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 <*подпись*> В. Д. Вараев

Руководитель

<*подпись*> А. А. Ле-Захаров

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

Санкт-Петербург

2021

Содержание

[Введение 3](#_Toc87648712)

[MPI функции 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) — библиотека функций, предназначенная для поддержки работы параллельных процессов в терминах передачи сообщений.

MPI является наиболее распространённым стандартом интерфейса обмена данными в параллельном программировании, существуют его реализации для большого числа компьютерных платформ. Номер процесса - целое неотрицательное число, являющееся уникальным атрибутом каждого процесса. Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура MPI\_Status, содержащая три поля: MPI\_Source (номер процесса отправителя), MPI\_Tag (идентификатор сообщения), MPI\_Error (код ошибки); могут быть и добавочные поля. Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.Процессы объединяются в группы, могут быть вложенные группы. Внутри группы все процессы перенумерованы. С каждой группой ассоциирован свой коммуникатор. Поэтому при осуществлении пересылки необходимо указать идентификатор группы, внутри которой производится эта пересылка. Все процессы содержатся в группе с предопределенным идентификатором MPI\_COMM\_WORLD.

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 были 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.

# MPI функции

В ходе реализации процесса распараллеливания были использованы следующие методы:

* MPI\_Init - инициализация параллельной части приложения. Реальная инициализация для каждого приложения выполняется не более одного раза, а если MPI уже был инициализирован, то никакие действия не выполняются и происходит немедленный возврат из подпрограммы. Все оставшиеся MPI-процедуры могут быть вызваны только после вызова MPI\_Init.
* MPI\_Comm\_rank - Определение номера процесса в группе *comm*. Значение, возвращаемое по адресу *&rank*, лежит в диапазоне от 0 до *size\_of\_group-1*
* MPI\_Comm\_size - Определение общего числа параллельных процессов в группе *comm*.
* 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 - Прием сообщения с идентификатором *msgtag* от процесса *source*. Число элементов в принимаемом сообщении не должно превосходить значения *count*. Если число принятых элементов меньше значения *count*, то гарантируется, что в буфере *buf* изменятся только элементы, соответствующие элементам принятого сообщения. Если нужно узнать точное число элементов в сообщении, то можно воспользоваться подпрограммой *MPI\_Probe*.
* MPI\_Finalize - завершение параллельной части приложения. Все последующие обращения к любым MPI-процедурам, в том числе к MPI\_Init, запрещены. К моменту вызова MPI\_Finalize некоторым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

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

Будем проводить вычисления определенного интеграла на 1, 2, 4, 8, 16 и 32 процессорах (при реально имеющихся 4 процессорах). При 10000000 разбиениях. Каждый процессор будет работать на своём отрезке 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>

#include <string>

#include <cstdlib>

#include <iostream>

using namespace std;

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

clock\_t start = clock();

MPI\_Init(NULL, NULL);

int my\_rank;

float N = 1000;

int p;

float a = 2;

float b = 6;

float ai;

float bi;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); // ,pi\_comm\_rank записывает значение в my\_rank, & - записывает в my\_rank

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

float n = N / p;

MPI\_Status status;

float integral\_value = 0;

if (my\_rank == 0)

{

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

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

float dx = (bi - ai) / n;

for (int i = 0; i < n; i++)

{

integral\_value += (2 \* (i \* dx + ai) + 4) \* dx;

}

for (int i = 1; i < p; i++)

{

float result;

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

integral\_value += result;

}

printf("\n %f", integral\_value);

clock\_t end = clock();

double cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("\n %f", cpu\_time\_used);

}

else

{

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

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

float dx = (bi - ai) / n;

float result = 0;

for (int i = 0; i < n; i ++)

{

result += (2 \*(i\*dx+ai) + 4)\*dx;

}

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

}

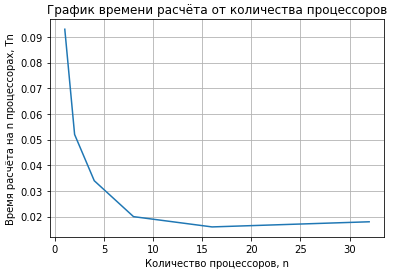
MPI\_Finalize();

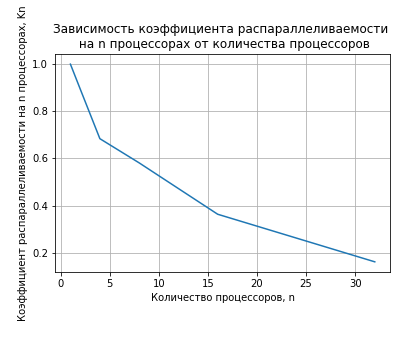
}

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 | Коэффициент распараллеливания |
| 1 | 50.795433 | 0.093000 | 1.0000 |
| 2 | 47.065781 | 0.052000 | 0.8942 |
| 4 | 48.360733 | 0.034000 | 0.6838 |
| 8 | 47.959103 | 0.020000 | 0.5812 |
| 16 | 48.022614 | 0.016000 | 0.3633 |
| 32 | 48.004139 | 0.018000 | 0.1615 |

  
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 после распараллеливания на 4 процессорах время вычисления уменьшается незначительно.

# Вычисление числа Пи

Необходимо вычислить значение числа Пи методом Монте – Карло на 1, 2, 4, 8, 16 и 32 процессорах (всего имеются 4 процессора). Для 100000000 точек. Каждый процессор считает своё количество точек и передаёт получившееся значение в корневой 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 приведен ниже:

#include <mpi.h>

#include <stdio.h>

#include <time.h>

#include <string>

#include <cstdlib>

#include <iostream>

using namespace std;

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

int id, np;

double x, y, eTime, sTime, pTime;

long int lhit, hit = 0, i, lN, N = 100000000;

MPI\_Init(&argc, &argv);

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

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

srand((unsigned)(time(0) + id));

MPI\_Status status;

sTime = MPI\_Wtime();

lN = N / np;

if (id == 0)

{

lhit = 0;

for (i = 1; i <= lN;i++)

{

x = ((double)rand()) / ((double)RAND\_MAX);

y = ((double)rand()) / ((double)RAND\_MAX);

if (((x \* x) + (y \* y)) <= 1) lhit++;

}

hit += lhit;

for (int i = 1; i < np; i++)

{

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

hit += lhit;

}

double est;

est = ((double)hit \* 4) / (double)N;

eTime = MPI\_Wtime();

pTime = eTime - sTime;

cout << "Points: " << N << endl;

cout << "Pi: " << est << endl;

cout << "Duration: " << pTime << endl;

}

else

{

lhit = 0;

for (i = 1; i <= lN;i++)

{

x = ((double)rand()) / ((double)RAND\_MAX);

y = ((double)rand()) / ((double)RAND\_MAX);

if (((x \* x) + (y \* y)) <= 1) lhit++;

}

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

}

MPI\_Finalize();

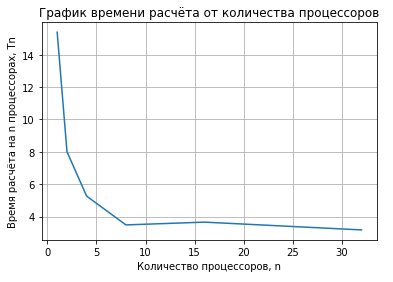
return 0;

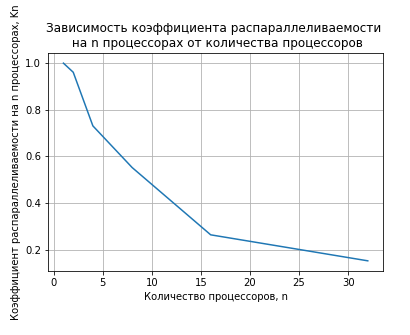
}

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 | Коэффициент распараллеливания |
| 1 | 3.14152 | 15.3879 | 1.0000 |
| 2 | 3.1416 | 8.00693 | 0.9609 |
| 4 | 3.14157 | 5.26451 | 0.7307 |
| 8 | 3.14129 | 3.48539 | 0.5519 |
| 16 | 3.14128 | 3.6567 | 0.2630 |
| 32 | 3.14147 | 3.17593 | 0.1514 |

  
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 процессор будет не только проводить вычисления на своём участке и передавать получившееся значение в корневой процессор с номером “0“, но еще и в каждый момент времени взаимодействовать с “соседними“ процессорами для передачи смежных 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;

clock\_t 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 координате

double T = 10; // Время моделирования

double dt = 0.001; // Шаг по времени

int N = 1 + (l / h); // Общее количество координат

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

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

double T0 = 1; //Температура на левой границе

double T1 = 0.5403; // Температура на правой границе

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

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

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

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

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

}

T\_i[0] = 1;

T\_i\_p[0] = 1;

T\_i[N - 1] = 0.5403;

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

//Определяем интервал,на котором текущему процессору нужно считать

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++) {

// Опрашиваем граничные значения соседей

if (my\_rank > 0) {

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

}

//Считаем на следующем шаге по времени

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

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

}

//Обновляем вместе с граничными

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

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

}

//Отправляем граничные значения соседям

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

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

}

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

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

}

}

//Отправляем всё корневому процессору

if (my\_rank > 0) {

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

}

//Принимаем в корневом процессоре

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]);

}

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

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

printf("%f , ", h \* i);

}

}

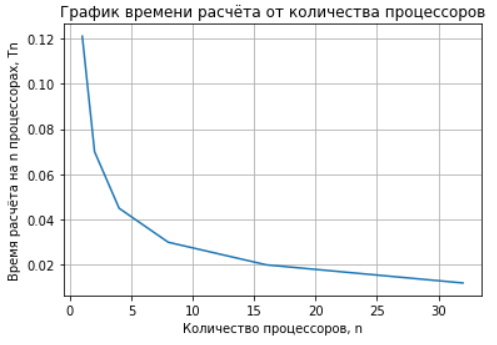
MPI\_Finalize();

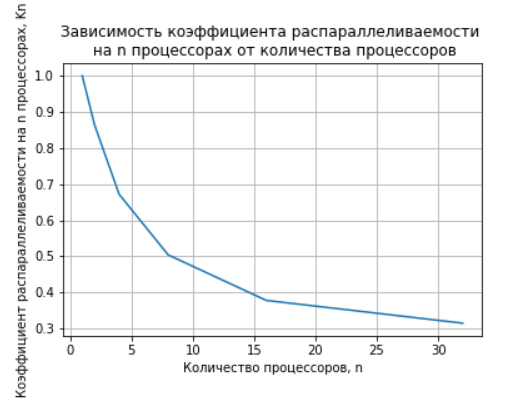
}

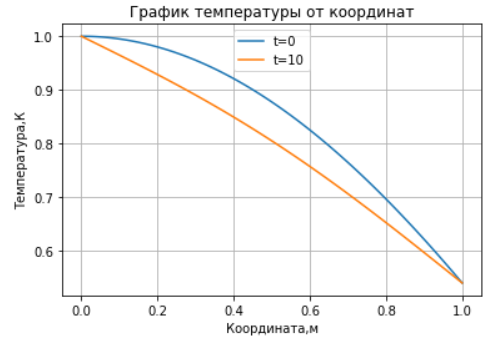
Полученные результаты приведены в таблице 3, а также на рис. 5, рис. 6 и рис. 7.

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

|  |  |  |
| --- | --- | --- |
| Количество процессоров | Время вычисления | Коэффициент распараллеливания |
| 1 | 0.12100 | 1.0000 |
| 2 | 0.07000 | 0.8643 |
| 4 | 0.04500 | 0.6722 |
| 8 | 0.03000 | 0.5042 |
| 16 | 0.02000 | 0.3781 |
| 32 | 0.01200 | 0.3151 |

  
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 вычисления для 4 процессоров правильно отображают закон теплопроводности.

Заключение

Таким образом, был изучен способ параллельного расчёта задачи несколькими процессорами с помощью библиотеки MPI. Было применено взаимодействие между процессорами. В результате выполнения трёх задач были получены результаты, точность которых достаточно велика, при этом получено, что при увеличении количества процессоров (до имеющихся 4) время расчёта уменьшается, но, как видно из зависимостей коэффициента распараллеливания от количества процессоров, время перестаёт уменьшаться значительно, поскольку больше времени начинает уходить на суммирование результатов и инициализацию процессоров.

# Список использованной литературы

# Kendall W. Beginning MPI (An Introduction in C) – MIT press, 2013.

1. Quinn M. Parallel Programming in C with Mpi and Openmp – McGraw-Hill Education, 2008.
2. Snir M., Dongara J., Kowalik J. MPI: The Complete Reference. – The MIT Press, 1998.