###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

**Отчет**

**о выполнении лабораторной работы**

«Параллельная реализация метода Якоби в трехмерной области»

студента 2 курса, 18204 группы

Хорошавина Алексея Константиновича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук,

Доцент.

А.Ю. Власенко

Новосибирск 2019

Оглавление

[Задание 3](#_Toc38972466)

[Результаты измерений 4](#_Toc38972467)

[График зависимости времени выполнения 4](#_Toc38972468)

[График зависимости ускорения 4](#_Toc38972469)

[График зависимости эффективности 5](#_Toc38972470)

[Профилирование 5](#_Toc38972471)

[Заключение 6](#_Toc38972472)

[Приложение 7](#_Toc38972473)

**Задание**

1. Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16. Размеры сетки и порог сходимости подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер. По профилю убедиться, что коммуникации происходят на фоне счета.

# Результаты измерений

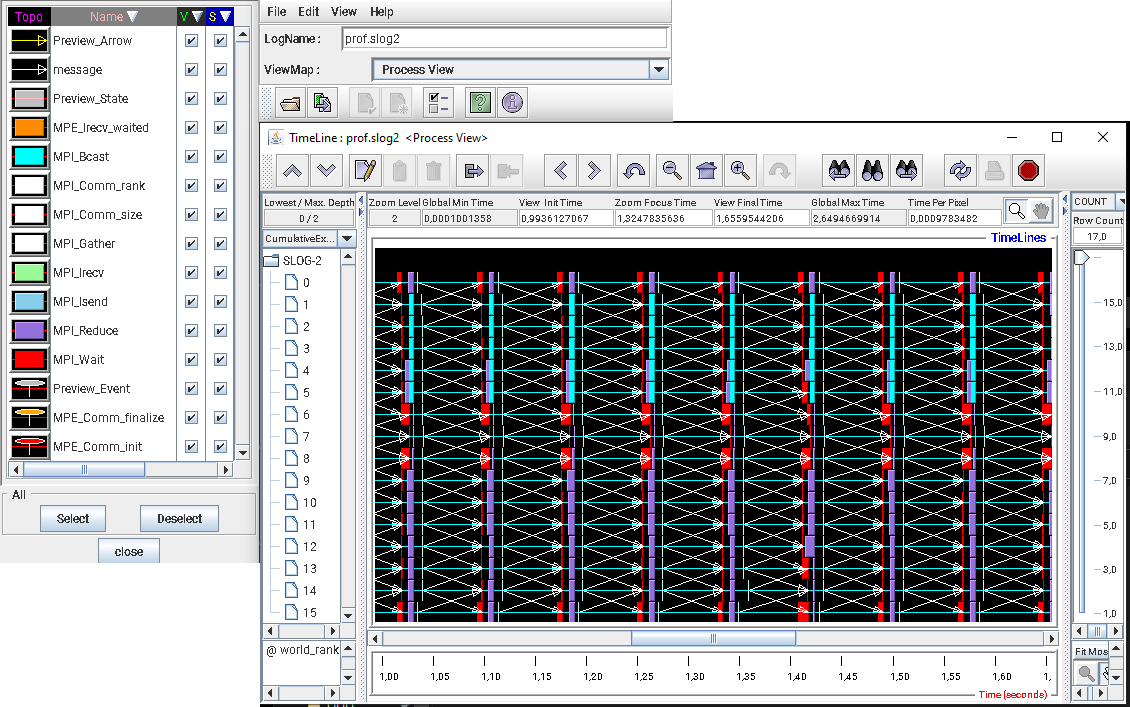
|  |  |  |  |
| --- | --- | --- | --- |
| Ядра | Время(сек.) | Ускорение | Эффективность |
| 1 | 30,4096 | 1 | 1 |
| 2 | 15,8465 | 1,9190 | 0,9595 |
| 4 | 8,1356 | 3,7378 | 0,9344 |
| 8 | 4,6266 | 6,5727 | 0,8215 |
| 16 | 2,6350 | 11,5406 | 0,7212 |

# График зависимости времени выполнения

# График зависимости ускорения

# График зависимости эффективности

# Профилирование



Из профилирования видно, что обмен (белые стрелки) граничными слоями между процессами происходит на фоне вычислений значений функции внутри подобласти.

# Заключение

Был изучен метод распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби. Построены графики зависимости времени, ускорения и эффективности от числа используемых ядер. Выполнено профилирование, которое показало, что обмен данных между процессами производился на фоне вычислений, что позволило сэкономить время на работу всей программы.

# Приложение

Листинг программы

#include **<cstring>**#include **<algorithm>**#include **<iostream>**#include **<math.h>**#include **<mpi.h>  
  
using namespace** std;  
  
**const double** eps = 1e-8;  
**const double** a = 1e5;  
  
**const int** Nx = 256;  
**const int** Ny = 256;  
**const int** Nz = 256;  
  
**const double** min\_x = -1, min\_y = -1, min\_z = -1;  
**const double** max\_x = 1, max\_y = 1, max\_z = 1;  
  
  
**double** phi\_finction(**double** x, **double** y, **double** z) {  
 **return** x\*x+y\*y+z\*z;  
}  
  
**double** ro(**double** x, **double** y, **double** z) {  
 **return** 6-a\*phi\_finction(x,y,z);  
}  
  
**double** update\_layer(**int** base\_z, **int** z, **double**\* values, **double**\* tmp\_values, **double** hx, **double** hy, **double** hz) {  
 **int** abs\_z = base\_z+z;  
  
 **if** (abs\_z==0 || abs\_z==Nz-1) {  
 *//Копируем этот слой в новый массив на старое место, не пересчитывая* memcpy(tmp\_values + z\*Nx\*Ny, values + z\*Nx\*Ny, Nx \* Ny \* **sizeof**(**double**));  
 **return** 0;  
 }  
 *//Иначе пересчитываем каждый элемент слоя с помощью итерационной формулы* **double** max\_delta = 0;  
 **double** cur\_z = min\_z+abs\_z\*hz;  
 **for** (**int** i=0;i<Nx;i++) {  
 **double** cur\_x = min\_x+i\*hx;  
 **for** (**int** j=0;j<Ny;j++) {  
 **double** cur\_y = min\_y+j\*hy;  
 *//Если элемент находится на границе слоя, то не пересчитываем его* **if** (i==0 || i==Nx-1 || j==0 || j==Ny-1) {  
 tmp\_values[z\*Nx\*Ny+i\*Nx+j] = values[z\*Nx\*Ny+i\*Nx+j];  
 **continue**;  
 }  
 **int** index = z\*Nx\*Ny+i\*Nx+j;  
 **double** tmp = (values[z\*Nx\*Ny+(i+1)\*Nx+j]+values[z\*Nx\*Ny+(i-1)\*Nx+j])/(hx\*hx);  
 tmp += (values[z\*Nx\*Ny+i\*Nx+(j+1)]+values[z\*Nx\*Ny+i\*Nx+(j-1)])/(hy\*hy);  
 tmp += (values[(z+1)\*Nx\*Ny+i\*Nx+j]+values[(z-1)\*Nx\*Ny+i\*Nx+j])/(hz\*hz);  
 tmp -= ro(cur\_x, cur\_y, cur\_z);  
 tmp\_values[index] = 1/(2/(hx\*hx)+2/(hy\*hy)+2/(hz\*hz)+a);  
 tmp\_values[index]\*=tmp;  
 max\_delta = max(max\_delta, fabs(tmp\_values[index]-values[index]));  
 }  
 }  
 **return** max\_delta;  
}  
  
**int** main(**int** argc, **char**\*\* argv) {  
 **int** size, rank;  
  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_size(**MPI\_COMM\_WORLD**, &size);  
 MPI\_Comm\_rank(**MPI\_COMM\_WORLD**, &rank);  
  
 **if** (Nz % size) {  
 **if** (rank == 0) {  
 cout << **"Invalid number of processes"** << endl;  
 }  
 MPI\_Finalize();  
 **return** 0;  
 }  
  
 **double** start = MPI\_Wtime();  
  
 **double** hx = (max\_x - min\_x) / (Nx - 1);  
 **double** hy = (max\_y - min\_y) / (Ny - 1);  
 **double** hz = (max\_z - min\_z) / (Nz - 1);  
  
 **int** part\_z = Nz / size;  
  
 **double**\* values = **new double**[(part\_z + 2) \* Nx \* Ny];  
 **double**\* tmp\_values = **new double**[(part\_z + 2) \* Nx \* Ny];  
  
 **int** base\_z = rank \* part\_z-1;  
  
 *//Инициализация слоя* **for** (**int** i = 0; i < part\_z+2; i++) {  
 **int** real\_i = i + base\_z;  
 **double** cur\_z = min\_z + hz\*real\_i;  
 **for** (**int** j = 0; j < Nx; j++) {  
 **double** cur\_x = min\_x + hx\*j;  
 **for** (**int** k = 0; k < Ny; k++) {  
 **double** cur\_y = min\_y + hy\*k;  
 **int** index = i\*Nx\*Ny + j\*Nx + k;  
  
 **if** (real\_i == 0 || real\_i == Nz-1 || j == 0 || j == Nx-1 || k == 0 || k == Ny-1) {  
 values[index] = phi\_finction(cur\_x, cur\_y, cur\_z);  
 } **else** {  
 values[index] = 0;  
 }  
 }  
 }  
 }  
  
  
 **while**(**true**) {  
 **double** max\_delta = 0;  
  
 **double** tmp\_delta = update\_layer(base\_z, 1, values, tmp\_values, hx, hy, hz);  
 max\_delta = max(max\_delta, tmp\_delta);  
  
 tmp\_delta = update\_layer(base\_z, part\_z, values, tmp\_values, hx, hy, hz);  
 max\_delta = max(max\_delta, tmp\_delta);  
  
 MPI\_Request rq[4];  
  
 **if** (rank != 0) {  
 MPI\_Isend(tmp\_values+Nx\*Ny, Nx\*Ny, **MPI\_DOUBLE**, rank-1, 123, **MPI\_COMM\_WORLD**, &rq[0]);  
 MPI\_Irecv(tmp\_values, Nx\*Ny, **MPI\_DOUBLE**, rank-1, 123, **MPI\_COMM\_WORLD**, &rq[2]);  
 }  
  
 **if** (rank != size-1) {  
 MPI\_Isend(tmp\_values+part\_z\*Nx\*Ny, Nx\*Ny, **MPI\_DOUBLE**, rank+1, 123, **MPI\_COMM\_WORLD**, &rq[1]);  
 MPI\_Irecv(tmp\_values+(part\_z+1)\*Nx\*Ny, Nx\*Ny, **MPI\_DOUBLE**, rank+1, 123, **MPI\_COMM\_WORLD**, &rq[3]);  
 }  
  
 **for** (**int** i = 2; i < part\_z; i++) {  
 **double** tmp\_delta = update\_layer(base\_z, i, values, tmp\_values, hx, hy, hz);  
 max\_delta = max(max\_delta, tmp\_delta);  
 }  
  
 **if** (rank != 0) {  
 MPI\_Wait(&rq[0], **MPI\_STATUS\_IGNORE**);  
 MPI\_Wait(&rq[2], **MPI\_STATUS\_IGNORE**);  
 }  
  
 **if** (rank != size-1) {  
 MPI\_Wait(&rq[1], **MPI\_STATUS\_IGNORE**);  
 MPI\_Wait(&rq[3], **MPI\_STATUS\_IGNORE**);  
 }  
  
 memcpy(values, tmp\_values, (part\_z+2)\*Nx\*Ny\***sizeof**(**double**));  
 **double** max\_delta\_shared;  
  
 MPI\_Reduce(&max\_delta, &max\_delta\_shared, 1, **MPI\_DOUBLE**, **MPI\_MAX**, 0, **MPI\_COMM\_WORLD**);  
  
 MPI\_Bcast(&max\_delta\_shared, 1, **MPI\_DOUBLE**, 0, **MPI\_COMM\_WORLD**);  
  
 **if** (max\_delta\_shared < eps) {  
 **break**;  
 }  
 }  
  
 **double**\* results;  
 **if** (rank == 0) {  
 results = **new double**[Nx\*Ny\*Nz];  
 }  
  
 MPI\_Gather(values+Nx\*Ny, part\_z\*Nx\*Ny, **MPI\_DOUBLE**, results, part\_z\*Nx\*Ny, **MPI\_DOUBLE**, 0, **MPI\_COMM\_WORLD**);  
  
 **if** (rank == 0) {  
 **double** max\_delta = 0;  
 **for** (**int** layer = 0;layer < Nz;layer++){  
 **double** z = min\_z + layer\*hz;  
 **for** (**int** j = 0;j < Nx;j++) {  
 **double** x= min\_x + j\*hx;  
 **for** (**int** k = 0; k < Ny; k++) {  
 **double** y = min\_y + k\*hy;  
 **double** tmp = results[layer\*Nx\*Ny + j\*Nx + k];  
 **double** val = phi\_finction(x, y, z);  
 max\_delta = max(max\_delta, fabs(tmp-val));  
 }  
 }  
 }  
 cout << **"Delta: "** << max\_delta << endl;  
 **if**(max\_delta < 100\*eps){  
 cout << **"Good"** << endl;  
 }  
 **else**{  
 cout << **"Bad"** << endl;  
 }  
 **delete**[] results;  
 }  
  
 **double** end = MPI\_Wtime();  
  
 **if** (rank == 0) {  
 printf(**"Time: %lf\n"**, end - start);  
 }  
  
 **delete**[](values);  
 **delete**[](tmp\_values);  
  
 MPI\_Finalize();  
 **return** 0;  
}