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

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

студента 2 курса, группы 21206

Балашова Вячеслава Вадимовича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023

Содержание

Цель

1. Научиться создавать параллельные программы на основе стандарта для распараллеливания программ на языке Си Open Multi-Processing (OpenMP).

Задание

1. Последовательную программу из предыдущей практической работы, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью OpenMP. ОБЯЗАТЕЛЬНОЕ УСЛОВИЕ: создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм.

2. Замерить время работы программы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках. 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.

Описание работы

Ход выполнения работы:

1. Был использован метод простой итерации:

a. Решение на каждом шаге задается формулой:

$$x_{n+1} = x_n - \tau(Ax_n - b)$$

Где τ – константа, параметр метода.

b. Критерий завершения счета:

$$\frac{\| Ax_n - b \|_{\mathbb{R}}}{\| b \|_{\mathbb{R}}} < \varepsilon$$

c. $\| u \|_{\mathbb{R}} = \sqrt{\sum_{i=0}^{N-1} u_i^2}$ – норма вектора.

d. $\varepsilon = 1 * 10^{-5}$

2. Алгоритм применялся на специально подготовленной матрице, которая заполняется следующим образом:

a. Выбирается специальное число – seed – «семя» генератора, которое гарантирует одинаковые псевдослучайные значения в заполнении матрицы для разных запусков программ.

b. Матрица имеет размер 3000 x 3000.

c. Для каждой ячейки матрицы с помощью генератора выбирается число из промежутка [-200; 200].

d. Если ячейка матрицы принадлежит главной диагонали, то к значению ячейки прибавляется число N*1.1, где N – размерность матрицы.

3. Также специально подготавливались вектор «b», ячейки которого заполнялись случайными целыми числами в промежутке [0; 99], и вектор начального приближения «x», ячейки которого заполнялись случайными целыми числами в промежутке [0; 9].

4. Была написана программа на языке программирования C, реализующая данный итерационный алгоритм с разбиением вычислений на несколько потоков, работающих параллельно с помощью директив стандарта OpenMP (см. Приложение 1.). Распараллелен алгоритм нахождения решения СЛАУ и функции, к которые вызываются внутри алгоритма. Была проверена корректность работы распараллеленного алгоритма. За счет распараллеливания вычислений были ускорены вычисления:

a. Скалярного квадрата вектора.

b. Умножения матрицы на вектор.

c. Умножение вектора на скаляр.

d. Разность векторов.

5. В программе был произведен замер времени работы алгоритма с помощью функции OpenMP omp_get_wtime() на разном количестве программных потоков: 1, 2, 4, 8, 16, 24 (сам алгоритм вынесен в отдельную функцию, замер времени работы которой и замеряется). Каждая программа запускалась по 5 раз, из этих 5-ти запускова бралось минимальное время работы.

6. Были построены графики зависимости времени работы программы, эффективности (Ep = Sp / p * 100%, где Sp – ускорение, p – количество потоков) и ускорения (Sp = T1 / Tp, где T1 – время работы последовательной программы, Tp – время работы программы на p потоках) от количества параллельных потоков.

График 1. Зависимость времени работы программы от количества потоков

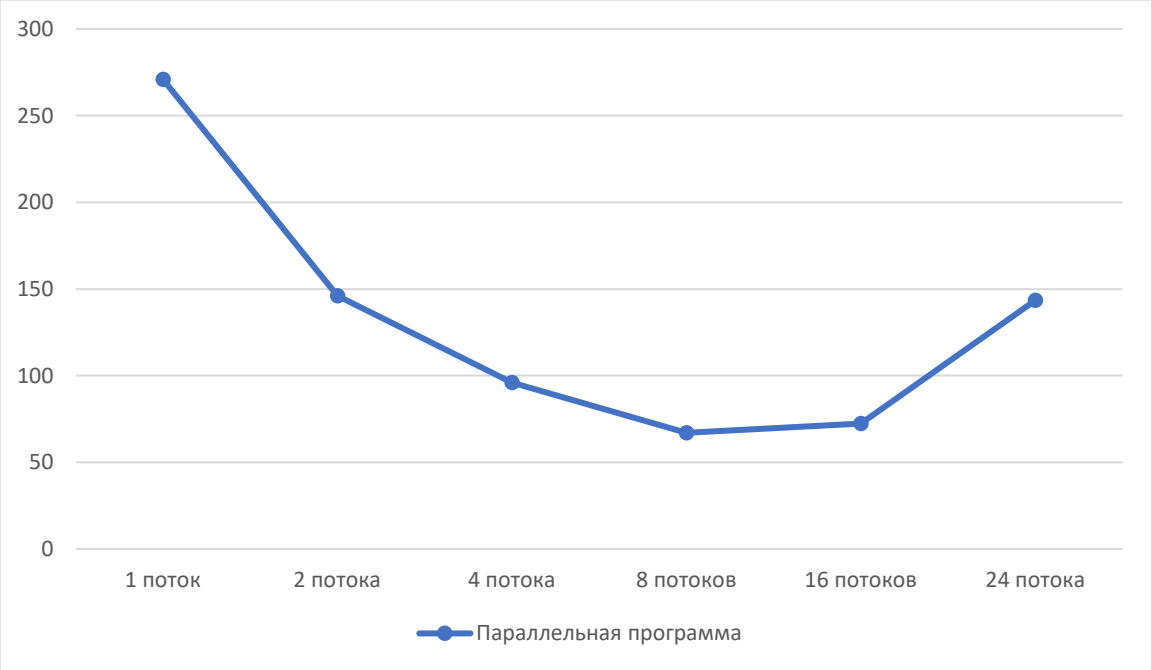

Таблица 1. Соответствие времени работы программы и количества параллельных потоков.

Время, сек.	Количество потоков, шт.
270,986699	1 поток
146,040529	2 потока
96,06813	4 потока
67,04605	8 потоков
72,34625	16 потоков
143,5094	24 потока

График 2. Зависимость ускорения программы от количества параллельных процессов.

Таблица 2. Соответствие ускорения программы и количества параллельных потоков.

Ускорение, коэфф.	Количество потоков, шт.
1	1 поток
1,855558	2 потока
2,820776	4 потока
4,0418	8 потоков
3,745691	16 потоков
1,888286	24 потока

График 3. Зависимость эффективности программы от количества параллельных процессов.

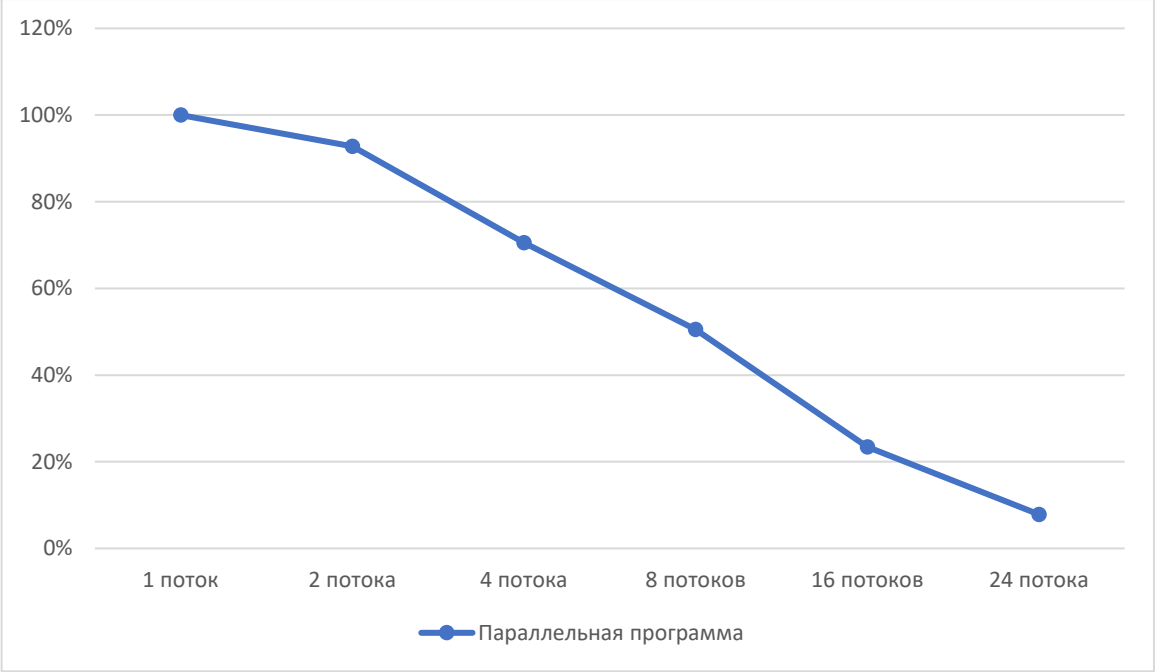

Таблица 3. Соответствие эффективности программы и количества параллельных потоков.

Эффективность, %	Количество потоков, шт.
100%	1 поток
93%	2 потока
71%	4 потока
51%	8 потоков
23%	16 потоков
8%	24 потока

7. Было проведено исследование влияния типа планирования для циклов for (clause schedule) для трех типов планирования и разных параметров размера секции. Для исследования были выбраны те же самые алгоритмы заполнения векторов и матрицы, но размером 700*700. Из пяти замеров брался минимальный по времени результат:

Размер секции	Static, сек	Dynamic, сек	Guided, сек
1	22,756	137,975	6,337
10	18,201	133,313	6,281
100	14,820	112,851	6,209
1000	12,136	37,848	5,937
10000	5,036	5,058	4,843

50000	6,626	6,674	6,678
61250	4,572	4,643	4,667
100000	6,642	6,716	6,708

Самой эффективной комбинацией оказалась пара «static, N / size», где N – длина цикла, size – количество потоков.

Заключение

В ходе выполнения практической работы был получен опыт работы со стандартом OpenMP. Были использованы основные директивы данного стандарта, такие как:

1. #pragma omp parallel – создание потоков для параллельного исполнения куска кода.
2. #pragma omp single – выполнение следующего участка кода одним любым потоком.
3. #pragma omp barrier – ожидание всех потоков.
4. #pragma omp atomic – «атомарность операции» - гарантия не перемешивания инструкций.
5. #pragma omp for – распределение итераций цикла по потокам.

Были использованы и изучены основные пункты (ориг. clauses) OpenMP директив.

Было выяснено влияние на цикл for пункта schedule(SCHEDULE_MODE), где вместо SCHEDULE_MODE может стоять выражение типа "static/dynamic/guided, %number”.

Также были выявлены особенности OpenMP, такие как работа с общей памятью, возможная гонка за данными, возможность распараллеливать отдельные участки кода с определенным числом потоков, а не всю программу сразу, отсюда следует невозможность использования данного стандарта для нескольких узлов (в случае кластера), но за счет отсутствия копирования данных можно увеличить производительность на одном узле, потратив время на инициализация многопоточных сегментов.

Приложение 1. Исходный код программы поиска решения СЛАУ.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <omp.h>

#define SCHEDULE_MODE guided, 100000

void makeMatrixRandomSymmetrical(double * matrix, const int size)
{
   for (int i = 0; i < size; ++i)
   {
      for (int j = i; j < size; ++j)
      {
         matrix[i * size + j] = matrix[j * size + i] = 1. * ((rand() % 2) ? 1 : -1) * (rand() % 200) / (rand()
% 200 + 1);
         if (i == j)
         {
            matrix[i * size + j] += 1.1 * size;
         }
      }
   }
}

void makeVectorZero(double * vector, const int size)
{
   for (int i = 0; i < size; ++i)
   {
      vector[i] = 0.;
   }
}




void makeVectorRandom(double * vector, const int size, const int limit)
```

```
{
   for (int i = 0; i < size; ++i)
   {
      vector[i] = rand() % limit;
   }
}

void printSLE(const double * matrix, const int matrixHeight, const int matrixWidth, const double *
vector)
{
   for (int i = 0; i < matrixHeight; ++i)
   {
      for (int j = 0; j < matrixWidth; ++j)
      {
         printf("%6.2lf ", matrix[i * matrixWidth + j]);
      }
      printf("| %6.2lf\n", vector[i]);
   }
}

void printMatrix(const double * matrix, const int matrixHeight, const int matrixWidth)
{
   for (int i = 0; i < matrixHeight; ++i)
   {
      for (int j = 0; j < matrixWidth; ++j)
      {
         printf("%6.2lf ", matrix[i * matrixWidth + j]);
      }
      printf("\n");
   }
}

void printVector(const double * vector, const int size)
{
   printf("[");
   for (int i = 0; i < size; ++i)
   {
```

```
        printf("%6.2lf, ", vector[i]);
    }
    printf("]\n");
}

void mulMatrixWithVector(const double * matrix, const int matrixHeight, const int matrixWidth, const double * vector, double * result)
{
    #pragma omp for schedule(SCHEDULE_MODE)
    for (int i = 0; i < matrixHeight; ++i)
    {
        result[i] = 0;
    }

    #pragma omp for collapse(2) schedule(SCHEDULE_MODE)
    for (int i = 0; i < matrixHeight; ++i)
    {
        for (int j = 0; j < matrixWidth; ++j)
        {
            #pragma omp atomic
            result[i] += matrix[i * matrixWidth + j] * vector[j];
        }
    }
}

void sumVectors(const double * v1, const double * v2, double * res, const int size)
{
    #pragma omp for
    for (int i = 0; i < size; ++i)
    {
        res[i] = v1[i] + v2[i];
    }
}

void subVectors(const double * v1, const double * v2, double * res, const int size)
{
    #pragma omp for schedule(SCHEDULE_MODE)
```

```
    for (int i = 0; i < size; ++i)
    {
        res[i] = v1[i] - v2[i];
    }
}

void mulVectorWithScalar(const double * vector, double * res, const int size, const double scalar)
{
    #pragma omp for schedule(SCHEDULE_MODE)
    for (int i = 0; i < size; ++i)
    {
        res[i] = vector[i] * scalar;
    }
}

int simpleIterationMethod(const double * matrix, const double * vector, double * result, const int size,
const double eps, const double tao)
{
    double * iterationVector = (double *) malloc(sizeof(double) * size); //* Ax^n - b

    double vectorBScalarSquare;
    double iterationVectorScalarSquare;
    double newEps;

    int iterationsCounter = 0;

    #pragma omp parallel
    {
        #pragma omp for reduction(+ : vectorBScalarSquare) schedule(SCHEDULE_MODE)
        for (int i = 0; i < size; ++i)
        {
            vectorBScalarSquare += vector[i] * vector[i];
        }

        mulMatrixWithVector(matrix, size, size, result, iterationVector);
        subVectors(iterationVector, vector, iterationVector, size);
```

```
    #pragma omp for reduction(+ : iterationVectorScalarSquare) schedule(SCHEDULE_MODE)
    for (int i = 0; i < size; ++i)
    {
        iterationVectorScalarSquare += iterationVector[i] * iterationVector[i];
    }

    #pragma omp single
    newEps = eps * eps * vectorBScalarSquare;

    #pragma omp barrier

    while (iterationsCounter < 10000 && iterationVectorScalarSquare >= newEps)
    {
        mulVectorWithScalar(iterationVector, iterationVector, size, tao);

        subVectors(result, iterationVector, result, size);

        mulMatrixWithVector(matrix, size, size, result, iterationVector);

        subVectors(iterationVector, vector, iterationVector, size);

        #pragma omp single
        ++iterationsCounter;

        #pragma omp single
        iterationVectorScalarSquare = 0.;

        #pragma omp barrier

        #pragma omp for reduction(+ : iterationVectorScalarSquare) schedule(SCHEDULE_MODE)
        for (int i = 0; i < size; ++i)
        {
            iterationVectorScalarSquare += iterationVector[i] * iterationVector[i];
        }
    }
}
```

```
    free(iterationVector);
    return iterationsCounter;
}


int main(int argc, char * argv[])
{
    srand(1678536002);

    int matrixSize = 3000;
    double tao = 0.000001;
    int maxThreads = 1;
    if (argc > 1)
    {
        maxThreads = atoi(argv[1]);
    }
    if (argc > 2)
    {
        matrixSize = atoi(argv[2]);
    }
    if (argc > 3)
    {
        sscanf(argv[3], "%lf", &tao);
    }

    omp_set_num_threads(maxThreads);

    double start = omp_get_wtime();

    double * A = (double *) malloc(sizeof(double) * matrixSize * matrixSize);
    double * b = (double *) malloc(sizeof(double) * matrixSize);
    double * x = (double *) malloc(sizeof(double) * matrixSize);

    makeMatrixRandomSymmetrical(A, matrixSize);
    makeVectorRandom(b, matrixSize, 100);
    makeVectorRandom(x, matrixSize, 10);
```

```
    printf("Answer found for %d iterations in %lf seconds\n", simpleIterationMethod(A, b, x, matrixSize, 1e-5, tao), omp_get_wtime() - start);

    free(A);
    free(b);
    free(x);
    return 0;
}
```