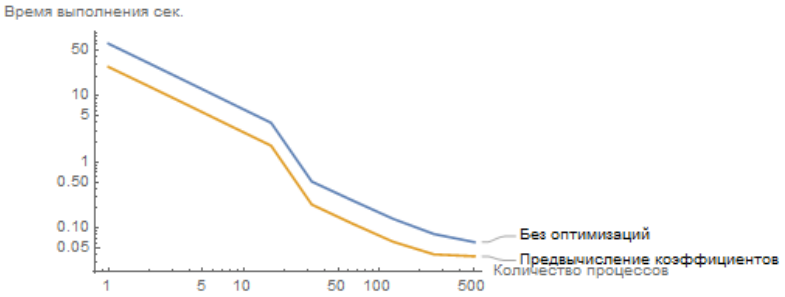

Рис. 3: Время выполнения для 10000 узлов

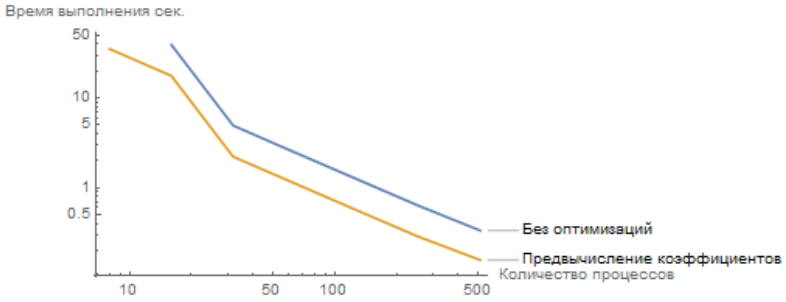

Рис. 4: Время выполнения для 100000 узлов

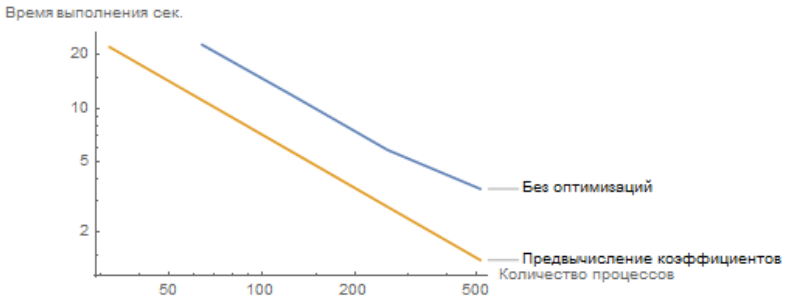

Рис. 5: Время выполнения для 1000000 узлов

Выводы

Распараллеливание даёт значительный выигрыш как на большом так и на малом объеме данных. В отличии от OpenMP на комплексе Polus выигрыш в производительности есть на данных самого малого размера, и при увеличении числа потоков прирост производительности остаётся примерно тем же. Причиной этому могут служить как и большие накладные расходы на IPC в OpenMP, так и то что архитектура комплекса Blue Gene больше подходит для многочисленных MPI процессов, чем архитектура Polus для многочисленных OpenMP нитей. Также любопытно резкое снижение времени работы при переходе от 16 к 32 процессам, это может объясняться особенностью архитектуры узлов Blue Gene а также различными топологиями IPC. для разных объемов вычислений целесообразны разные параметры параллелизации. Так для 1000 и 10000 узлов имеет смысл использовать 64-128 процессов т.к. выигрыш от большего числа незначителен, 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 <mpi.h>
#include <cstdlib>

#include "types.h"
#include "functions/arcsin.h"
#include "functions/heaviside_step.h"
#include "functions/exp.h"
#include "messaging.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 Result {
public:
    void timer_start() {
        time = MPI_Wtime();
    }

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

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

    data_t get_result() const {
        return result;
    }

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

data_t run(Partition *x_, const func_type func) {
    data_t result = 0.0;

    size_t n = x_->get_n();
    for (size_t i = 0; i < n; i++) {
        result += func(x_->get(i));
```

```
    }

    return result;
}

Result *run_full(const size_t n, const data_t a, const data_t b,
    const func_type func) {
    Result *res = new Result();
    res->timer_start();

    Partition *full = new Partition(a, b, n);

    int num_process;
    MPI_Comm_size(MPI_COMM_WORLD, &num_process);

    size_t partition_step = n / num_process;
    for (size_t i = 0, j = 0; j < num_process; j++) {
        if (j < n % num_process) {
            Partition *p = new Partition(full->get(i), full->get(i
    + partition_step + 1), partition_step + 1);
            //printf("Partition %lu %lu %lu\n", i, i +
    partition_step + 1, partition_step + 1);
            i += partition_step + 1;
            master_send_job(p, (int) j);

            delete p;
        } else if (partition_step != 0) {
            Partition *p = new Partition(full->get(i), full->get(i
    + partition_step), partition_step);
            //printf("Partition %lu %lu %lu\n", i, i +
    partition_step, partition_step);
            i += partition_step;
            master_send_job(p, (int) j);

            delete p;
        }
    }

    Partition *p = new Partition(full->get(0), full->get(
    partition_step + (n % num_process > 0)),
                                 partition_step + (n % num_process
    > 0));
    data_t r = run(p, func);
    r = reduce(r);
    delete p;

    r += (func(full->get(n)) - func(full->get(0))) / 2;
    r *= full->get_delta();

    res->set_result(r);
    res->timer_end();

    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_full(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_full(n, a, b, func);
        avg  = (avg * i + res->get_runtime()) / (i + 1);
        delete res;
    }

    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_full(n, a, b, func);

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

    delete res;
}

void run_slave() {
    while (true) {
        Partition *p = slave_receive_job();
        if (p == NULL)
            return;
        const func_type func = func_big;
        data_t res = run(p, func);
        reduce(res);
        delete p;

        // Invalidate coefficient table
        invalidate = true;
        arcsin(0);
        exp(0);
```

```
        heaviside_step(0);
        invalidate = false;
    }
}

void run_master(int argc, char *argv[]) {
    size_t n = 100000;
    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
    int num_process;
    MPI_Comm_size(MPI_COMM_WORLD, &num_process);
    for (int i = 1; i < num_process; ++i) {
        master_send_terminate(i);
    }
}

int main(int argc, char *argv[]) {
    MPI_Init(&argc,&argv);
    int mpi_proc_id;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_proc_id);
    if (mpi_proc_id != 0)
        run_slave();
    else
        run_master(argc, argv);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Файл messaging.cpp

```
#include <cctype>
#include <exception>
#include <mpi.h>

#include "messaging.h"

#define MESSAGE_JOB 0
#define MESSAGE_TERMINATE 1

void master_send_job(const Partition *p, int dest) {
    if (!dest)
        return;
```

```
    unsigned char buff[128];
    data_t a = p->get_a();
    data_t b = p->get_b();
    unsigned long long n = (unsigned long long) p->get_n();

    int pos = 0;
    int m = MESSAGE_JOB;

    MPI_Pack(&m, 1, MPI_INT,                buff, 128, &pos,
   MPI_COMM_WORLD);
    MPI_Pack(&a, 1, MPI_DATA_T,             buff, 128, &pos,
   MPI_COMM_WORLD);
    MPI_Pack(&b, 1, MPI_DATA_T,             buff, 128, &pos,
   MPI_COMM_WORLD);
    MPI_Pack(&n, 1, MPI_UNSIGNED_LONG_LONG, buff, 128, &pos,
   MPI_COMM_WORLD);

    MPI_Send(buff, pos, MPI_PACKED, dest, 0, MPI_COMM_WORLD);
}

void master_send_terminate(int dest) {
    unsigned char buff[128];
    int pos = 0;
    int m = MESSAGE_TERMINATE;

    MPI_Pack(&m, 1, MPI_INT, buff, 128, &pos, MPI_COMM_WORLD);

    MPI_Send(buff, pos, MPI_PACKED, dest, 0, MPI_COMM_WORLD);
}

Partition *slave_receive_job() {
    MPI_Status status;
    unsigned char buff[128];
    MPI_Recv(buff, 128, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);

    int pos = 0;
    int m;
    MPI_Unpack(buff, 128, &pos, &m, 1, MPI_INT, MPI_COMM_WORLD);

    if (m == MESSAGE_TERMINATE)
        return NULL;

    data_t a;
    data_t b;
    unsigned long long n;
    MPI_Unpack(buff, 128, &pos, &a, 1, MPI_DATA_T,
   MPI_COMM_WORLD);
    MPI_Unpack(buff, 128, &pos, &b, 1, MPI_DATA_T,
   MPI_COMM_WORLD);
    MPI_Unpack(buff, 128, &pos, &n, 1, MPI_UNSIGNED_LONG_LONG,
   MPI_COMM_WORLD);

    return new Partition(a, b, n);
}
```

```
data_t reduce(data_t local_sum) {
    data_t global_sum;

    MPI_Allreduce(&local_sum, &global_sum, 1, MPI_DATA_T, MPI_SUM,
   MPI_COMM_WORLD);
    return global_sum;
}
```

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_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;
    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;
    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_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;
    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;
    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_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;
    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;
    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;
}
```