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»

студентки 2 курса, 19202 группы

Хаецкой Дарьи Владимировны

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.Ю. Власенко

Новосибирск 2021

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

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.

ОПИСАНИЕ РАБОТЫ

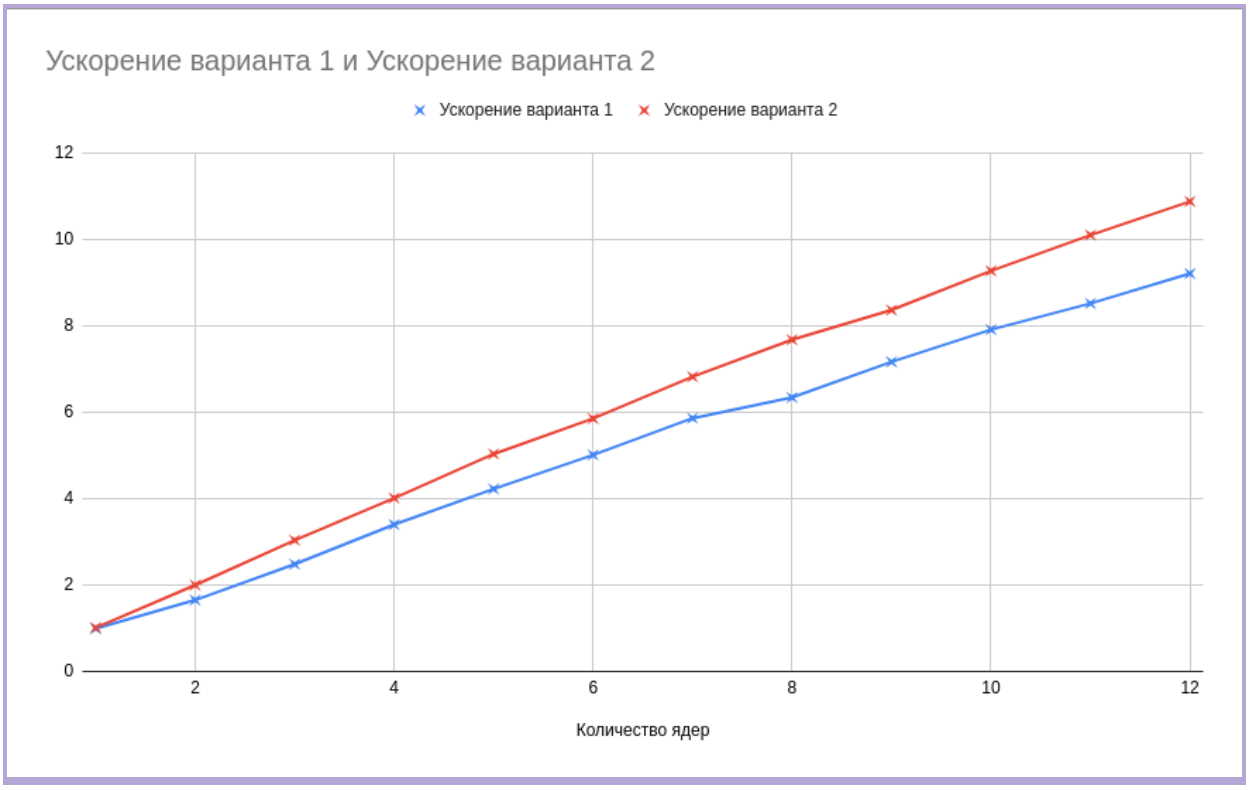
Написано два варианта распараллеливания итерационного алгоритма:

- с использованием директив `#pragma omp parallel for` во всех функциях
- одна параллельная секция `#pragma omp parallel`, охватывающая весь итерационный алгоритм

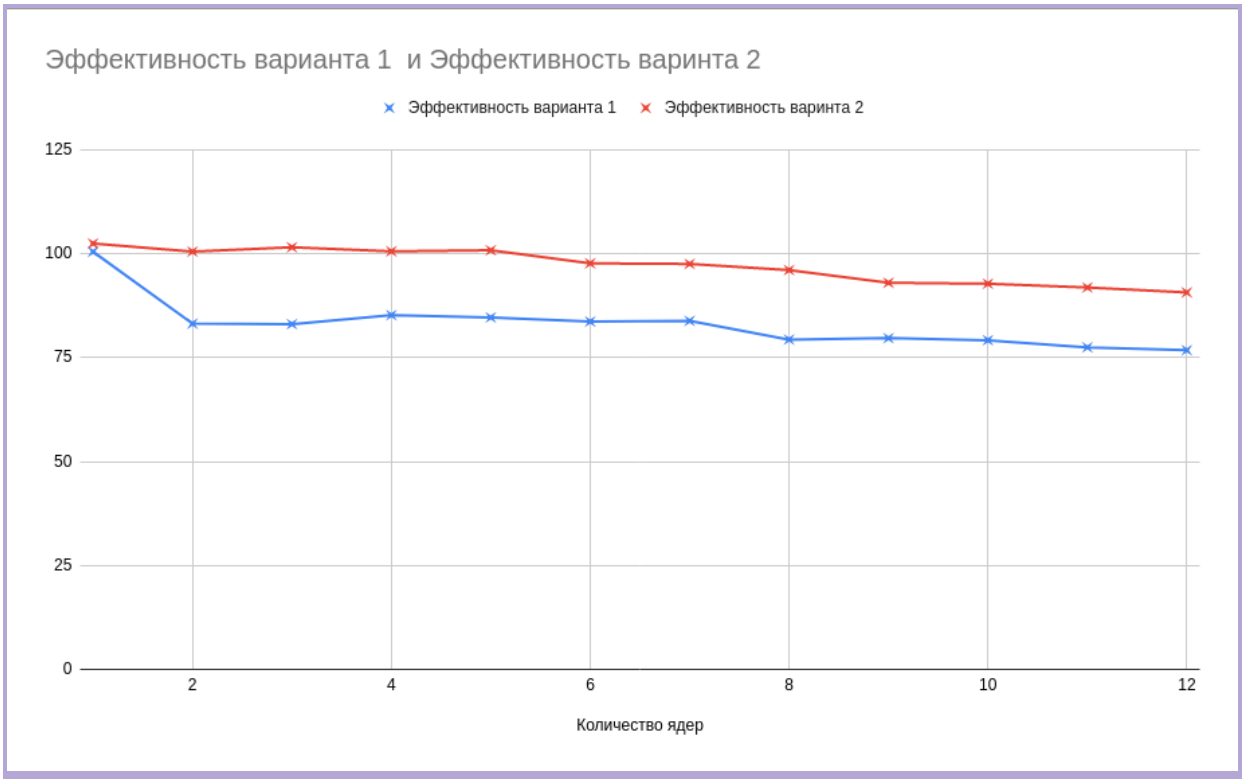
В ходе запуска на кластере были получены следующие результаты:
(все замеры проводились на фиксированной матрице 2400*2400 для чистоты эксперимента, эпсилон был взят равным 10^-7)

Количество ядер	Вариант 1	Вариант 2	Ускорение 1	Эффективность 1	Ускорение 1	Эффективность 1
1	44.811691	43.943158	1.004202229	100.4202229	1.024050206	102.4050206
2	27.063089	22.396765	1.662781362	83.13906812	2.009218742	100.4609371
3	18.066414	14.778222	2.490809742	83.02699141	3.045021248	101.5007083
4	13.207218	11.191506	3.407227775	85.18069437	4.02090657	100.5226642
5	10.634772	8.930026	4.231402422	84.62804844	5.039179057	100.7835811
6	8.966503	7.681276	5.018678965	83.64464942	5.858401651	97.64002752
7	7.670834	6.593336	5.866376459	83.80537799	6.825073074	97.50104391
8	7.091042	5.858641	6.346034899	79.32543623	7.680962189	96.01202736
9	6.277331	5.378182	7.168651773	79.65168636	8.367139677	92.96821863
10	5.685173	4.852317	7.915326411	79.15326411	9.273920067	92.73920067
11	5.28183	4.454663	8.519774396	77.45249451	10.10177425	91.83431139
12	4.88565	4.13695	9.210647509	76.7553959	10.87757889	90.64649077

Сравнительный график ускорения для первого и второго вариантов:



Сравнительный график эффективности для первого и второго вариантов:



Как видно из данных графиков, ускорение в обоих вариантах растет линейно, однако в первом варианте растёт оно медленнее.

Эффективность, на удивление, уменьшается довольно медленно у обеих реализаций, и опять же, первый вариант программы уступает второму.

Проигрыш первого варианта, на мой взгляд, связан с тем, что для реализации второго пришлось вынести операции в тело основного цикла while, соответственно время на передачу и возврат аргументов на каждой итерации не тратится.

Сравнение времени работы программы при разных параметрах опции schedule:

Были произведены замеры времени работы в зависимости от параметров опции schedule (распределение итераций у pragma omp parallel for)

- static - блочно-циклическое распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок — следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера и полученные порции итераций распределяются между нитями.

  Значение chunk не было указано, поскольку один из главных принципов параллельного программирования - равномерное распределение работы, и поделить количество итераций поровну является самым оптимальным решением в данной ситуации.

- dynamic - динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны.

- guided - динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству еще не распределенных итераций, деленному на количество нитей, выполняющих цикл.

2400*2400	static	dynamic	guided
epsilon = 10 ^(-7)	11.383713	25.027811	11.133074
4 threads	chunk = null *	chunk = 1 (default)	chunk = 1 (default)
iterations: 1167		11.073902	11.756149
		chunk = 100	chunk = 100
		15.883861	16.153293
		chunk = 400	chunk = 400
	* количество итераций разбивается поровну		

Как видно из сравнительной таблицы, dynamic и guided режимы требуют ручного подбора оптимального размера чанка. Режим static же с равномерным делением итераций работает стабильно хорошо на разных размерах матриц.

Приложение 1. *Листинг программы вариант 1*

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

const long int N = 2400;
const double epsilon = pow(10, -7);
const float parameter =  0.001;

double EuclideanNorm(const double* u){
  double norm = 0;
  #pragma omp parallel for reduction (+:norm)
  for (int i = 0; i < N; i++){
     norm += u[i]*u[i];
  }
  return sqrt(norm);
}

void sub(double* a, double* b, double* c){
#pragma omp parallel for
  for (int i = 0; i < N; i++) {
     c[i] = a[i] - b[i];
  }
}

void mul(double* A, double* b, double* result, int n) {
  unsigned int i, j;
  #pragma omp parallel for private (j)
  for(i = 0; i < n; i++) {
     result[i] = 0;
     for(j = 0; j < n; j++) {
        result[i] += A[i * n + j] * b[j];
     }
  }
}


void scalMul(double* A, double tau){
  int i;
  #pragma omp parallel for
  for (i = 0; i < N; ++i) {
     A[i] = A[i] * tau;
  }
}

void printMatrix(double* A){
  printf("\n");

  for(unsigned int i = 0; i < N; i++) {
     for(unsigned int j = 0; j < N; j++) {
        printf("%lf ", A[i * N + j]);
     }
     printf("\n");
  }
  printf("\n");
}

void printVec(double* A, int n){
  printf("\n");
  printf("\n");
  for (int i = 0; i < n; ++i){
     printf("%lf ", A[i]);
  }
```

```
  printf("\n");
  printf("\n");
}

double drand(double low, double high) {
  double f = (double)rand() / RAND_MAX;
  return low + f * (high - low);
}

double rand_double(){
  return (double)rand()/RAND_MAX*4.0 - 2.0;
}

void generate_matrix(double* matrix) {
  double rand_value = 0;

  for(int i = 0; i < N; i++){
    for(int j = 0; j < i; j++){
      matrix[i*N + j] = matrix[j*N + i];
    }
    for(int j = i; j < N; j++){
      matrix[i * N + j] = rand_double();
      if(i == j){
        matrix[i*N + j] = fabs(matrix[i*N + j]) + 124.0;
      }
    }
  }
}

int main(int argc, char *argv[]) {
  srand(100);
  omp_set_num_threads(2);
  double* prevX = (double*)malloc(N * sizeof(double));
  double* Ax = (double*)malloc(N * sizeof(double));
  double tau = parameter;

  double normAxb = 0; // ||A*xn - b||
  double normb = 0;
  double saveRes = 0;
  double res = 0;
  double lastres = 0;

  double start, end;

  double* b = (double*)malloc(N * sizeof(double));
  double* nextX = (double*)malloc(N * sizeof(double));
  double* A = (double*)malloc(N * N * sizeof(double));

  for (long i = 0; i < N; i++) {
    prevX[i] = rand_double();
    b[i] = rand_double();

  }

  generate_matrix(A);

  mul(A, prevX, Ax, N); // A*xn

  sub(Ax, b, Ax); // A*xn - b
  normAxb = EuclideanNorm(Ax); // ||A*xn - b||
  normb = EuclideanNorm(b);
  scalMul(Ax, tau); // TAU*(A*xn - b)
  sub(prevX, Ax, nextX); // xn - TAU * (A*xn - b)
  saveRes = normAxb / normb;
  res = normAxb / normb;
  lastres = res;
```

```
  int countIt = 1;

  start = omp_get_wtime();

  while (res > epsilon) {
    for (long i = 0; i < N; i++) {
      prevX[i] = nextX[i];
    }

    mul(A, prevX, Ax, N); //A*xn
    sub(Ax, b, Ax); //A*xn - b
    normAxb = EuclideanNorm(Ax); // ||A*xn - b||
    scalMul(Ax, tau); // TAU*(A*xn - b)
    sub(prevX, Ax, nextX); // xn - TAU * (A*xn - b)
    res = normAxb / normb;
    countIt++;

    if ((countIt > 100000 && lastres > res) || res == INFINITY) {
      if (tau < 0) {
        printf("Does not converge\n");
        res = 0;
      } else {
        tau = -0.01;
        countIt = 0;
        res = saveRes;
      }
    }
    lastres = res;
    printf("init res = %lf, epsil = %lf\n", res, epsilon);
  }

  end = omp_get_wtime();
  printf("time: %lf\n", end - start);

  printf("res = %lf, iterations: %d\n", res, countIt);
  free(Ax);
  free(nextX);
  free(prevX);
  free(b);
  free(A);
  return 0;
}
```

Приложение 2. *Листинг программы вариант 2*

```
#include <iostream>
#include <cmath>
#include <omp.h>
#include <cstdlib>
#include <cstddef>
#include <stdio.h>

const long int N = 2400;
const double epsilon = pow(10, -7);
const float parameter =  0.001;

double EuclideanNorm(const double* u){
  double norm = 0;
  for (int i = 0; i < N; i++){
    norm += u[i]*u[i];
  }
  return sqrt(norm);
}

void sub(double* a, double* b, double* c){
  for (int i = 0; i < N; i++) {
    c[i] = a[i] - b[i];
  }
}

void mul(double* A, double* b, double* result, int n) {
  unsigned int i, j;
  for(i = 0; i < n; i++) {
    result[i] = 0;
    for(j = 0; j < n; j++) {
      result[i] += A[i * n + j] * b[j];
    }
  }
}


void scalMul(double* A, double tau){
  int i;
  for (i = 0; i < N; ++i) {
    A[i] = A[i] * tau;
  }
}


double drand(double low, double high) {
  double f = (double)rand() / RAND_MAX;
  return low + f * (high - low);
}

double rand_double(){
  return (double)rand()/RAND_MAX*4.0 - 2.0;
}

void generate_matrix(double* matrix) {
  double rand_value = 0;

  for(int i = 0; i < N; i++){
    for(int j = 0; j < i; j++){
      matrix[i*N + j] = matrix[j*N + i];
    }
    for(int j = i; j < N; j++){
      matrix[i * N + j] = rand_double();
      if(i == j){
```

```
            matrix[i*N + j] = fabs(matrix[i*N + j]) + 124.0;
        }
      }
    }
}

int main(int argc, char *argv[]) {
  srand(100);
  omp_set_num_threads(8);
  double* prevX = (double*)malloc(N * sizeof(double));
  double* Ax = (double*)malloc(N * sizeof(double));
  double tau = parameter;

  double normAxb = 0; // ||A*xn - b||
  double normb = 0;
  double saveRes = 0;
  double res = 0;
  double lastres = 0;

  double start, end;

  double* b = (double*)malloc(N * sizeof(double));
  double* nextX = (double*)malloc(N * sizeof(double));
  double* A = (double*)malloc(N * N * sizeof(double));

  for (long i = 0; i < N; i++) {
     prevX[i] = rand_double();
     b[i] = rand_double();

  }

  generate_matrix(A);

  mul(A, prevX, Ax, N); // A*xn

  sub(Ax, b, Ax); // A*xn - b
  normAxb = EuclideanNorm(Ax); // ||A*xn - b||
  normb = EuclideanNorm(b);
  scalMul(Ax, tau); // TAU*(A*xn - b)
  sub(prevX, Ax, nextX); // xn - TAU * (A*xn - b)
  saveRes = normAxb / normb;
  res = normAxb / normb;
  lastres = res;

  int countIt = 1;

  start = omp_get_wtime();
  unsigned int i, j;
  double norm = 0;

#pragma omg parallel private(i, j)
  {
     while (res > epsilon) {
        #pragma omp single
        {
           for (long i = 0; i < N; i++) {
              prevX[i] = nextX[i];
           }
        }

        #pragma omp parallel for private (j)
        for(i = 0; i < N; i++) {
           Ax[i] = 0;
           for(j = 0; j < N; j++) {
              Ax[i] += A[i * N + j] * prevX[j];
           }
```

```
        }

        #pragma omp parallel for
        for (i = 0; i < N; i++) {
            Ax[i] = Ax[i] - b[i];
        }

        #pragma omp atomic write
        norm = 0;

        #pragma omp parallel for reduction (+:norm)
        for (i = 0; i < N; i++){
            norm += Ax[i]*Ax[i];
        }

        #pragma omp atomic write
        norm = sqrt(norm);

        #pragma omp parallel for
        for (i = 0; i < N; ++i) {
            Ax[i] = Ax[i] * tau;
        }

        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
            nextX[i] = prevX[i] - Ax[i];
        }

        #pragma omp critical
        {
            res = norm / normb;
            countIt++;

            if ((countIt > 100000 && lastres > res) || res == INFINITY) {
                if (tau < 0) {
                    printf("Does not converge\n");
                    res = 0;
                } else {
                    tau = (-1)*parameter;
                    countIt = 0;
                    res = saveRes;
                }
            }
            lastres = res;
        }
    }
  }
  end = omp_get_wtime();
  printf("time: %lf\n", end - start);

  printf("res = %lf, iterations: %d\n", res, countIt);
  free(Ax);
  free(nextX);
  free(prevX);
  free(b);
  free(A);
  return 0;
}
```

Вывод

OpenMP - простой и мощный инструмент для написания параллельных программ. Множество различных директив предоставляют широкий инструментал для программиста, а инкапсуляция позволяет избежать множества ошибок (и частично - дебага параллельных программ 🙂)

При грамотном использовании, OpenMP может избавить от необходимости “ручного” распараллеливания, хоть и можно сказать, что здесь мы имеем меньше свободы в реализации.