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, 21211 группы

Петрова Сергея Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Антон Юрьевич Кудинов

СОДЕРЖАНИЕ

ЦЕЛЬ

Ознакомиться с написанием параллельных программ при помощи OpenMP;

ЗАДАНИЕ

1. *Написать программу на языке C или C++, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида* $Ax = b$ *в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double;*

2. *Программу распараллелить с помощью OpenMP. Реализовать два варианта программы:*

   - *Для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for;*
   - *Создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм;*

   Уделить внимание тому, чтобы при запуске программы на различном числе OpenMP-потоков решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

3. *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;*

4. *Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков;*

5. *На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы;*

ВАРИАНТ ЗАДАНИЯ

Пусть есть система из N линейных алгебраических уравнений в виде $Ax = b$*, где A – матрица коэффициентов уравнений размером N×N, b – вектор правых частей размером N, x – вектор решений размером N.*

Решение системы уравнений итерационным методом состоит в выполнении следующих шагов:

1. *Задается* x^0 *– произвольное начальное приближение решения (вектор с произвольными начальными значениями);*
2. *Приближение многократно улучшается с использованием формулы вида* $x^{n+1} = f(x^n)$*, где функция f определяется используемым итерационным методом;*
3. *Процесс продолжается, пока не выполнится условие* $g(x^n) < \varepsilon$*, где функция g определяется используемым методом, а величина ε – требуемая точность;*

Метод простой итерации

В методе простой итерации преобразование решения на каждом шаге задается формулой: $x^{n+1} = x^n - \tau(Ax^n - b)$*, где τ – константа, параметр метода. Знак параметра* τ *зависит от задачи. Если с некоторым знаком решение начинает расходиться, то следует сменить его на противоположный.*

Критерий завершения счета $\frac{||Ax^n - b||_2}{||b||_2} < \varepsilon$*, где* $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$.

Исходные данные

Элементы главной диагонали матрицы A равны 2, остальные равны 1.

Все элементы вектора b равны N+1. В этом случае решением системы будет вектор, элементы которого равны 1.0.

Начальные значения элементов вектора x можно взять равными 0.

ОПИСАНИЕ РАБОТЫ

Описание выполненной работы

1. *Написал две версии программы, распараллеленной при помощи OpenMP, реализующей метод простых итераций;*
2. *Замерил время работы двух версий программы при использовании различного числа процессорных ядер (от 1 до 8 ядер);*
3. *Построил графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер;*
4. *Провел исследование на определение оптимальных параметров #pragma omp for schedule(...) при следующих параметрах:*
    - *OMP_NUM_THREADS = 4;*
    - *N = 4000;*
    - $\varepsilon = 10^{-10}$;
    - $\tau = 10^{-5}$;

Команды для компиляции и запуска

Команда для компиляции программы

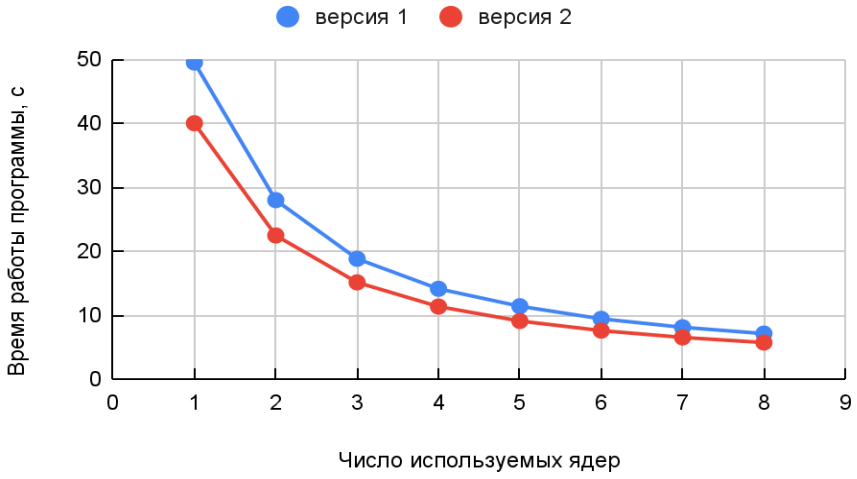
```
hpcuser265@clu:~> gcc
--std=c99
-fopenmp
-o openmp_<version>
openmp_<version>.c
-lm
```

Команда для постановки задачи, описанной в скрипте run.sh, в очередь на кластере

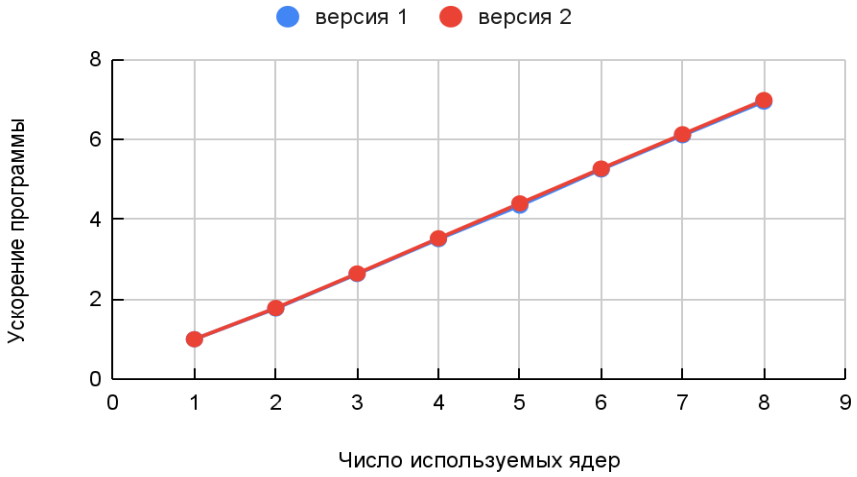
```
hpcuser265@clu:~> qsub run.sh
```

Результаты измерения

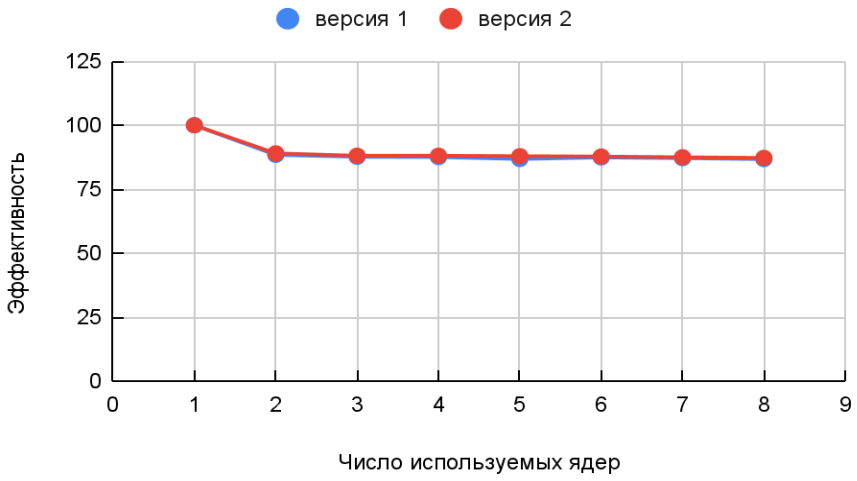
Зависимость времени работы программы от числа ядер



Зависимость ускорения программы от числа ядер



Зависимость эффективности распараллеливания программы от числа ядер



Время работы программы в зависимости от параметров опции schedule

Chunk size	*Time*		
	Static	*Dynamic*	*Guided*
default	*14,98037*	*33,337268*	*15,074164*
50	*15,108433*	*15,242135*	*15,204396*
100	*15,094884*	*15,195899*	*15,216028*
150	*15,740991*	*15,756213*	*15,782151*
200	*15,125272*	*15,20264*	*15,152964*
250	*15,142549*	*15,135584*	*15,314896*
500	*15,174383*	*15,415095*	*16,14119*
750	*22,333499*	*22,36985*	*22,358915*
1000	*15,425274*	*15,429925*	*15,445725*
1250	*18,976665*	*19,175632*	*19,170129*
1500	*22,945553*	*22,709319*	*22,86713*

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы:

- *ознакомился с написанием параллельных программ при помощи OpenMP;*
- *написал две версии 2 версии параллельной программы, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида* $Ax = b$ *в соответствии с методом простой итерации;*
- *построил графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер;*
- *провел исследование на определение оптимальных параметров #pragma omp for schedule(...);*

Анализируя результаты измерений, можно сделать следующие выводы:

- *Первая версия программы работает медленнее второй. Объясняется это тем, что в первой версии на каждой итерации создаются и уничтожаются параллельные секции, что занимает некоторое время, замедляющее работу программы;*
- *Использование параметров dynamic, guided опции schedule требует ручного подбора параметра chunk_size. В данной программе оптимальнее всего использование параметр static с параметром chunk_size по умолчанию.*

ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

openmp_slae_1.c

```c
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 5000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 1000000

void generate_A(double* A, int size);
void generate_x(double* x, int size);
void generate_b(double* b, int size);
double calc_norm_square(const double* vector, int size);
void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size);
void calc_next_x(const double* Axb, double* x,
                 double tau, int size);

int main(int argc, char **argv)
{
    int iter_count;
    double accuracy = EPSILON + 1;
    double b_norm;
    double start_time;
    double finish_time;
    double* A = malloc(sizeof(double) * N * N);
    double* x = malloc(sizeof(double) * N);
    double* b = malloc(sizeof(double) * N);
    double* Axb = malloc(sizeof(double) * N);

    generate_A(A, N);
    generate_x(x, N);
    generate_b(b, N);

    b_norm = sqrt(calc_norm_square(b, N));
    start_time = omp_get_wtime();

    for (iter_count = 0;
         accuracy > EPSILON && iter_count < MAX_ITERATION_COUNT;
         ++iter_count)
    {
        calc_Axb(A, x, b, Axb, N);
        calc_next_x(Axb, x, TAU, N);
        accuracy = sqrt(calc_norm_square(Axb, N)) / b_norm;
    }

    finish_time = omp_get_wtime();

    if (iter_count == MAX_ITERATION_COUNT)
        printf("Too many iterations\n");
```

```
    else
    {
        printf("Norm: %lf\n", sqrt(calc_norm_square(x, N)));
        printf("Time: %lf sec\n", finish_time - start_time);
    }

    free(A);
    free(x);
    free(b);
    free(Axb);

    return 0;
}

void generate_A(double* A, int size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; ++j)
            A[i * size + j] = 1;

        A[i * size + i] = 2;
    }
}

void generate_x(double* x, int size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++)
        x[i] = 0;
}

void generate_b(double* b, int size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++)
        b[i] = N + 1;
}

double calc_norm_square(const double* vector, int size)
{
    double norm_square = 0.0;

#pragma omp parallel for \
                    reduction(+ : norm_square)
    for (int i = 0; i < size; ++i)
        norm_square += vector[i] * vector[i];

    return norm_square;
}

void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size)
{
```

```
#pragma omp parallel for
    for (int i = 0; i < size; ++i)
    {
        Axb[i] = -b[i];
        for (int j = 0; j < N; ++j)
            Axb[i] += A[i * N + j] * x[j];
    }
}

void calc_next_x(const double* Axb, double* x,
                 double tau, int size)
{
#pragma omp parallel for
    for (int i = 0; i < size; ++i)
        x[i] -= tau * Axb[i];
}
```

openmp_slae_2.c

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 5000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 1000000

void set_matrix_part(int* line_counts, int* line_offsets,
                     int size, int thread_count);
void generate_A(double* A, int size);
void generate_x(double* x, int size);
void generate_b(double* b, int size);
double calc_norm_square(const double* vector, int size);
void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size);
void calc_next_x(const double* Axb, double* x,
                 double tau, int size);

int main(int argc, char **argv)
{
    int iter_count = 0;
    int thread_count = omp_get_max_threads();
    double accuracy = EPSILON + 1;
    double b_norm;
    double start_time;
    double finish_time;
    int* line_counts = malloc(sizeof(int) * thread_count);
    int* line_offsets = malloc(sizeof(int) * thread_count);
    double* A = malloc(sizeof(double) * N * N);
    double* x = malloc(sizeof(double) * N);
```

```
    double* b = malloc(sizeof(double) * N);
    double* Axb = malloc(sizeof(double) * N);

    set_matrix_part(line_counts, line_offsets, N, thread_count);

    generate_A(A, N);
    generate_x(x, N);
    generate_b(b, N);

    b_norm = sqrt(calc_norm_square(b, N));

    start_time = omp_get_wtime();

#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        for (iter_count = 0;
             accuracy > EPSILON && iter_count < MAX_ITERATION_COUNT;
             ++iter_count)
        {
            calc_Axb(A + line_offsets[thread_id] * N, x,
                     b + line_offsets[thread_id],
                     Axb + line_offsets[thread_id],
                     line_counts[thread_id]);
#pragma omp barrier

            calc_next_x(Axb + line_offsets[thread_id],
                        x + line_offsets[thread_id],
                        TAU, line_counts[thread_id]);

#pragma omp single
            accuracy = 0;

#pragma omp atomic
            accuracy +=
            calc_norm_square(Axb + line_offsets[thread_id],
                             line_counts[thread_id]);
#pragma omp barrier

#pragma omp single
            accuracy = sqrt(accuracy) / b_norm;
        }
    }

    finish_time = omp_get_wtime();

    if (iter_count == MAX_ITERATION_COUNT)
        printf("Too many iterations\n");
    else
    {
        printf("Norm: %lf\n", sqrt(calc_norm_square(x, N)));
        printf("Time: %lf sec\n", finish_time - start_time);
    }

    free(A);
```

```
    free(x);
    free(b);
    free(Axb);

    return 0;
}

void set_matrix_part(int* line_counts, int* line_offsets,
                     int size, int thread_count)
{
    int offset = 0;
    for (int i = 0; i < thread_count; ++i)
    {
        line_counts[i] = size / thread_count;

        if (i < size % thread_count)
            ++line_counts[i];

        line_offsets[i] = offset;
        offset += line_counts[i];
    }
}

void generate_A(double* A, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; ++j)
            A[i * size + j] = 1;

        A[i * size + i] = 2;
    }
}

void generate_x(double* x, int size)
{
    for (int i = 0; i < size; i++)
        x[i] = 0;
}

void generate_b(double* b, int size)
{
    for (int i = 0; i < size; i++)
        b[i] = N + 1;
}

double calc_norm_square(const double* vector, int size)
{
    double norm_square = 0.0;
    for (int i = 0; i < size; ++i)
        norm_square += vector[i] * vector[i];

    return norm_square;
}
```

```
void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size)
{
    for (int i = 0; i < size; ++i)
    {
        Axb[i] = -b[i];
        for (int j = 0; j < N; ++j)
            Axb[i] += A[i * N + j] * x[j];
    }
}

void calc_next_x(const double* Axb, double* x, double tau, int size)
{
    for (int i = 0; i < size; ++i)
        x[i] -= tau * Axb[i];
}
```

openmp_slae_schedule.c

```
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define N 4000
#define EPSILON 1E-10
#define TAU 1E-5
#define MAX_ITERATION_COUNT 1000000

void generate_A(double* A, int size);
void generate_x(double* x, int size);
void generate_b(double* b, int size);
double calc_norm_square(const double* vector, int size);
void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size);
void calc_next_x(const double* Axb, double* x,
                 double tau, int size);

int main(int argc, char **argv)
{
    int iter_count;
    double accuracy = EPSILON + 1;
    double b_norm;
    double start_time;
    double finish_time;
    double* A = malloc(sizeof(double) * N * N);
    double* x = malloc(sizeof(double) * N);
    double* b = malloc(sizeof(double) * N);
    double* Axb = malloc(sizeof(double) * N);

    generate_A(A, N);
    generate_x(x, N);
    generate_b(b, N);
```

```
    b_norm = sqrt(calc_norm_square(b, N));

    start_time = omp_get_wtime();

    for (iter_count = 0;
         accuracy > EPSILON && iter_count < MAX_ITERATION_COUNT;
         ++iter_count)
    {
        calc_Axb(A, x, b, Axb, N);
        calc_next_x(Axb, x, TAU, N);
        accuracy = sqrt(calc_norm_square(Axb, N)) / b_norm;
    }

    finish_time = omp_get_wtime();

    if (iter_count == MAX_ITERATION_COUNT)
        printf("Too many iterations\n");
    else
    {
        printf("Norm: %lf\n", sqrt(calc_norm_square(x, N)));
        printf("Time: %lf sec\n", finish_time - start_time);
        printf("Iteration: %d\n", iter_count);
    }

    free(A);
    free(x);
    free(b);
    free(Axb);

    return 0;
}

void generate_A(double* A, int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; ++j)
            A[i * size + j] = 1;

        A[i * size + i] = 2;
    }
}

void generate_x(double* x, int size)
{
    for (int i = 0; i < size; i++)
        x[i] = 0;
}

void generate_b(double* b, int size)
{
    for (int i = 0; i < size; i++)
        b[i] = N + 1;
}
```

```
double calc_norm_square(const double* vector, int size)
{
    double norm_square = 0.0;

#pragma omp parallel for schedule(runtime) \
                    reduction(+ : norm_square)
    for (int i = 0; i < size; ++i)
        norm_square += vector[i] * vector[i];

    return norm_square;
}

void calc_Axb(const double* A, const double* x,
              const double* b, double* Axb, int size)
{
#pragma omp parallel for schedule(runtime)
    for (int i = 0; i < size; ++i)
    {
        Axb[i] = -b[i];
        for (int j = 0; j < N; ++j)
            Axb[i] += A[i * N + j] * x[j];
    }
}

void calc_next_x(const double* Axb, double* x, double tau, int size)
{
#pragma omp parallel for schedule(runtime)
    for (int i = 0; i < size; ++i)
        x[i] -= tau * Axb[i];
}
```

run.sh

```
#!/bin/bash

#PBS -l walltime=00:05:00
#PBS -l select=1:ncpus=8

cd $PBS_O_WORKDIR
echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
echo

for (( i = 1; i <= 8; i++))
do
    echo "Number of threads: $i"
    OMP_NUM_THREADS=$i ./openmp_slae_<version>
    echo
done
```

run_schedule.sh

```
#!/bin/bash

#PBS -l walltime=00:20:00
#PBS -l select=1:ncpus=4:ompthreads=4

cd $PBS_O_WORKDIR
echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
echo

echo "Number of threads: $OMP_NUM_THREADS"

echo "Static default:"
OMP_SCHEDULE="static" ./openmp_slae_schedule
echo

for (( i = 1; i <= 10; i++ ))
do
    echo "Static $(( i * 250 ))"
    OMP_SCHEDULE="static,$(( i * 250 ))" ./openmp_slae_schedule
    echo
done

echo "Dynamic default:"
OMP_SCHEDULE="dynamic" ./openmp_slae_schedule
echo

for (( i = 1; i <= 10; i++ ))
do
    echo "Dynamic $(( i * 250 ))"
    OMP_SCHEDULE="dynamic,$(( i * 250 ))" ./openmp_slae_schedule
    echo
done

echo "Guided default:"
OMP_SCHEDULE="guided" ./openmp_slae_schedule
echo

for (( i = 1; i <= 10; i++ ))
do
    echo "Guided $(( i * 250 ))"
    OMP_SCHEDULE="guided,$(( i * 250 ))" ./openmp_slae_schedule
    echo
done
```