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 ПРАКТИЧЕСКОЙ РАБОТЫ №3

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»»

студентки 2 курса, группы 21204

Егоренко Ксении Владиславовны

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

(к.т.н., доцент)

А. Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

ЦЕЛЬ

Ознакомиться со стандартом OpenMP, сутью которого является создание нескольких потоков с целью параллельного исполнения вычислительных частей программы.

ЗАДАНИЕ

1. Последовательную программу из лабораторной работы 1 (см. Приложение 1), реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP (см. Приложение 2). Создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.
2. 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.
3. . Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.
4. На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.

ХОД РАБОТЫ

1. Была написана программа на языке C++, реализующая итерационный алгоритм решения СЛАУ вида Ax = b (см. Приложение 1);
2. С помощью OpenMP была написана параллельная программа реализующая итерационный алгоритм решения СЛАУ вида Ax = b (см. Приложение 2);
3. Обе программы были проверены на корректность и запущены на кластере НГУ, параллельный вариант программы был на различном числе потоков (1, 2, 4, 8, 12, 16)
4. На основании измерений были построены графики времени, ускорения и эффективности для параллельного варианта программы (см. Приложение 3).
5. Был построен график измерений времени с различными параметрами #pragma omp schedule(...) (см. Приложение 4).
6. Был составлен отчет по результатам практической работы.

ЗАКЛЮЧЕНИЕ

1. Из плюсов использования стандарта OpenMP для распараллеливания программ можно выделить простоту использования данного стандарта (например, в сравнении с MPI): для того, чтобы распараллелить программу нам достаточно всего лишь добавить нужные прагмы. Из минусов данного стандарта: наша программа может быть распараллелена только на одном вычислительном узле, что является довольно строгим ограничением.
2. По результатам измерения времени выполнения, ускорения и эффективности можно сделать вывод, что все эти показатели остаются оптимальными до тех пор, пока количество потоков не превысит максимальное число ядер на одном вычислительном узле кластера. Это можно объяснить тем, что на кластере несколько потоков не могут одновременно работать на одном ядре, таким образом 12 потоков будут выполнять работу сразу, оставшиеся встанут в очередь и будут “ждать” пока не освободится одно из ядер.
3. Был определен наилучший параметр для schedule: в моем случае это schedule(dynamic, 50) (время выполнения с использованием этих параметров наименьшее см. Приложение 4).

ПРИЛОЖЕНИЕ 1. Листинг последовательной программы

```
#include <iostream>
#include <cmath>
#include <time.h>
#include <omp.h>

using namespace std;

const double eps = 0.000000001;
const double taul = 0.00001;
const int N = 7000;

void firstInit(double* A, double* b, double* x) {
   for (int i = 0; i < N; ++i) {
         for (int j = 0; j < N; ++j) {
               A[i * N + j] = i == j ? 2 : 1; // утяжеляем
   диагональ двойками, остальные значения в матрице 1
         }
         b[i] = N + 1;
         x[i] = 0; // при первой инициализации заподняем вектор x
   нулями
   }
}

void countVecMatrMult(double* matr, double* vec, double* result)
   {
   for (int i = 0; i < N; ++i) {
         result[i] = 0;
         for (int j = 0; j < N; ++j) {
               result[i] += matr[i * N + j] * vec[j];
         }
   }
}

void countSubOfVectors(double* result, double* left, double*
   right) {
   for (int i = 0; i < N; ++i) {
         result[i] = left[i] - right[i];
   }
}

void countScalarMatrMult(double* result, double* vec) {
   for (int i = 0; i < N; ++i) {
         result[i] = vec[i] * taul;
   }
}

double countAbs(double* vec) {
   double abs = 0;
   for (int i = 0; i < N; ++i) {
         abs += pow(vec[i], 2);
   }
   return abs;
}
```

```
int main(int argc, char** argv[]) {
   double start, end;
   start = omp_get_wtime();

   double* A = new double[N * N];
   double* b = new double[N];
   double* x = new double[N];

   // создадим вектора для хранения промежуточных вычислений
   double* Ax = new double[N];
   double* subAx_b = new double[N];
   double* multTaulAx_b = new double[N];

   // инициализируем переменные для проверки конца итераций
   double absAx_b = 0, abs_b = 0;

   firstInit(A, b, x);

   countVecMatrMult(A, x, Ax);
   countSubOfVectors(subAx_b, Ax, b);
   absAx_b = countAbs(subAx_b);
   abs_b = countAbs(b);

   long double sqrEps = pow(eps, 2);
   double g_x = absAx_b / abs_b;
   int itearationsNum = 0;

   while (g_x >= sqrEps && itearationsNum < 10000) {
         countScalarMatrMult(multTaulAx_b, subAx_b);
         countSubOfVectors(x, x, multTaulAx_b);
         countVecMatrMult(A, x, Ax);
         countSubOfVectors(subAx_b, Ax, b);

         absAx_b = countAbs(subAx_b);
         g_x = absAx_b / abs_b;
         itearationsNum++;
   }
   end = omp_get_wtime();

   for (int i = 0; i < N; ++i) {
         cout << x[i] << " ";
   }

   printf("Time spend: %lf", end - start);


   delete[] A;
   delete[] x;
   delete[] Ax;
   delete[] b;
   delete[] subAx_b;
   delete[] multTaulAx_b;

   return 0;
}
```

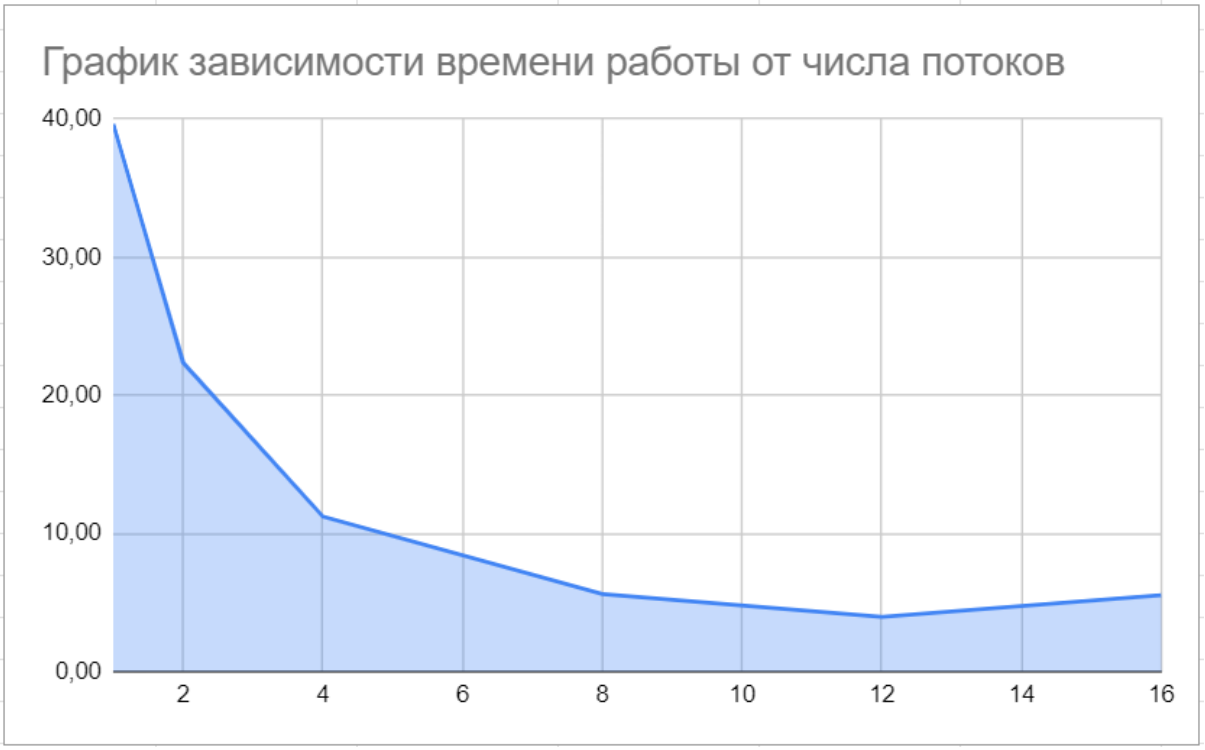
ПРИЛОЖЕНИЕ 2. Листинг параллельной программы

```
#include <iostream>
#include <cmath>
#include <time.h>
#include <omp.h>

using namespace std;

const double eps = 0.00001;
const double taul = 0.00001;
const int N = 7000;

void firstInit(double* A, double* b, double* x) {
   for (int i = 0; i < N; ++i) {
         for (int j = 0; j < N; ++j) {
               A[i * N + j] = i == j ? 2 : 1;
         }
         b[i] = N + 1;
         x[i] = 0;
   }
}

void countVecMatrMult(double* matr, double* vec, double* result)
   {
#pragma omp for schedule(auto)
   for (int i = 0; i < N; ++i) {
         result[i] = 0;
         for (int j = 0; j < N; ++j) {
               result[i] += matr[i * N + j] * vec[j];
         }
   }
}

void countSubOfVectors(double* result, double* left, double*
   right) {
#pragma omp for schedule(auto)
   for (int i = 0; i < N; ++i) {
         result[i] = left[i] - right[i];
   }
}

void countScalarMatrMult(double* result, double* vec) {
#pragma omp for schedule(auto)
   for (int i = 0; i < N; ++i) {
         result[i] = vec[i] * taul;
   }
}

int main(int argc, char** argv) {
   double* A = new double[N * N];
```

```
   double* b = new double[N];
   double* x = new double[N];
   double* Ax = new double[N];
   double* subAx_b = new double[N];
   double* multTaulAx_b = new double[N];

   double absAx_b = 0;
   double abs_b = 0;
   double startTime = 0;
   double endTime = 0;
   long double sqrEps = pow(eps, 2);
   int itearationsNum = 0;
   double g_x = 0.0;


   firstInit(A, b, x);
   startTime = omp_get_wtime();
   omp_set_num_threads(1);
#pragma omp parallel
   {
         countVecMatrMult(A, x, Ax);
         countSubOfVectors(subAx_b, Ax, b);
#pragma omp for schedule(auto) reduction(+ : abs_b)
         for (int i = 0; i < N; i++) {
               abs_b += pow(b[i], 2);
         }
#pragma omp for schedule(auto) reduction(+ : absAx_b)
         for (int i = 0; i < N; i++) {
               absAx_b += pow(subAx_b[i], 2);
         }

#pragma omp single
         {
               g_x = absAx_b / abs_b;
         }

         while (g_x >= sqrEps && itearationsNum < 10000) {
               countScalarMatrMult(multTaulAx_b, subAx_b);
               countSubOfVectors(x, x, multTaulAx_b);
               countVecMatrMult(A, x, Ax);
               countSubOfVectors(subAx_b, Ax, b);

#pragma omp single
               {
                     absAx_b = 0;
               }
#pragma omp for schedule(auto) reduction(+:absAx_b)
               for (int i = 0; i < N; i++) {
                     absAx_b += pow(subAx_b[i], 2);
               }
#pragma omp single
               {
                     g_x = absAx_b / abs_b;
                     itearationsNum++;
               }
         }
```

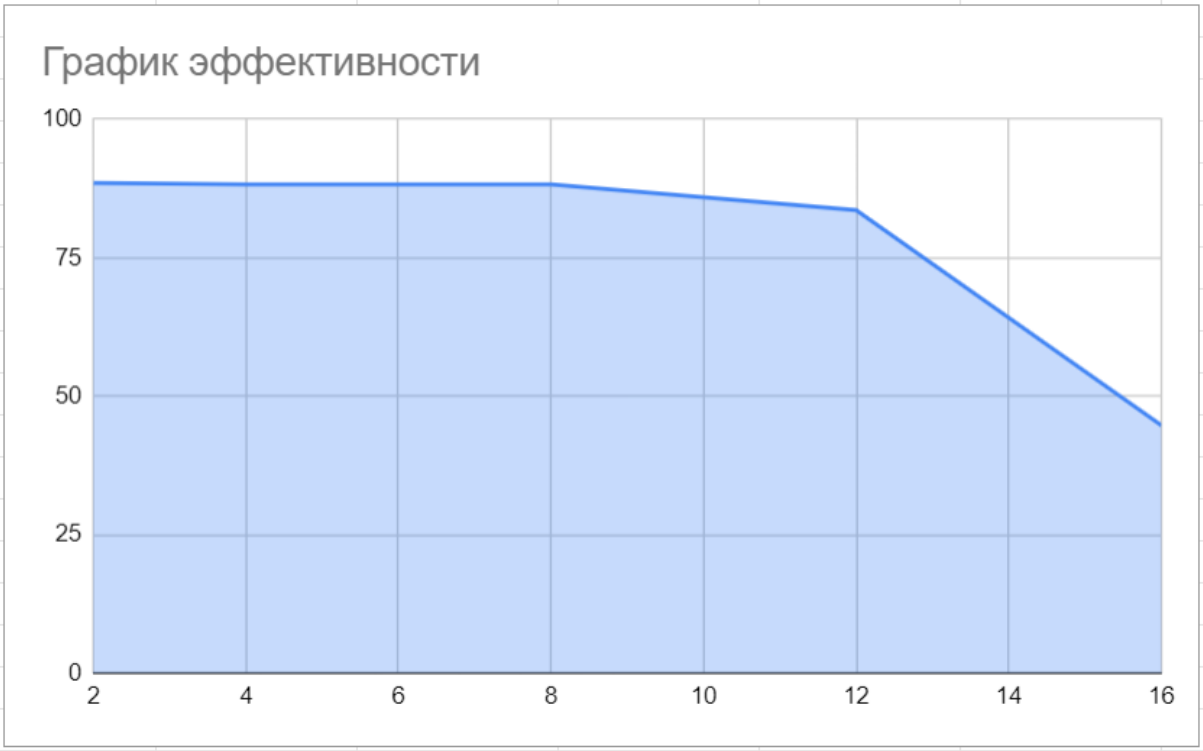
```
    }
    endTime = omp_get_wtime();
    std::cout << "Time spent: " << endTime - startTime <<
    std::endl;

    delete[] A;
    delete[] x;
    delete[] Ax;
    delete[] b;
    delete[] subAx_b;
    delete[] multTaulAx_b;

    return 0;
}
```

ПРИЛОЖЕНИЕ 3. Графики времени, ускорения и эффективности для параллельной программы



1	2	4	8	12	16
39,64	22,37	11,22	5,61	3,95	5,53

2	4	8	12	16
1,77	3,53	7,06	10,03	7,16



2	4	8	12	16
88,5	88,25	88,25	83,58	44,75

ПРИЛОЖЕНИЕ 4. Графики параметров для schedule

Размерность задачи: N = 7000

Кол-во потоков: 12

размер чанка	auto	static	dynamic	guided
1	5,60	5,82	23,42	4,71
50	5,60	5,50	4,01	4,63
100	5,60	5,60	4,04	4,63
150	5,60	5,49	4,11	4,62
200	5,60	5,70	4,11	4,60
250	5,60	5,83	4,9	4,83