###### 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 решения системы линейных алгебраических уравнений с помощью OpenMP»

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](#_Toc35382711)

[График зависимости времени выполнения 4](#_Toc35382712)

[График зависимости ускорения 4](#_Toc35382713)

[График зависимости эффективности 5](#_Toc35382714)

[Заключение 6](#_Toc35382715)

[Приложение 7](#_Toc35382716)

**Задание**

1. Последовательную программу из лабораторной работы 1, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. Реализовать два варианта программы:

* Вариант 1: для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for,
* Вариант 2: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: от 1 до числа доступных в узле. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
2. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.
3. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

# **График зависимости времени выполнения**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 ядро | 2 ядра | 3 ядра | 4 ядра | 5 ядер | 6 ядер | 7 ядер | 8 ядер | 9 ядер | 10 ядер | 11 ядер | 12 ядер |
| Вариант1 | 51,86 | 34,58 | 24,19 | 17,06 | 14,11 | 11,93 | 10,37 | 8,95 | 8,18 | 7,34 | 6,68 | 6,12 |
| Вариант2 | 51,86 | 28,47 | 19,04 | 13,52 | 11,32 | 9,5 | 8,18 | 7,14 | 6,49 | 5,8 | 5,4 | 4,96 |

**График зависимости ускорения**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 ядро | 2 ядра | 3 ядра | 4 ядра | 5 ядер | 6 ядер | 7 ядер | 8 ядер | 9 ядер | 10 ядер | 11 ядер | 12 ядер |
| Вариант1 | 51,86 | 1,49 | 2,14 | 3,03 | 3,67 | 4,34 | 5,00 | 5,79 | 6,33 | 7,06 | 7,76 | 8,47 |
| Вариант2 | 51,86 | 1,82 | 2,72 | 3,83 | 4,58 | 5,45 | 6,33 | 7,26 | 7,99 | 8,94 | 9,6 | 10,45 |

**График зависимости эффективности**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 ядро | 2 ядра | 3 ядра | 4 ядра | 5 ядер | 6 ядер | 7 ядер | 8 ядер | 9 ядер | 10 ядер | 11 ядер | 12 ядер |
| Вариант1 | 1 | 0,74 | 0,71 | 0,75 | 0,73 | 0,72 | 0,71 | 0,72 | 0,7 | 0,7 | 0,71 | 0,7 |
| Вариант2 | 1 | 0,91 | 0,9 | 0,95 | 0,91 | 0,9 | 0,9 | 0,91 | 0,88 | 0,89 | 0,87 | 0,87 |

**Заключение**

Был реализован алгоритм решения СЛАУ итерационным методом. Были реализованы два варианта распараллелить программу с помощью OpenMP, построены графики зависимости времени работы, ускорения и эффективности. По результатам на графиках видно, что второй вариант с созданием одной параллельной секции оказался эффективней, чем первый вариант с созданием параллельных секций для каждого цикла.

Было проведено исследование параметров для #pragma omp schedule():

Static – самый эффективный, поскольку в данном алгоритме итерации в циклах выполняются примерно за одинаковое время, и потоки выполняют примерно одинаковое количество итераций.

Guided – по эффективности такой же, как и static, так как равномерно распределяет данные между потоками, в зависимости от числа оставшихся итераций.

Dynamic – работает дольше чем два предыдущих, так как зависит от размера самого пакета итераций для каждого потока.

Runtime – самый медленный, так как распределяет итерации между потоками во время исполнения, на что тратится дополнительное время.

**Приложение**

Листинг программы:

(Последовательная программа)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <time.h>  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R)  
{  
 *for*(*int* i = 0; i < N; i++){  
 R[i] = 0;  
 *for*(*int* j = 0; j < N; j++)  
 {  
 R[i] += M[i][j]\*V[j];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps)  
{  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\*N);  
  
 *double* crit\_module;  
 *double* chisl\_Tau;  
 *double* del\_Tau;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 Matrix\_by\_vector(N, A, Xn, R);  
 *for*(*int* i = 0; i < N; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R);  
  
 chisl\_Tau = 0.0;  
 del\_Tau = 0.0;  
  
 *for*(*int* i = 0; i < N; i++)  
 {  
 chisl\_Tau += R[i]\*Y[i];  
 del\_Tau += R[i]\*R[i];  
 }  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
  
 printf("%lf\n", chisl\_Tau);  
  
 *for*(*int* i = 0; i < N; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
  
 Matrix\_by\_vector(N, A, X, R);  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
 *for*(*int* i = 0; i < N; i++){  
 crit\_1 += pow(R[i] - b[i], 2);  
 crit\_2 += pow(b[i], 2);  
 }  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = X[i];  
 }  
 }  
 *while* (crit\_module >= eps);  
  
 free(R);  
 free(Y);  
 free(Xn);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 9000;  
  
 *struct* timespec start, end;  
  
 *double* \*\*A;  
 A = (*double*\*\*)malloc(N \**sizeof*(*double*\*));  
 *for*(*int* i = 0; i < N; ++i){  
 A[i] = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 }  
  
 *for*(*int* i = 0; i < N; i++){  
 *for*(*int* j = 0; j < N; j++){  
 *if*(i == j){  
 A[i][j] = 2.0;  
 } *else*{  
 A[i][j] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *for*(*int* i = 0; i < N; i++){  
 u[i] = sin((2\*M\_1\_PI\*i));  
 }  
  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 Matrix\_by\_vector(N, A, u, b);  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*) \* N);  
 *double* epsilon = pow(10, -5);  
  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);  
 Minimal\_Nevazki(N, A, b, X, epsilon);  
 clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);  
  
  
 printf("Time: %lf sec \n", end.tv\_sec - start.tv\_sec + 0.000000001\*(end.tv\_nsec - start.tv\_nsec));  
  
 */\*for(int i = 0; i < N; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/  
  
 for* (*int* i = 0; i < N; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
  
 *return* 0;  
}

(OpenMP 1 Вариант: для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <time.h>  
#include <omp.h>  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R)  
{  
 *int* j;  
#pragma omp parallel *for private*(j)  
 *for*(*int* i = 0; i < N; i++){  
 R[i] = 0;  
 *for*(j = 0; j < N; j++)  
 {  
 R[i] += M[i][j]\*V[j];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps)  
{  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\* N);  
  
 *double* crit\_module;  
 *double* chisl\_Tau;  
 *double* del\_Tau;  
  
 *for*(*int* i = 0; i < N; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 Matrix\_by\_vector(N, A, Xn, R);  
  
#pragma omp parallel *for  
 for*(*int* i = 0; i < N; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R);  
  
 chisl\_Tau = 0.0;  
 del\_Tau = 0.0;  
  
#pragma omp parallel *for* reduction(+:chisl\_Tau, del\_Tau)  
 *for*(*int* i = 0; i < N; i++)  
 {  
 chisl\_Tau += R[i]\*Y[i];  
 del\_Tau += R[i]\*R[i];  
 }  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
  
#pragma omp parallel *for  
 for*(*int* i = 0; i < N; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
 Matrix\_by\_vector(N, A, X, R);  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
#pragma omp parallel *for* reduction(+:crit\_1, crit\_2)  
 *for*(*int* i = 0; i < N; i++){  
 crit\_1 += pow(R[i] - b[i], 2);  
 crit\_2 += pow(b[i], 2);  
 }  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
#pragma omp parallel *for  
 for*(*int* i = 0; i < N; i++){  
 Xn[i] = X[i];  
 }  
 }  
 *while* (crit\_module >= eps);  
  
 free(R);  
 free(Y);  
 free(Xn);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 9000;  
 *double* start, end;  
  
 *double* \*\*A;  
 A = (*double*\*\*)malloc(N \**sizeof*(*double*\*));  
 *for*(*int* i = 0; i < N; ++i){  
 A[i] = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 }  
  
 *for*(*int* i = 0; i < N; i++){  
 *for*(*int* j = 0; j < N; j++){  
 *if*(i == j){  
 A[i][j] = 2.0;  
 } *else*{  
 A[i][j] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *for*(*int* i = 0; i < N; i++){  
 u[i] = sin(2\*M\_1\_PI\*i);  
 }  
  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 Matrix\_by\_vector(N, A, u, b);  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*) \* N);  
 *double* epsilon = pow(10, -5);  
  
 start = omp\_get\_wtime();  
 Minimal\_Nevazki(N, A, b, X, epsilon);  
 end = omp\_get\_wtime();  
  
 printf("Time: %lf sec. \n", end - start);  
  
 */\*for(int i = 0; i < N; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/  
  
 for* (*int* i = 0; i < N; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
  
 *return* 0;  
}

(OpenMP 2 Вариант: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм)

#include <cmath>  
#include <cstdio>  
#include <cstdlib>  
#include <omp.h>  
  
*void* Matrix\_by\_vector(*int* N, *double* \*\*M, *const double* \*V, *double* \*R)  
{  
*int* j;  
#pragma omp *for private*(j)  
 *for*(*int* i = 0; i < N; i++){  
 R[i] = 0;  
 *for*(j = 0; j < N; j++)  
 {  
 R[i] += M[i][j]\*V[j];  
 }  
 }  
}  
  
*void* Minimal\_Nevazki(*int* N, *double* \*\*A, *const double* \*b, *double* \*X, *double* eps)  
{  
 *double* \*R = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Y = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *double* \*Xn = (*double*\*)malloc(*sizeof*(*double*)\* N);  
  
 *double* crit\_module;#pragma omp parallel  
 {  
#pragma omp *for  
 for*(*int* i = 0; i < N; i++){  
 Xn[i] = 0;  
 }  
  
 *do*{  
 Matrix\_by\_vector(N, A, Xn, R);  
  
#pragma omp *for  
 for*(*int* i = 0; i < N; i++){  
 Y[i] = R[i] - b[i];  
 }  
  
 Matrix\_by\_vector(N, A, Y, R);  
  
 *double* chisl\_Tau = 0.0;  
 *double* del\_Tau = 0.0;  
  
#pragma omp shared(chisl\_Tau, del\_Tau) *for* reduction(+:chisl\_Tau, del\_Tau)  
 *for*(*int* i = 0; i < N; i++)  
 {  
 chisl\_Tau += R[i]\*Y[i];  
 del\_Tau += R[i]\*R[i];  
 }  
  
 chisl\_Tau = chisl\_Tau/del\_Tau;  
  
#pragma omp *for  
 for*(*int* i = 0; i < N; i++){  
 X[i] = Xn[i] - chisl\_Tau\*Y[i];  
 }  
  
 Matrix\_by\_vector(N, A, X, R);  
  
 *double* crit\_1 = 0.0;  
 *double* crit\_2 = 0.0;  
  
#pragma omp shared(crit\_1, crit\_2) *for* reduction(+:crit\_1, crit\_2)  
 *for*(*int* i = 0; i < N; i++){  
 crit\_1 += pow(R[i] - b[i], 2);  
 crit\_2 += pow(b[i], 2);  
 }  
  
 crit\_1 = sqrt(crit\_1);  
 crit\_2 = sqrt(crit\_2);  
 crit\_module = crit\_1/crit\_2;  
  
#pragma omp *for  
 for*(*int* i = 0; i < N; i++){  
 Xn[i] = X[i];  
 }  
 }  
 *while* (crit\_module >= eps);  
 }  
 free(R);  
 free(Y);  
 free(Xn);  
}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 *int* N = 9000;  
  
 *double* start, end;  
  
 *double* \*\*A;  
 A = (*double*\*\*)malloc(N \**sizeof*(*double*\*));  
 *for*(*int* i = 0; i < N; ++i){  
 A[i] = (*double*\*)malloc(*sizeof*(*double*) \* N);  
 }  
  
 *for*(*int* i = 0; i < N; i++){  
 *for*(*int* j = 0; j < N; j++){  
 *if*(i == j){  
 A[i][j] = 2.0;  
 } *else*{  
 A[i][j] = 1.0;  
 }  
 }  
 }  
  
 *double* \*u = (*double*\*)malloc(*sizeof*(*double*)\*N);  
 *for*(*int* i = 0; i < N; i++){  
 u[i] = sin(2\*M\_1\_PI\*i);  
 }  
  
 *double* \*b = (*double*\*)malloc(*sizeof*(*double*) \* N);  
  
#pragma omp parallel  
{  
 Matrix\_by\_vector(N, A, u, b);  
}  
  
 *double* \*X = (*double*\*) malloc(*sizeof*(*double*) \* N);  
 *double* epsilon = pow(10, -5);  
  
 start = omp\_get\_wtime();  
 Minimal\_Nevazki(N, A, b, X, epsilon);  
 end = omp\_get\_wtime();  
  
 printf("Time: %lf sec. \n", end - start);  
  
 */\*for(int i = 0; i < N; i++){  
 printf("X[%d] = %lf u[%d] = %lf\n",i, X[i], i, u[i]);  
 }\*/  
  
 for* (*int* i = 0; i < N; ++i)  
 free(A[i]);  
 free(A);  
  
 free(u);  
 free(b);  
 free(X);  
  
 *return* 0;  
}