Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт

**Высшая школа «Теоретическая механика»**

**КУРСОВАЯ РАБОТА**

по дисциплине «Проектирование по компьютерным технологиям в механике»



Выполнил

студент гр.5040103/00301 С. С. Барсуков

Руководитель

<*подпись*> А. А. Ле-Захаров

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 202\_\_ г.

Санкт-Петербург

2021

Содержание

[Введение 3](#_Toc87648712)

[Основные функции MPI 4](#_Toc87648712)

[Вычисление интеграла 5](#_Toc87648713)

[Вычисление числа Пи 8](#_Toc87648714)

[Решение одномерной задачи теплопроводности 11](#_Toc87648715)

[Заключение 16](#_Toc87648716)

[Список использованной литературы 17](#_Toc87648717)

# Введение

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс (API) для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан Уильямом Гроуппом и другими [1].

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Используется при разработке программ для кластеров и суперкомпьютеров [2]. Основным средством коммуникации между процессами в MPI является передача сообщений друг другу. Параллельная программа должна эффективно использовать вычислительные мощности и коммуникационную среду. В MPI вся работа по распределению нагрузки на узлы и сеть ложатся на программиста и для максимальной производительности необходимо знать особенности конкретного кластера.

Идея распараллеливания программ состоит в том, чтобы ускорить их работу, не повлияв на точность получаемых результатов. Достигается распараллеливание путем использования двух и более процессоров/ядер в комбинации для решения одной задачи [3].

В ходе данной работы будут решены задачи: вычисления определенного интеграла, вычисления числа Пи методом Монте-Карло и решения одномерной задачи теплопроводности. А также построены зависимости времени выполнения программы и коэффициента распараллеливания от числа процессоров.

# Основные функции MPI

В ходе реализации процесса распараллеливания были использованы следующие методы:

* MPI\_Init - Инициализация среды MPI.
* MPI\_Comm\_rank - Определение номера процессора.
* MPI\_Comm\_size - Количество задействованных процессоров.
* MPI\_Send - Функция, позволяющая отправлять полученный результат, которая на вход получает: адрес буфера, в который помещаются данные, размер буфера, тип ячейки буфера, номер процессора, с которым происходит обмен данными, идентификатор сообщения, описатель области связи.
* MPI\_Recv - Функция, позволяющая принимать сообщения от других процессоров, которая на вход получает: адрес буфера, из которого берутся данные, размер буфера, тип ячейки буфера, номер процессора, с которым происходит обмен данными, идентификатор сообщения, описатель области связи, статус завершения приёма.
* MPI\_Finalize - Деактивация среды MPI.

В ходе решения поставленных задач будет высчитана зависимость коэффициента эффективности распараллеливаемости от числа процессоров , которая определятся следующим образом:

Где – время вычисления на 1 процессоре, – время вычисления на n процессорах.

Величина эффективности определяет среднюю долю времени выполнения алгоритма, в течение которой процессоры реально задействованы для решения задачи.

# Вычисление интеграла

Необходимо посчитать определенный интеграл на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 8 процессорах). При 10000000 разбиениях. Каждый расчёт проведен 5 раз для получения усредненного результата времени вычисления. Каждый процессор будет работать на своём участке и передавать получившееся значение в процессор с номером “0“.

Функция для расчёта:

Код реализации приведен ниже:

#include <mpi.h>

#include <stdio.h>

#include <time.h>

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

clock\_t start, end;

start = clock();

MPI\_Init(NULL, NULL);

int my\_rank;

int my\_size;

float N = 10000000; // число разбиений

float a = 2;

float b = 6;

float result = 0;

float dx = (b - a) / N;

MPI\_Status status;

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

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

float n = N / my\_size;

if (my\_rank == 0) {

float ai = a + (b - a) \* my\_rank / my\_size;

float bi = a + (b - a) \* (my\_rank + 1) / my\_size;

for (int k = 0; k < n; k++) {

result += (2 \* (dx \* k + ai) + 4) \* dx; // Сам интеграл

}

for (int i = 1;i < my\_size;i++) {

float message;

MPI\_Recv(&message, 1, MPI\_FLOAT, i, 0, MPI\_COMM\_WORLD, &status);

result += message;

}

end = clock();

printf("%f\n", result);

double time\_taken = double(end - start) / double(CLOCKS\_PER\_SEC);

printf("Time used: %f", time\_taken);

}

else {

float message = 0;

float ai = a + (b - a) \* my\_rank / my\_size;

float bi = a + (b - a) \* (my\_rank + 1) / my\_size;

for (int k = 0; k < n; k++) {

message += (2 \* (dx \* k + ai) + 4) \* dx; // Сам интеграл

}

MPI\_Send(&message, 1, MPI\_FLOAT, 0, 0, MPI\_COMM\_WORLD);

}

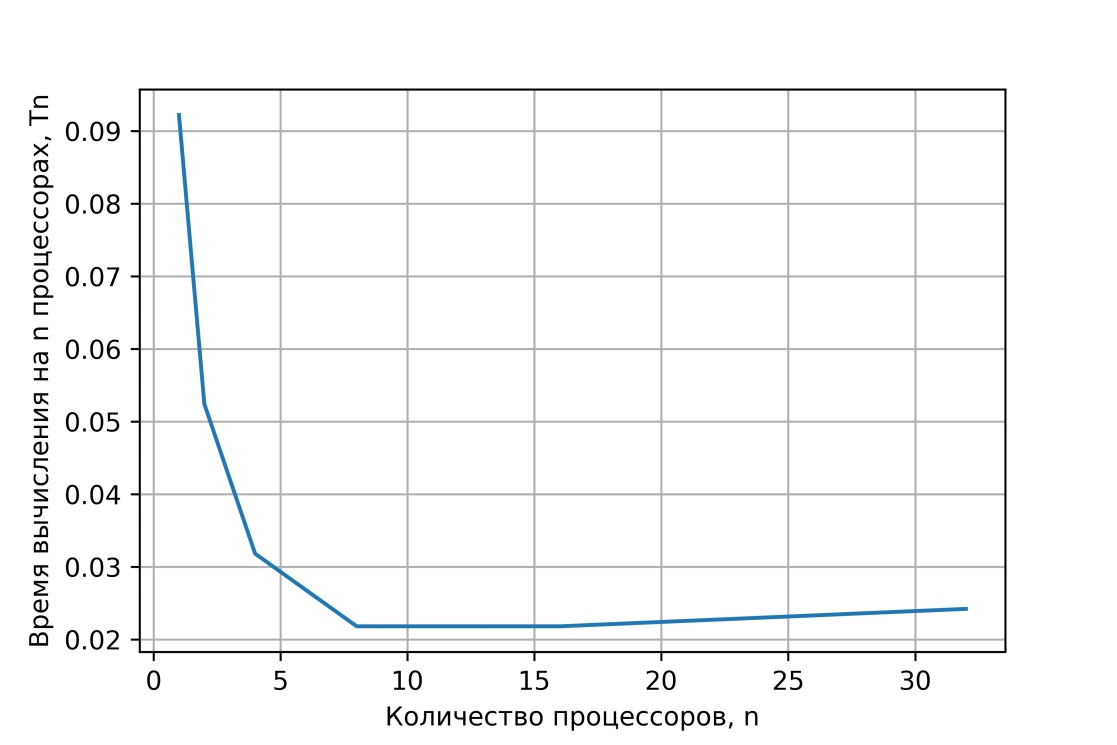
MPI\_Finalize();

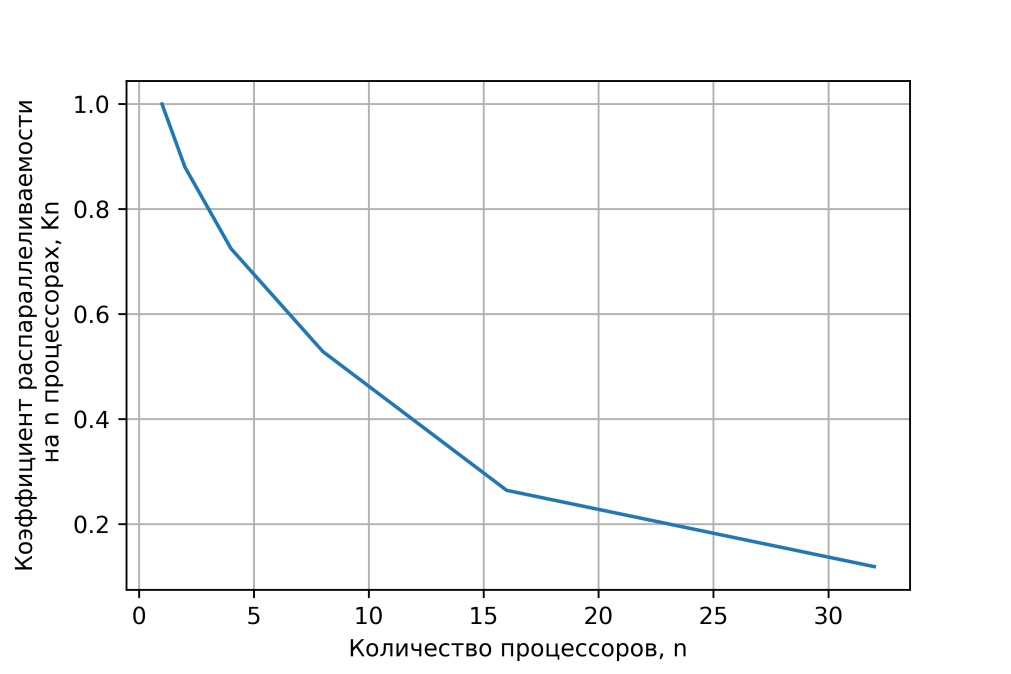
}

Полученные результаты приведены в таблице 1, а также на рис. 1 и рис. 2.

Таблица 1. Зависимость времени вычисления и коэффициента распараллеливания от количества процессоров

|  |  |  |  |
| --- | --- | --- | --- |
| Количество процессоров | Результат вычисления интеграла | Время вычисления | Коэф. распараллеливания |
| 1 | 50,29 | 0.0921 | 1.0000 |
| 2 | 47.09 | 0.0524 | 0.8797 |
| 4 | 48,36 | 0.0318 | 0.7248 |
| 8 | 47,95 | 0.0218 | 0.5286 |
| 16 | 48,02 | 0.0218 | 0.2643 |
| 32 | 48,01 | 0.0242 | 0.119 |

  
Рис. 1. Зависимость времени вычисления на n процессорах от количества процессоров

  
Рис. 2. Зависимость коэффициента распараллеливания на n процессорах от количества процессоров

Таким образом, при увеличении количества процессоров мы получаем приемлемый по точности результат (значение определенного интеграла равно 48) быстрее, нежели без использования распараллеливания, но только при распараллеливании до 8 процессоров, поскольку столько процессоров имеется всего.

# Вычисление числа Пи

Необходимо вычислить значение числа Пи методом Монте – Карло на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 8 процессорах). Для 10000000 точек. Каждый расчёт проведен 5 раз для получения усредненного результата времени вычисления. Каждый процессор будет работать на своём участке и передавать получившееся значение в процессор с номером “0“. Для решения этой задачи было рассмотрено отношение площадей квадрата и вписанного в него круга.

Таким образом, получаем:

Так, при большом количестве точек в численном эксперименте:

Где – количество точек попавших в круг, – общее количество точек.

Код реализации приведен ниже:

#include <mpi.h>

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <cmath>

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

clock\_t start, end;

start = clock();

MPI\_Init(NULL, NULL);

int my\_rank;

int my\_size;

int N = 10000000; // всего точек

double a = -0.5;

double b = 0.5;

double r = 0.5;

MPI\_Status status;

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

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

double dx = (b - a) / my\_size;

int n = N / my\_size;

if (my\_rank == 0) {

double ai = a + dx \* my\_rank;

double bi = a + dx \* (my\_rank + 1);

double x;

double y;

int point\_in\_circle = 0;

srand((unsigned)time(NULL) + my\_rank);

for (int k = 0; k < n; k++) {

x = ((double)rand() / RAND\_MAX)\* (bi - ai) + ai;

y = ((double)rand() / RAND\_MAX) \* (b - a) + a;

if (sqrt(pow(x, 2) + pow(y, 2)) <= r) {

point\_in\_circle += 1;

}

}

for (int i = 1;i < my\_size;i++) {

int message;

MPI\_Recv(&message, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, &status);

point\_in\_circle += message;

}

float number\_pi = (float) 4 \* point\_in\_circle / (n \* my\_size);

end = clock();

printf("%f\n", number\_pi);

double time\_taken = double(end - start) / double(CLOCKS\_PER\_SEC);

printf("Time used: %f", time\_taken);

}

else {

double ai = a + dx \* my\_rank;

double bi = a + dx \* (my\_rank + 1);

double x;

double y;

int point\_in\_circle = 0;

srand((unsigned)time(NULL) + my\_rank);

for (int k = 0; k < n; k++) {

x = ((double)rand() / RAND\_MAX) \* (bi - ai) + ai;

y = ((double)rand() / RAND\_MAX) \* (b - a) + a;

if (sqrt(pow(x, 2) + pow(y, 2)) <= r) {

point\_in\_circle += 1;

}

}

MPI\_Send(&point\_in\_circle, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

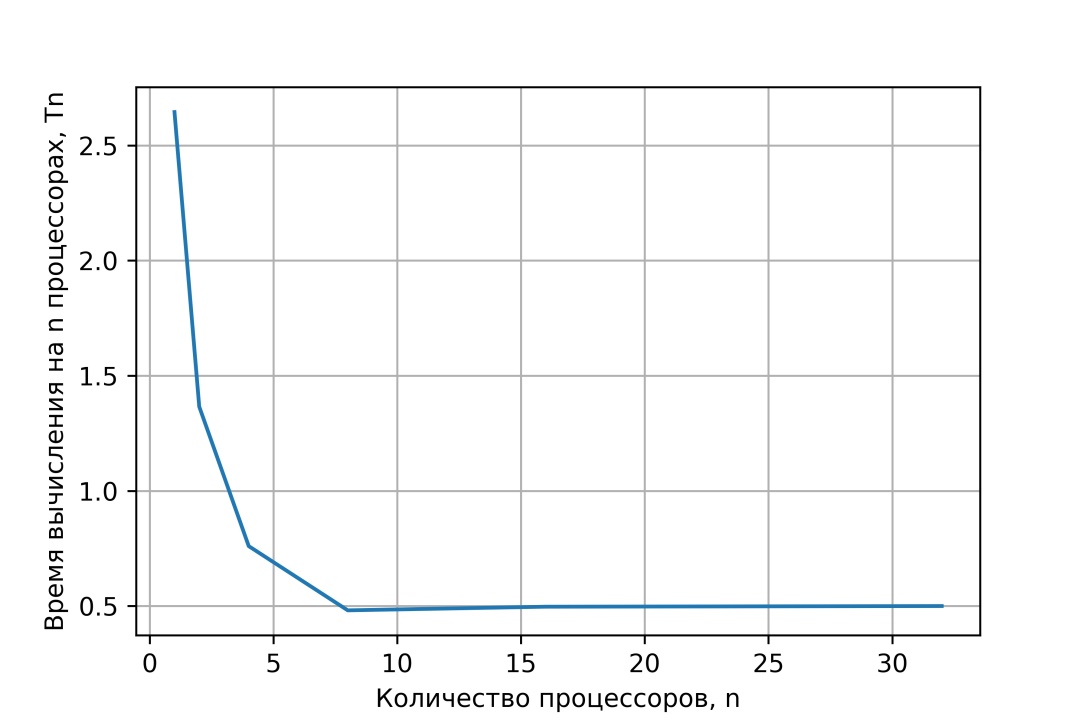
MPI\_Finalize();

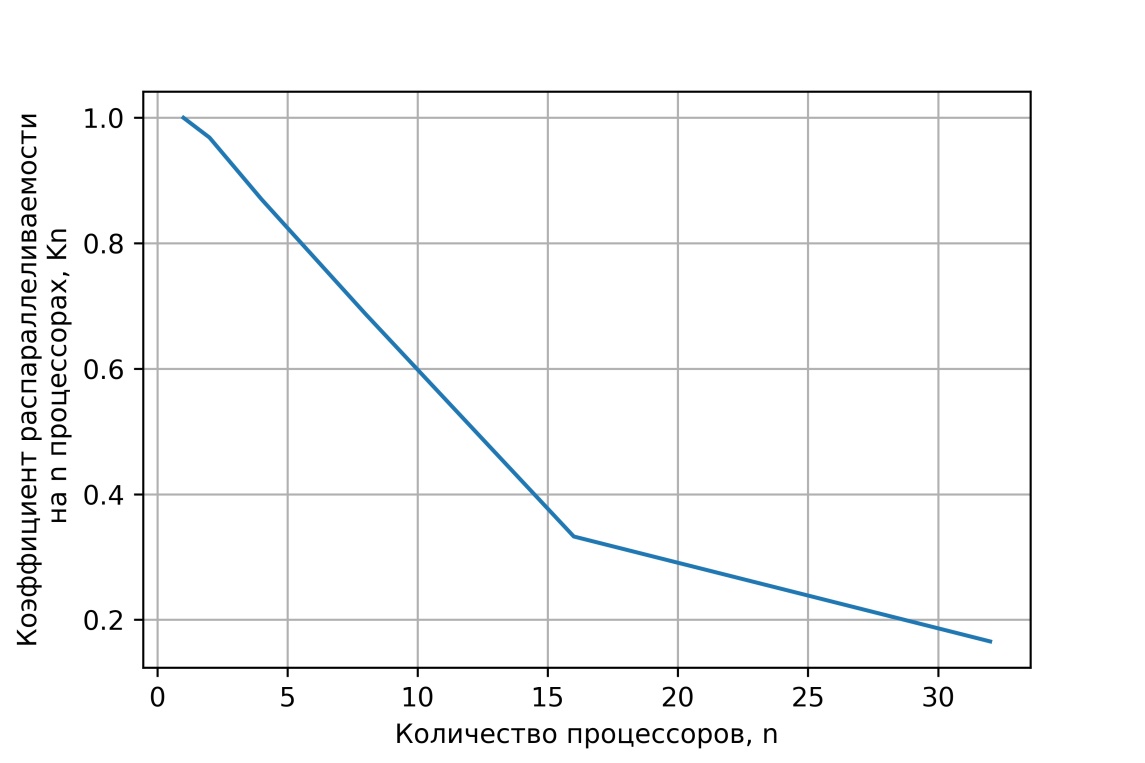
}

Полученные результаты приведены в таблице 2, а также на рис.3 и рис.4.

Таблица 2. Зависимость времени вычисления и коэффициента распараллеливания от количества процессоров

|  |  |  |  |
| --- | --- | --- | --- |
| Количество процессоров | Результат вычисления числа Пи | Время вычисления | Коэф. распараллеливания |
| 1 | 3.14168 | 2.645 | 1.0000 |
| 2 | 3.14087 | 1.3656 | 0.9684 |
| 4 | 3.14169 | 0.76 | 0.87 |
| 8 | 3.14172 | 0.4811 | 0.687 |
| 16 | 3.14114 | 0.497 | 0.3326 |
| 32 | 3.14086 | 0.4998 | 0.1653 |

  
Рис. 3. Зависимость времени вычисления на n процессорах от количества процессоров

  
Рис. 4. Зависимость коэффициента распараллеливания на n процессорах от количества процессоров

В итоге при увеличении количества процессоров мы получаем результат быстрее, чем при расчёте на 1 процессоре. При этом отклонение от числа Пи = 3,14159, полученное численным экспериментом небольшое.

# Решение одномерной задачи теплопроводности

Необходимо решить одномерное уравнение теплопроводности на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 8 процессорах):

При применении метода конечных разностей получаем уравнение:

где α – коэффициент температуропроводности, который равен:

Каждый расчёт проведен 5 раз для получения усредненного результата времени вычисления. Каждый процессор будет не только работать на своём участке и передавать получившееся значение в процессор с номером “0“, но еще и в каждый момент времени взаимодействовать с “соседними“ процессорами для передачи граничных значений. Решим задачу для стержня при следующих н.у. и г.у:

*,*

где – длина стержня, – шаг по пространству, – время моделирования, – шаг по времени, – температура.

Код реализации приведен ниже:

#include <mpi.h>

#include <stdio.h>

#include <math.h>

#include <iostream>

#include <fstream>

#include <time.h>

using namespace std;

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

{

int my\_rank;

int my\_size;

MPI\_Status status;

clock\_t start, end;

start = clock();

MPI\_Init(NULL, NULL);

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

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

double l = 1; // Длина стержня

double h = 0.01; // Шаг по пространству

double T = 10; // Время моделирования

double dt = 0.001; // Шаг по времени

double alpha = 1.172 \* pow(10, -2); // Коэффициент температуропроводности

int N = 1 + (l / h); // Количество точек

int K = 1 + (T / dt); // Количество шагов по времени

double T0 = 1; // Левая граница

double T1 = 0.5403; // Правая граница

double\* T\_i = new double[N];

double\* T\_i\_next = new double[N];

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

T\_i[i] = cos(i \* h);

T\_i\_next[i] = cos(i \* h);

}

T\_i[0] = 1;

T\_i\_next[0] = 1;

T\_i[N-1] = 0.5403;

T\_i\_next[N-1] = 0.5403;

int ai = ((N - 1) \* my\_rank) / my\_size + 1;

int batch\_size = (N - 1) / my\_size;

int bi;

if (my\_rank == my\_size - 1) {

bi = N - 1 - 1;

}

else {

bi = ai + batch\_size - 1;

}

for (int t = 1; t <= K; t++) {

if (my\_rank > 0) {

MPI\_Recv(&T\_i\_next[ai - 1], 1, MPI\_DOUBLE, my\_rank - 1, 1, MPI\_COMM\_WORLD, &status);

}

if (t > 1 && my\_rank < my\_size - 1) {

MPI\_Recv(&T\_i\_next[bi + 1], 1, MPI\_DOUBLE, my\_rank + 1, 2, MPI\_COMM\_WORLD, &status);

}

for (int i = ai;i <= bi;i++) {

T\_i\_next[i] = (alpha \* (T\_i[i + 1] - 2 \* T\_i[i] + T\_i[i - 1]) \* dt) / (h \* h) + T\_i[i];

}

for (int i = ai - 1;i <= bi + 1;i++) {

T\_i[i] = T\_i\_next[i];

}

if (my\_rank < my\_size - 1) {

MPI\_Send(&T\_i\_next[bi], 1, MPI\_DOUBLE, my\_rank + 1, 1, MPI\_COMM\_WORLD);

}

if (t < K && my\_rank > 0) {

MPI\_Send(&T\_i\_next[ai], 1, MPI\_DOUBLE, my\_rank - 1, 2, MPI\_COMM\_WORLD);

}

}

if (my\_rank > 0) {

MPI\_Send(&T\_i\_next[ai], bi - ai + 1, MPI\_DOUBLE, 0, 3, MPI\_COMM\_WORLD);

}

if (my\_rank == 0) {

for (int i = 1;i < my\_size;i++) {

MPI\_Recv(&T\_i[i \* (bi - ai + 1) + 1], bi - ai + 1, MPI\_DOUBLE, i, 3, MPI\_COMM\_WORLD, &status);

}

end = clock();

double time\_taken = double(end - start) / double(CLOCKS\_PER\_SEC);

printf("Time used: %f", time\_taken);

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

printf("%f ",T\_i[i]);

}

}

MPI\_Finalize();

}

Полученные результаты приведены в таблице 3, а также на рис. 5, рис. 6 и рис. 7.

Таблица 3. Зависимость времени вычисления и коэффициента распараллеливания от количества процессоров

|  |  |  |
| --- | --- | --- |
| Количество процессоров | Время вычисления | Коэф. распараллеливания |
| 1 | 0.0172 | 1.0000 |
| 2 | 0.0111 | 0.7678 |
| 4 | 0.0072 | 0.5922 |
| 8 | 0.0047 | 0.4497 |
| 16 | 0.0047 | 0.2287 |
| 32 | 0.0048 | 0.1173 |

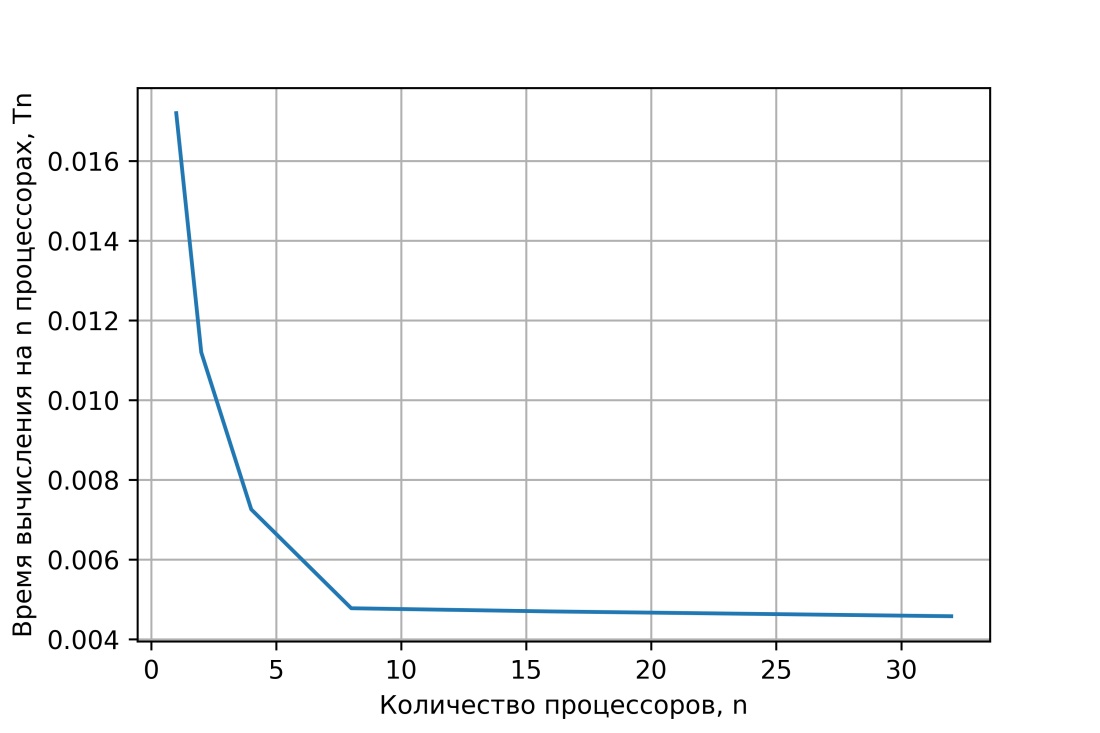
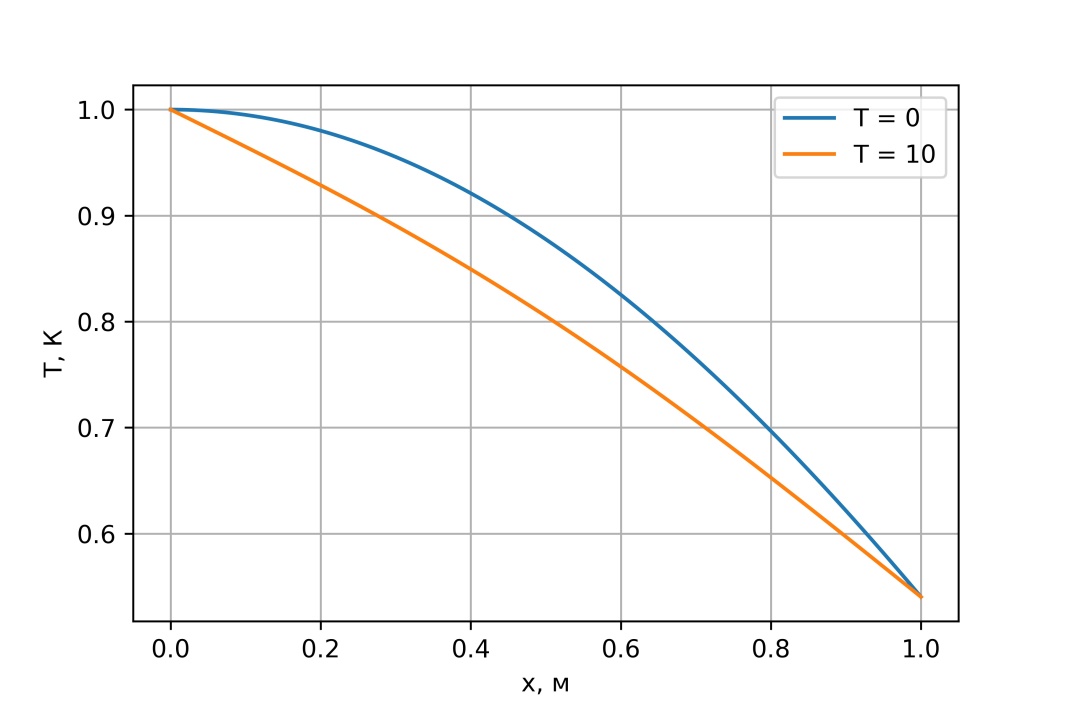
  
Рис. 5. Зависимость времени вычисления на n процессорах от количества процессоров

  
Рис. 6. Зависимость коэффициента распараллеливания на n процессорах от количества процессоров

  
Рис. 7. Зависимость температуры от координаты при t = 0, t = 10 при расчёте на 4х процессорах

Как видно из рис. 5, рис. 6 и рис. 7: время расчёта при увлечении количества процессоров уменьшается (до имеющегося количества процессоров), результаты моделирования для 4 процессоров правильно отображают закон теплопроводности.

Заключение

Таким образом, был приобретен навык распараллеливания задачи на процессы, которые могут, как взаимодействовать друг с другом, так и работать независимо. В результате выполнения трёх задач были получены результаты, которые приемлемы по точности, при этом при увеличении количества процессоров (до имеющихся 8) время моделирования уменьшалось, но, как видно из зависимостей коэффициента распараллеливания от количества процессоров, время уменьшалось нелинейно, поскольку происходит трата времени на инициализацию процессоров, а также на суммирование результатов.

# Список использованной литературы

1. Gropp W. et al. Using MPI: portable parallel programming with the message-passing interface. – MIT press, 1999. – Т. 1.
2. Pacheco P. Parallel programming with MPI. – Morgan Kaufmann, 1997.
3. Snir M. et al. MPI--the Complete Reference: the MPI core. – MIT press, 1998. – Т. 1.