|  |  |
| --- | --- |
|  | **Правительство Российской Федерации**  **Федеральное государственное автономное образовательное учреждение высшего образования  «Национальный исследовательский университет «Высшая школа экономики»** |

**Отчёт по домашнему заданию №3 по курсу**

**«Высокопроизводительные Вычисления»**

**Тема работы: «Технология *MPI*»**

Выполнил:  
Забурунов Л. В., МСМТ221

(подпись)

Проверил(а):  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

6 декабря 2022 г.

г. Москва

Оглавление

[Цель работы 3](#_Toc119339297)

[Постановка задачи 3](#_Toc119339298)

[1. Решение первой задачи 4](#_Toc119339299)

[2. Переход к pinned памяти 7](#_Toc119339300)

[3. Разбиение на асинхронные потоки 8](#_Toc119339301)

[4. Использование блочной памяти в ядре 10](#_Toc119339302)

[5. Проверка работоспособности всех вариантов 12](#_Toc119339303)

[6. Анализ производительности различных вариантов 16](#_Toc119339304)

[7. Анализ влияния размера блока на производительность 17](#_Toc119339305)

[Выводы 18](#_Toc119339306)

[Приложение 1. Замеры производительности 19](#_Toc119339307)

[Приложение 2. Влияние размера блока 23](#_Toc119339308)

# Цель работы

Ознакомиться с особенностями реализации многопроцессорных программ при работе с технологией *MPI.*

# Постановка задачи

Требуется создать программу на языке *C/C++*, осуществляющую расчёт уравнения теплопроводности стержня. Уравнение должно быть решено с помощью разностной схемы, где каждый процесс подсчитывает свой участок стержня и использует механизм обмена сообщениями со смежными процессами для передачи и приёма температур на граничных точках.

Для программы нужно:

1. Провести проверку корректности решения, сравнив с точным решением, представленным в виде суммы ряда;
2. Провести анализ производительности для разных количеств процессов и точек.

# Создание программы

Во-первых, для программы понадобится множество констант (как математических, так и составляющих уравнения). Запишем их в отдельном файле *constants.h*:

#pragma once

#include <cmath>

constexpr double PI = 3.1415926535897932;

// Длина стержня (начинается в точке 0.0)

constexpr double ROD\_LENGTH = 1.0;

// Целевое значение времени

constexpr double TIME\_LENGTH = 1e-01;

// Начальная температура стержня

constexpr double ROD\_INITIAL\_TEMP = 1.0;

// Поддерживаемая температура среды

constexpr double ENVIROMENT\_TEMP = 0.0;

// К-т температуропроводности

constexpr double CONDUCTIVITY\_RATIO = 1.0;

// Шаг по пространству вдоль стержня

constexpr double SPATIAL\_STEP = 2e-02;

// Шаг по времени

constexpr double TIME\_STEP = 2e-04;

// Общее число подсчитываемых точек

constexpr int N\_POINTS = ROD\_LENGTH / SPATIAL\_STEP + 1;

// Коэффициент, используемый для конечно-разностной схемы

constexpr double EQUATION\_RATIO = (CONDUCTIVITY\_RATIO \* TIME\_STEP) / (SPATIAL\_STEP \* SPATIAL\_STEP);

Далее, нам требуется вычисление температуры стержня в требуемое время двумя способами: с помощью конечно-разностной схемы и технологии *MPI* и с помощью подсчёта суммы функционального ряда:

// Функция подсчёта температуры куска стержня в MPI-реализации

void SolveRodPiece(double startPoint, double endPoint, int processRank, int amountOfProcessors);

// Функция подсчёта температуры стержня с помощью вычисления суммы ряда

double\* CalculateRodWithSeries(int seriesMembersToCalculate);

Функция подсчёта суммы ряда достаточно проста в реализации; более того, здесь можно применить технологию *OpenMP* для ускорения вычислений:

double\* CalculateRodWithSeries(int seriesMembersToCalculate) {

int spatialSteps = static\_cast<int>(round(ROD\_LENGTH / SPATIAL\_STEP)) + 1;

double\* result = static\_cast<double\*>(calloc(spatialSteps, sizeof(double)));

constexpr double expConstRatio = -CONDUCTIVITY\_RATIO \* PI \* PI \* TIME\_LENGTH / (ROD\_LENGTH \* ROD\_LENGTH);

int i;

//#pragma omp parallel for private(i)

for (i = 0; i < spatialSteps; i = i + 1) {

double point = i \* SPATIAL\_STEP;

double value = 0;

#pragma omp parallel for reduction(+:value)

for (int j = 0; j < seriesMembersToCalculate; j = j + 1) {

double seriesMemberRatio = 2 \* j + 1;;

value += exp(expConstRatio \* seriesMemberRatio \* seriesMemberRatio) \* sin(PI \* seriesMemberRatio \* point / ROD\_LENGTH) / seriesMemberRatio;

}

result[i] = value \* 4 \* ROD\_INITIAL\_TEMP / PI;

}

return result;

}

Функция подсчёта через конечно-разностную схему более сложная. На вход принимается отрезок стержня, который мы должны подсчитать, а также сведения об *MPI-*конфигурации: номер процесса, для которого вызвали функцию, и общее число процессов. Особенная обработка требуется для самого левого и самого правого участка стержней, где вместо рассылки сообщений о температурах может использоваться значение температуры окружающей среды:

bool isInLeftRodEdge = processRank == 0,

isInRightRodEdge = processRank == amountOfProcessors - 1;

Конечно-разностная схема предполагает использование чисел, полученных для предыдущего момента времени, в подсчётах для текущего. Это означает, что нам достаточно хранить в памяти не все числа, а только те, что относятся к текущему и предыдущему моментам. Это позволит также использовать программу для широкого диапазона значений констант, поскольку затраты по памяти не зависят от длины временного интервала.

Значения для начального момента времени предварительно инициализируются исходными данными. При этом, для поддержания обобщённости архитектуры мы сразу же рассылаем сообщения о граничных значениях:

double \*newData = static\_cast<double\*>(calloc(2 \* spatialSteps, sizeof(double)));

// Начальные значения инициализируем вручную

for (int i = 0; i < spatialSteps; i = i + 1) {

newData[i] = ROD\_INITIAL\_TEMP;

}

MPI\_Request request;

// Рассылаем начальные значения (в цикле на них тоже идёт запрос)

if (!isInLeftRodEdge) {

MPI\_Send(newData, 1, MPI\_DOUBLE, processRank - 1, 10, MPI\_COMM\_WORLD);

//printf("Proc #%d sent data (%e) on time %f at point %f\n", processRank + 1, \*newData, 0.0f, startPoint);

}

if (!isInRightRodEdge) {

MPI\_Send(newData + spatialSteps - 1, 1, MPI\_DOUBLE, processRank + 1, 11, MPI\_COMM\_WORLD);

//printf("Proc #%d sent data (%e) on time %f at point %f\n", processRank + 1, \*(newData + spatialSteps - 1), 0.0f, endPoint);

}

Далее запускается основной цикл, где граничные значения обрабатываются особенно, поскольку требуется получение сообщения от соседнего процесса:

// Внешний цикл - по времени, внутренний - по пространству

// Это допускается, поскольку есть прямая зависимость между результатами для одной точки в разное время,

// но нет прямой зависимости между результатами для соседних точек в одно время

for (int i = 1; i < timeSteps; i = i + 1) {

int currentTimeBaseIndex = i \* spatialSteps,

lastTimeBaseIndex = (i - 1) \* spatialSteps;

// j = 0

double our = data[0], left, right = data[1];

if (isInLeftRodEdge) {

left = ENVIROMENT\_TEMP;

} else {

// Нумеровать сообщения будем следующим образом:

// 1. 10 - это чтобы освободить первые номера под общение с главным процессом;

// 2. i \* 2 - это смещение, однозначно определяющее значение времени, к которому относятся данные;

// 3. + 0 - это левая граница интервала, + 1 - правая

// Наша левая граница - это правая граница процесса с номером на 1 меньше

// То есть, 10 + (i - 1) \* 2 + 1 - это запрос к процессу, считающему отрезок левее нашего, на температуру его правой границы отрезка в предыдущий момент времени

MPI\_Status status;

MPI\_Recv(&left, 1, MPI\_DOUBLE, processRank - 1, 10 + (i - 1) \* 2 + 1, MPI\_COMM\_WORLD, &status);

//printf("Proc #%d received data (%e) on time %f at point %f\n", processRank + 1, leftData, i \* TIME\_STEP, startPoint + j \* SPATIAL\_STEP);

}

data[spatialSteps] = our + EQUATION\_RATIO \* (left - 2 \* our + right);

if (!isInLeftRodEdge) {

MPI\_Send(data + spatialSteps, 1, MPI\_DOUBLE, processRank - 1, 10 + i \* 2, MPI\_COMM\_WORLD);

//printf("Proc #%d sent data (%e) on time %f at point %f\n", processRank + 1, \*(newData + spatialSteps + j), i \* TIME\_STEP, startPoint + j \* SPATIAL\_STEP);

}

// loop

//#pragma omp parallel for

for (int j = 1; j < spatialSteps - 1; j = j + 1) {

data[spatialSteps + j] = data[j] + EQUATION\_RATIO \* (data[j - 1] - 2 \* data[j] + data[j + 1]);

}

// j = spatialSteps - 1

our = data[spatialSteps - 1], left = data[spatialSteps - 2], right;

if (isInRightRodEdge) {

right = ENVIROMENT\_TEMP;

} else {

MPI\_Status status;

MPI\_Recv(&right, 1, MPI\_DOUBLE, processRank + 1, 10 + (i - 1) \* 2, MPI\_COMM\_WORLD, &status);

//printf("Proc #%d received data (%e) on time %f at point %f\n", processRank + 1, leftData, i \* TIME\_STEP, startPoint + j \* SPATIAL\_STEP);

}

data[2 \* spatialSteps - 1] = our + EQUATION\_RATIO \* (left - 2 \* our + right);

if (!isInRightRodEdge) {

MPI\_Send(data + 2 \* spatialSteps - 1, 1, MPI\_DOUBLE, processRank + 1, 10 + i \* 2 + 1, MPI\_COMM\_WORLD);

//printf("Proc #%d sent data (%e) on time %f at point %f\n", processRank + 1, \*(newData + spatialSteps + j), i \* TIME\_STEP, startPoint + j \* SPATIAL\_STEP);

}

// После рассылки данных можно сдвинуть массив влево, чтобы предыдущие данные удалить (мы уже всё разослали в блокирующем виде), а текущие превратить в предыдущие и пойти дальше

double\* updatedData = static\_cast<double\*>(calloc(2 \* spatialSteps, sizeof(double)));

for (int k = 0; k < spatialSteps; k = k + 1) {

updatedData[k] = data[spatialSteps + k];

}

free(data);

data = updatedData;

}

В конце мы отсылаем главному процессу результат вычислений:

MPI\_Isend(data, spatialSteps, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD, &request);

Главный процесс в начале своей работы определяет границы интервалов для всех других процессов и рассылает данные для начала работы:

mpiTime = MPI\_Wtime();

printf("MSMT221, Zaburunov Leonid V., High-Performance Computations, Homework #3.\n\n");

printf("Using %d processes for calculation.\nSetup: rod length = %f; target time is %e; total points = %d; equation ratio: %e.\n\n", amountOfProcessors, ROD\_LENGTH, TIME\_LENGTH, N\_POINTS, EQUATION\_RATIO);

// Для каждого процессора формируем по два числа: номер первой точки и их число

int\* processorPointsInfo = static\_cast<int\*>(calloc(amountOfProcessors \* 2 - 2, sizeof(int)));

int div = N\_POINTS / amountOfProcessors,

mod = N\_POINTS % amountOfProcessors;

pointsInfo[0] = 0;

pointsInfo[1] = div + (mod > 0 ? 1 : 0);

processorPointsInfo[0] = pointsInfo[1];

processorPointsInfo[1] = div + (mod > 1 ? 1 : 0);

for (int i = 2; i < amountOfProcessors; i = i + 1) {

processorPointsInfo[(i - 1) \* 2] = processorPointsInfo[(i - 2) \* 2] + processorPointsInfo[(i - 2) \* 2 + 1];

processorPointsInfo[(i - 1) \* 2 + 1] = div + (i <= mod - 1 ? 1 : 0);

}

printf("Calculating points [%f; %f] with step %f on processor 1\n", 0.0f, SPATIAL\_STEP \* (pointsInfo[1] - 1), SPATIAL\_STEP);

printf("Calculating points [%f; %f] with step %f on processor 2\n", SPATIAL\_STEP \* processorPointsInfo[0], SPATIAL\_STEP \* (processorPointsInfo[0] + processorPointsInfo[1] - 1), SPATIAL\_STEP);

for (int i = 2; i < amountOfProcessors - 1; i = i + 1) {

double firstPoint = SPATIAL\_STEP \* static\_cast<float>(processorPointsInfo[(i - 1) \* 2]),

secondPoint = firstPoint + SPATIAL\_STEP \* static\_cast<float>(processorPointsInfo[(i - 1) \* 2 + 1] - 1);

printf("Calculating points [%f; %f] with step %f on processor %d\n", firstPoint, secondPoint, SPATIAL\_STEP, i + 1);

}

printf("Calculating points [%f; %f] with step %f on processor %d\n",

SPATIAL\_STEP \* processorPointsInfo[amountOfProcessors \* 2 - 4],

SPATIAL\_STEP \* (processorPointsInfo[amountOfProcessors \* 2 - 4] + processorPointsInfo[amountOfProcessors \* 2 - 3] - 1),

SPATIAL\_STEP, amountOfProcessors);

// Всем процессам, кроме себя самого, рассылаем сообщения

for (int i = 1; i < amountOfProcessors; i = i + 1) {

MPI\_Send(processorPointsInfo + (i - 1) \* 2, 2, MPI\_INT, i, 0, MPI\_COMM\_WORLD);

}

Далее – ожидание результата и вывод на консоль итоговых значений:

int totalLength = 0;

mpiData = static\_cast<double\*>(calloc(0, sizeof(double)));

for (int i = 0; i < amountOfProcessors; i = i + 1) {

MPI\_Status status;

MPI\_Probe(i, 1, MPI\_COMM\_WORLD, &status);

int count;

MPI\_Get\_count(&status, MPI\_DOUBLE, &count);

double\* procResult = static\_cast<double\*>(calloc(count, sizeof(double)));

MPI\_Recv(procResult, count, MPI\_DOUBLE, i, 1, MPI\_COMM\_WORLD, &status);

double\* newResults = static\_cast<double\*>(calloc(totalLength + count, sizeof(double)));

for (int i = 0; i < totalLength; i = i + 1) {

newResults[i] = mpiData[i];

}

free(mpiData);

for (int i = 0; i < count; i = i + 1) {

newResults[totalLength + i] = procResult[i];

}

free(procResult);

mpiData = newResults;

totalLength = totalLength + count;

}

mpiTime = MPI\_Wtime() - mpiTime;

seriesTime = MPI\_Wtime();

seriesData = CalculateRodWithSeries(500);

seriesTime = MPI\_Wtime() - seriesTime;

printf("\n-----------------------------\n\n");

printf("MPI version results\n\n");

for (int i = 0; i < N\_POINTS; i = i + 1) {

printf("Point %f has temp %e\n", i \* SPATIAL\_STEP, mpiData[i]);

}

printf("\n-----------------------------\n\n");

printf("Precise calculation results\n\n");

for (int i = 0; i < N\_POINTS; i = i + 1) {

printf("Point %f has temp %e\n", i \* SPATIAL\_STEP, seriesData[i]);

}

printf("\n-----------------------------\n\n");

printf("Element-wise difference\n\n");

for (int i = 0; i < N\_POINTS; i = i + N\_POINTS / 40) {

printf("Point %f - %e\n", i \* SPATIAL\_STEP, abs(seriesData[i] - mpiData[i]));

}

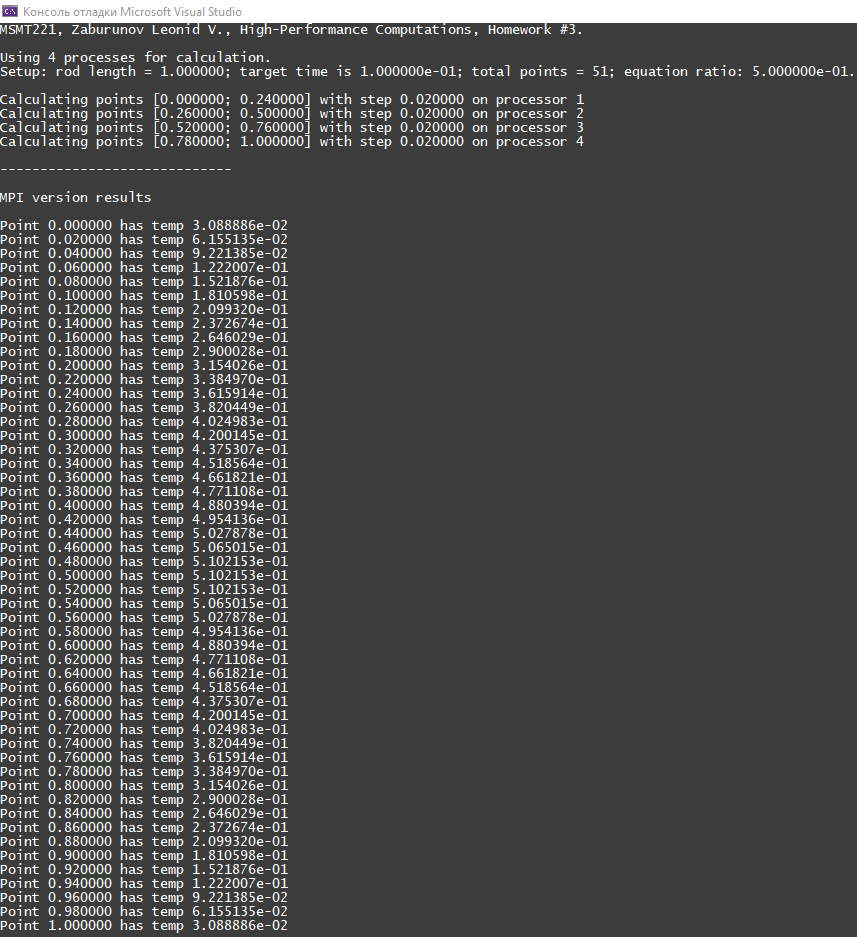
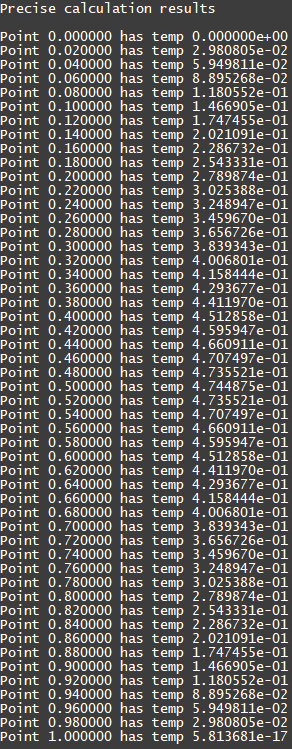
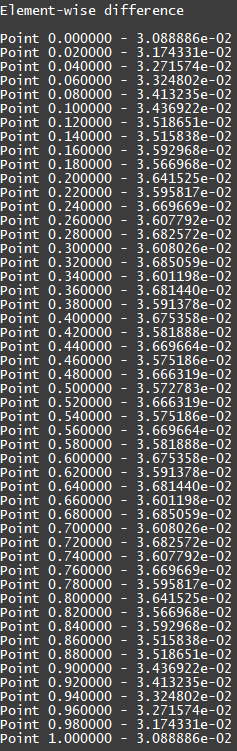
printf("\n-----------------------------\n\n");

printf("MPI version time complexity:%e\nPrecise version time complexity: %e\n", mpiTime, seriesTime);

# Проверка точности решения

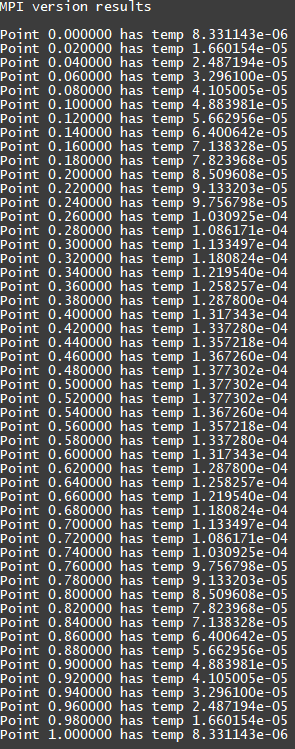
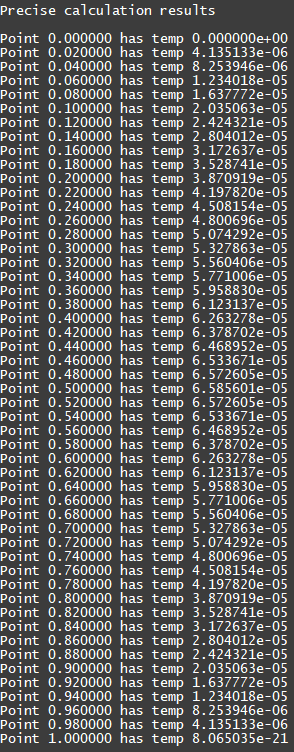
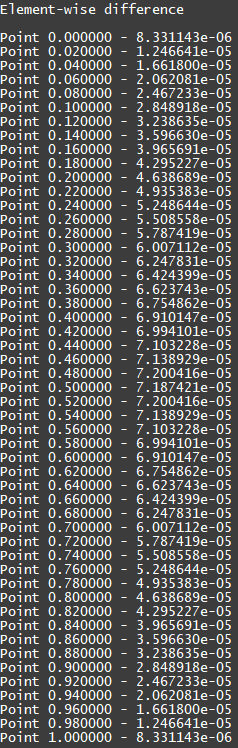
В программу был добавлен блока определения поэлементной разности решений на базе конечно-разностной схемы и функционального ряда. По заданию требуется провести проверку точности для следующих значений:

Результаты работы:

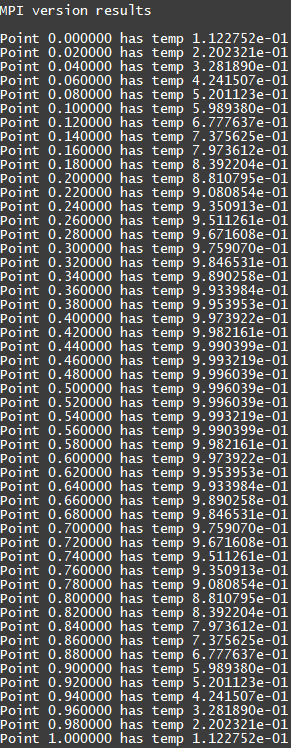
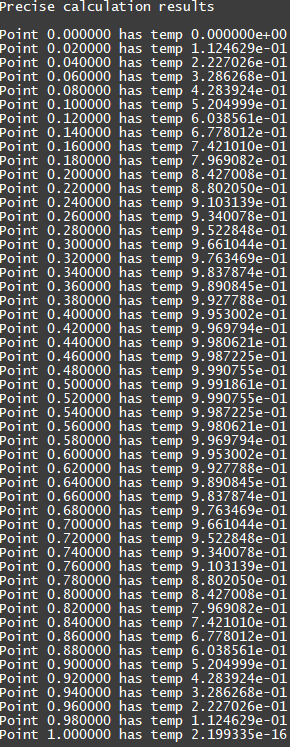
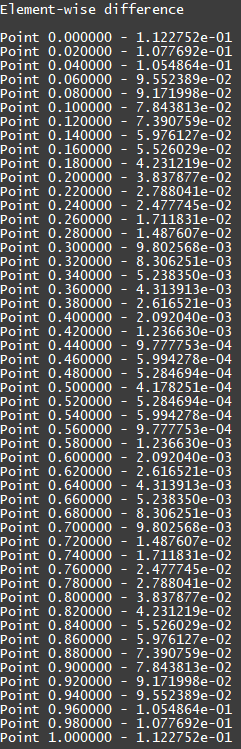
  

Посмотрим на зависимость погрешности от шагов по времени и по стержню.

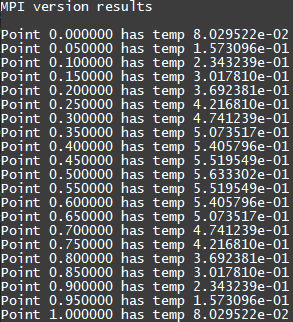
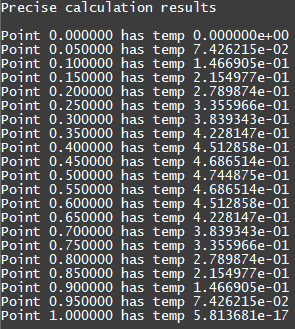
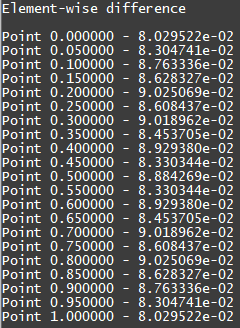
Приведём два примера для времени: в первом время изменено с 0.1 на 1, во втором – с 0.1 на 0.01. Результаты работы в первом случае:

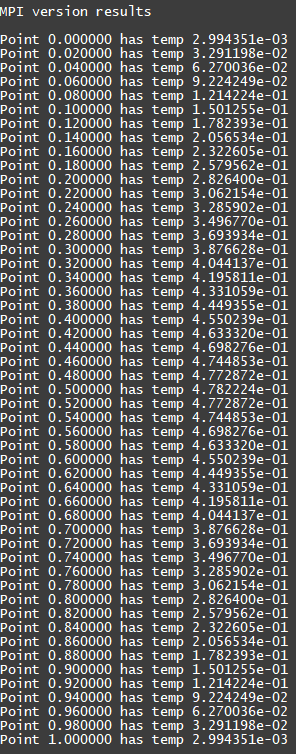
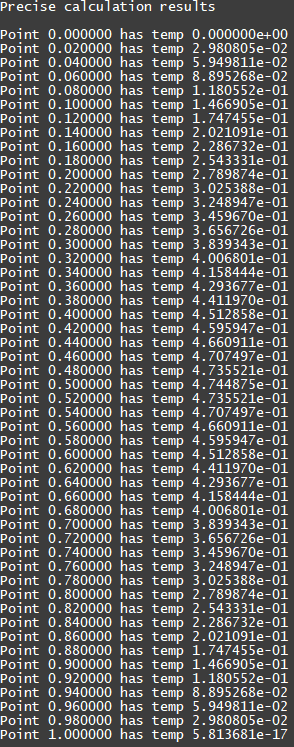
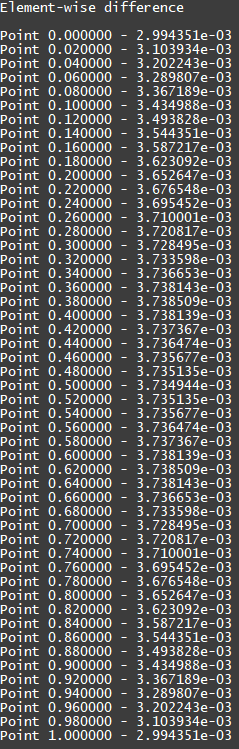
Результаты работы во втором случае:

Теперь пример по пространству. Заменим шаг вдоль стержня с 0.02 на 0.05:

Теперь заменим шаг по пространству на 0.002, но также и уменьшим шаг по времени, чтобы соблюдать соотношение для конечно-разностной схемы. Для вывода в консоль идут не все значения, а 50 с равномерным шагом вдоль стержня, поскольку исходных точек слишком много:

# Оценка производительности

Требуется провести анализ производительности для различных конфигураций: 1, 2, 4, 8, 12, 16 и 24 создаваемых процесса для 10000, 25000 и 50000 точек на стержне.

Ниже будут приведены результаты замеров производительности. В приложении будут представлены графики.

:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| 1 | 3,00E+01 | 3,00E+01 | 3,01E+01 | 3,00E+01 | 3,01E+01 | 30,057506 |
| 2 | 1,52E+01 | 1,52E+01 | 1,52E+01 | 1,53E+01 | 1,52E+01 | 15,225112 |
| 4 | 7,78E+00 | 7,77E+00 | 7,77E+00 | 7,78E+00 | 7,79E+00 | 7,7776678 |
| 8 | 4,05E+00 | 4,06E+00 | 4,06E+00 | 4,08E+00 | 4,07E+00 | 4,0644126 |
| 12 | 3,12E+00 | 3,10E+00 | 3,12E+00 | 3,09E+00 | 3,09E+00 | 3,103866 |
| 16 | 2,70E+00 | 2,71E+00 | 2,72E+00 | 2,71E+00 | 2,70E+00 | 2,7071986 |
| 24 | 1,98E+00 | 2,02E+00 | 1,99E+00 | 2,00E+00 | 2,01E+00 | 2,0006978 |

:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| 1 | 7,51E+01 | 7,50E+01 | 7,51E+01 | 7,49E+01 | 7,50E+01 | 75,035566 |
| 2 | 3,78E+01 | 3,78E+01 | 3,78E+01 | 3,78E+01 | 3,79E+01 | 37,82145 |
| 4 | 1,90E+01 | 1,91E+01 | 1,91E+01 | 1,91E+01 | 1,91E+01 | 19,07864 |
| 8 | 9,70E+00 | 9,70E+00 | 9,70E+00 | 9,73E+00 | 9,70E+00 | 9,7059376 |
| 12 | 7,27E+00 | 7,22E+00 | 7,33E+00 | 7,55E+00 | 7,75E+00 | 7,4220938 |
| 16 | 6,11E+00 | 6,14E+00 | 6,15E+00 | 6,17E+00 | 6,11E+00 | 6,1360222 |
| 24 | 4,33E+00 | 4,35E+00 | 4,32E+00 | 4,33E+00 | 4,32E+00 | 4,3323082 |

:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество процессов | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| 1 | 1,56E+02 | 1,57E+02 | 1,57E+02 | 1,55E+02 | 1,56E+02 | 156,2187 |
| 2 | 7,55E+01 | 7,53E+01 | 8,03E+01 | 7,76E+01 | 7,76E+01 | 77,24829 |
| 4 | 3,78E+01 | 3,78E+01 | 3,78E+01 | 3,79E+01 | 3,78E+01 | 37,79242 |
| 8 | 1,91E+01 | 1,91E+01 | 1,91E+01 | 1,91E+01 | 1,92E+01 | 19,13914 |
| 12 | 1,46E+01 | 1,51E+01 | 1,48E+01 | 1,46E+01 | 1,42E+01 | 14,65517 |
| 16 | 1,18E+01 | 1,18E+01 | 1,18E+01 | 1,18E+01 | 1,18E+01 | 11,80465 |
| 24 | 8,10E+00 | 8,03E+00 | 8,03E+00 | 8,11E+00 | 8,08E+00 | 8,071897 |

# Внедрение коллективных операций

# Выводы

В ходе выполнения домашнего задания была освоена работа с многопроцессорными программами и технологией *MPI*. Анализ производительности в данной задаче даёт стандартную картину: линейный рост, переходящий в выход на плато. Это связано с тем, что задача имеет большой объём данных для вычисления и имеет смысл проводить вычисления частей асинхронно, однако задача не настолько громоздкая, чтобы имело смысл создавать несколько десятков процессов.

# Приложение