САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил(а): Ступников Александр Сергеевич

студ. гр. М3135

Санкт-Петербург

2020

Цель работы: знакомство со стандартом распараллеливания команд OpenMP.

Инструментарий и требования к работе: рекомендуется использовать C, C++. Возможно использовать Python и Java.

Теоретическая часть

Принципы работы OpenMP

OpenMP API – стандарт для распараллеливания программ на C, C++ и Fortran. Описывает набор директив компилятора, библиотечных процедур и переменных окружения.

OpenMP использует fork-join модель параллельного исполнения (см. рис. 1). Программа, написанная с помощью OpenMP API, начинает выполняться в одном потоке, называемом главным потоком(master thread). Код в главном потоке выполняется последовательно, пока не будет найдена первая конструкция parallel (parallel construction). Как только это произошло, главный поток создаёт совокупность потоков, называемую командой потоков (team of threads). Каждый поток в команде выполняет код в параллельном регионе (блок кода, связанный с конструкцией parallel) программы, за исключением кода внутри конструкций распределения работы (work-sharing constructs) (о них немного позже). В конце каждого параллельного региона все потоки синхронизируются и только главный поток продолжает выполнение. Все локальные переменные внутри потоков уничтожаются. Таким образом, для сохранения результата выполнения потока необходимо использовать «внешнюю» (shared) переменную, то есть переменную общую для всех потоков.

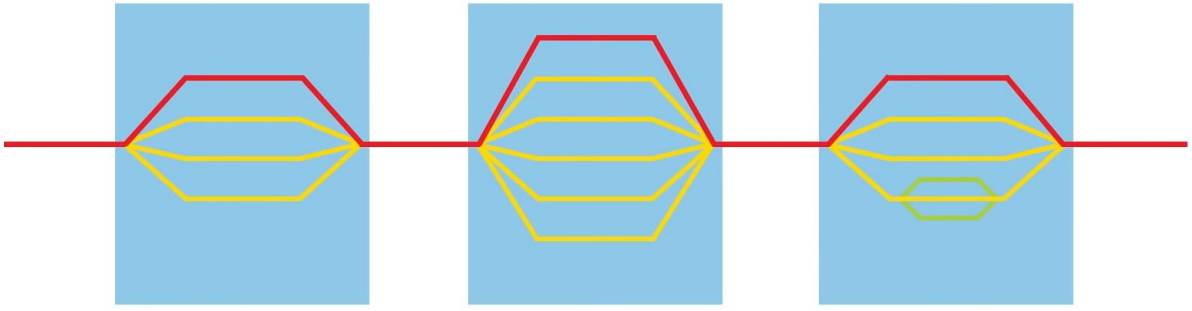

Рисунок №1 – Fork-join модель (красным обозначен главный поток)

В программе может быть определено любое количество конструкций parallel. В итоге программа может «ветвиться и объединяться» (fork and join) много раз за время исполнения.

Синтаксис директив OpenMP API C++ неформально выглядит так:

#pragma omp directive-name [clause[[,] clause]...] new-line

Теперь рассмотрим некоторые основные функции (runtime library routines), конструкции (constructs) и операторы (clauses), входящие в OpenMP API.

Функции

void omp_set_num_threads(int nthreads): Запрашивает nthreads потоков для следующих параллельных регионов. Заметим, что запрошенное число потоков может отличаться от предоставленного.

int omp_get_num_threads(): Возвращает число потоков в параллельном регионе. При вызове вне параллельного региона возвращает 1.

int omp_get_max_threads(): Возвращает число, не меньшее, чем количество потоков, которые будут предоставлены, если в этой точке кода будет определён параллельный регион.

int omp_get_thread_num(): Возвращает номер (id) потока, исполняющегося в данный момент внутри параллельного региона.

Конструкции распределения работы (Work-sharing constructs)

Используются, чтобы независимо распределить работу между одним или несколькими потоками.

omp for: распределяет итерации цикла между потоками.

Конструкции распределения работы не создают потоков. Чтобы создать потоки необходимо вызвать omp parallel. Таким образом, код, выполняющий цикл в несколько потоков может выглядеть следующим образом:

```
#pragma omp parallel
{
	#pragma omp for
	for (int i = 0; i < n; i++) {
		//some code
	}
}
```

Это может быть упрощено до:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
      //some code
}
```

Стоит отметить, что work-sharing конструкции лишь упрощают работу с OpenMP. При желании можно обойтись без них. Например, два следующих фрагмента кода в целом эквивалентны. В первом фрагменте итерации цикла распределены между потоками вручную, во втором фрагменте – с помощью omp for.

```
#pragma omp parallel
{
      int id, i, nthrds, istart, iend;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      istart = id * n / nthrds;
      iend = (id + 1) * n / nthrds;
      if (id == nthrds - 1) iend = n;
      for (i = istart; i < iend; i++) {
            a[i] = a[i] + b[i];
      }
}
```

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
      a[i] = a[i] + b[i];
}
```

Операторы синхронизации потоков (Synchronization clauses)

Синхронизация используется для защиты общих данных потоков (shared data) от race conditions. Заметим, что операторы синхронизации заметно влияют на производительность программы. При их частом вызове параллельный код может работать даже медленнее, чем последовательный.

critical: следующий блок кода может выполняться одновременно только одним потоком.

atomic: обращение к памяти для следующей инструкции будет выполнено атомарно (atomically). Компилятор может использовать специальные аппаратные инструкции для лучшей производительности в сравнении с

critical. Заметим, что atomic имеет смысл применять к простым операциям обновления (например, `x++, x--, --x, x &= 3`).

barrier: каждый поток ждёт пока все потоки в команде достигнут этой точки. Конструкции распределения работы, а также параллельные регионы по умолчанию имеют barrier в конце.

Операторы планирования (Scheduling clauses)

schedule(type, [chunk]): итерации в work-sharing конструкциях распределяют работу между потоками согласно методу планирования (type). Существует четыре основных метода планирования:

static: итерации между всеми потоками распределяются заранее, каждый поток выполняет одинаковое число итераций. Опциональный параметр chunk определяет размер блока данных, который будет выполнен каждым потоком.

dynamic: работа между потоками распределяется динамически во время исполнения программы. Размеры блоков данных, выполняемых разными потоками, могут отличаться. Такое планирование имеет смысл применять, если объём работы в зависимости от потока может значительно отличаться.

guided: блоки итераций динамически распределяются между потоками. В начале размер блока большой, но во время выполнения вычислений он уменьшается, пока не достигает размера chunk.

Определитель матрицы

Для нахождения определителя матрицы приведём её к верхнетреугольному виду. Для этого воспользуемся методом Гаусса (Gaussian elimination a.k.a. row reduction).

Пусть дана матрица A размером $n \times n$, $A = (A_1, A_2, \dots, A_n) = (S_1, S_2, \dots, S_2)^T = (a_{i,j})_{i=1}^{n}{}_{j=1}^{n}$. Обнулим элементы A, лежащие под главной диагональю. Операция добавления к одной из строк матрицы другой строки, умноженной на некоторое число, не меняет определитель. А операция перестановки двух строк матрицы меняет знак определителя на противоположный. Используя эти факты, можно привести любую матрицу к верхнетреугольной форме. Для этого сначала обнулим все элементы столбца A_1, лежащие ниже главной диагонали. Если $a_{1,1} = 0$, то необходимо поменять местами ряд S_1 и некоторый ряд S_j, в котором $a_{j,1} \neq 0$. Если такого ряда S_j не нашлось, то определитель A равен 0, заканчиваем алгоритм. Иначе для всех $i = 2, 3, \dots, n$ вычтем из ряда S_i ряд S_1,

умноженный на $\frac{a_{i,1}}{a_{1,1}}$, то есть коэффициент, позволяющий обнулить первый элемент ряда i. После этого все элементы столбца A_1, лежащие ниже главной диагонали, то есть $a_{2,1}, a_{2,1}, \dots, a_{n,1}$ обратятся в 0.

Теперь можно мысленно вычеркнуть первую строку и первый столбец из матрицы A и обнулить первый столбец в получившейся матрице. Для этого применим тот же алгоритм, который использовался для обнуления первого столбца матрицы A. Подобные действия нужно повторять до тех пор, пока после вычёркивания не останется матрица размера 1×1. После этого возвратим все вычеркнутые строки и столбцы, получится верхнетреугольная матрица U размера $n \times n$, определитель которой $\det(U)$ равен произведению элементов, лежащих на главной диагонали U (свойство верхнетреугольной матрицы), и равен по модулю определителю матрицы A. Чтобы найти знак определителя A, определим число раз, когда в процессе выполнения алгоритма нужно было переставлять строки местами. Если это число чётное, то определитель матрицы A равен $\det(U)$, если нечётное, то определитель A равен $-\det(U)$.

Практическая часть

Алгоритм, представленный в коде, несколько отличается от описанного в теоретической части, однако по большому счёту делает то же самое. Функция LUPDecomposeParallel не обнуляет элементы матрицы, лежащие ниже главной диагонали, таким образом работа алгоритма немного ускоряется. В сущности, после преобразования матрицы a, в ней будет храниться её LU-разложение. Но в данном случае это неважно, так как для вычисления определителя необходимы только элементы главной диагонали получившейся матрицы и число перестановок её строк.

Сначала рассмотрим последовательную версию программы. Самая важная функция это `int LUPDecompose(double** a, int n, double tol)`, она принимает исходную матрицу A, хранящуюся в двумерном динамическом массиве a, размер матрицы n и погрешность tol, меньше которой число в матрице считается равным 0. Функция возвращает число перестановок строк матрицы A, сделанных в процессе выполнения алгоритма. Также функция изменяет динамический массив a таким образом, что после выполнения алгоритма в нём хранится матрица B, такая что не

ниже её главной диагонали находится верхнетреугольная матрица U, а ниже главной диагонали – нижнетреугольная матрица $L - E$. Другими словами $B = (L - E) + U$, $P * A = L * U$, где P – матрица перестановок (она не хранится в программе, хранится только число перестановок). Далее представлен исходный код функции `LUPDecompose`, снабжённый подробными комментариями.

```
int LUPDecompose(double** a, int n, double tol) {
    int i, j, k, imax;
    double maxA, * ptr, absA;
    //Число перестановок
    int p = 0;

    //Итерируемся по столбцам матрицы
    for (i = 0; i < n; i++) {
        maxA = fabs(a[i][i]);
        imax = i;
        //Проверяем, что i-ый элемент в столбце i не равен 0
        if (maxA < tol) {
            //Находим imax - номер строки, в которой i-ый элемент максимален
            for (k = i; k < n; k++) {
                if ((absA = fabs(a[k][i])) > maxA) {
                    maxA = absA;
                    imax = k;
                }
            }

            //Если в i-ом стоблце все элементы равны нулю, заканчиваем алгоритм
            if (maxA < tol) return -1;

            //Меняем i-ую строку с строкой номер imax
            ptr = a[i];
            a[i] = a[imax];
            a[imax] = ptr;
            //Увеличиваем число перестановок на 1
            p++;
        }
        //Итерируемся по строкам, лежащим ниже строки i
        for (j = i + 1; j < n; j++) {
            //Вычисляем коэффициент, на который необходимо умножить
            //элементы i-ой строки перед тем, как вычесть их из j-ой строки,
            //чтобы получить в i-ом столбце нули
            a[j][i] /= a[i][i];
            for (k = i + 1; k < n; k++) {
                //Вычитаем из элементов j-ой строки элементы i-ой строки,
                //умноженные на коэффициент
                a[j][k] -= a[j][i] * a[i][k];
            }
        }
    }

    return p;
}
```

После применения к матрице A функции `LUPDecompose` можно вызвать функцию `double LUPDeterminant(double** a, int p, int n)`. Она вычисляет и возвращает определитель A, перемножая элементы матрицы из массива a, лежащие на его главной диагонали и умножая это произведение на -1^p, где p – число перестановок. Если $p = -1$, то функция возвращает 0, так как это значит, что в матрице обнулился некоторый столбец.

Теперь рассмотрим параллельный код:

```
int LUPDecomposeParallel(double** a, int n, double tol) {
    int i, j, k, imax;
    double maxA, * ptr, absA;
    int p = 0;

    for (i = 0; i < n; i++) {
        maxA = fabs(a[i][i]);
        imax = i;
        if (maxA < tol) {
            for (k = i; k < n; k++) {
                if ((absA = fabs(a[k][i])) > maxA) {
                    maxA = absA;
                    imax = k;
                }
            }

            if (maxA < tol) return -1;

            ptr = a[i];
            a[i] = a[imax];
            a[imax] = ptr;
            p++;
        }
        #pragma omp parallel for schedule(SHEDULE)
        for (j = i + 1; j < n; j++) {
            a[j][i] /= a[i][i];
            for (int p = i + 1; p < n; p++) {
                a[j][p] -= a[j][i] * a[i][p];
            }
        }
    }

    return p;
}
```

Параллельность достигается за счёт того факта, что вычитание i-ой строки, умноженной на коэффициент, из строк с номерами $j = i + 1, i + 2, \dots, n - 1$ при обработке i-го столбца происходят независимо друг от друга. Следовательно, эти вычисления можно сделать параллельно.

Несложно заметить, что параллельный код отличается от последовательного лишь в двух строчках:

```
#pragma omp parallel for schedule(SHEDULE)
```

```
for (int p = i + 1; p < n; p++) {
```

Первая строчка – это work-sharing конструкция, совмещённая с конструкцией создания потоков parallel. Также в директиве присутствует оператор schedule, который на основании значения, хранящего в константе SHEDULE, определяет, как именно итерации цикла for будут распределены между потоками. Вторая строчка заменяет shared переменную k на переменную p, локальную для каждого потока. Это сделано для предотвращения raise condition.

Теперь рассмотрим графики времени выполнения программы. Тесты проводились на случайно сгенерированной матрице размером 1700×1700, состоящей из чисел в формате double с абсолютным значением, не превышающим 10. Всё время вычислялось как среднее арифметическое времени выполнения 10 тестов. Как видно из фрагмента исходного кода, приведённого ниже, учитывалось только время, требующееся непосредственно для вычисления определителя матрицы. Для измерений использовалась функция omp_get_wtime.

```
double start_time = omp_get_wtime();
int p = LUPDecomposeParallel(a, n, TOL);
double det = LUPDeterminant(a, p, n);
double end_time = omp_get_wtime();
```

На графике 1 изображена зависимость времени выполнения программы от числа потоков. В качестве метода schedule использован static. Значение OpenMP off соответствует выключенному OpenMP. Значение auto соответствует числу потоков по умолчанию (в моём случае 12). Больше 12 потоков запрашивать не имеет смысла, так как на моём процессоре только 6 физических и 6 логических ядер, именно число 12 возвращает функция omp_get_max_threads на моей системе.

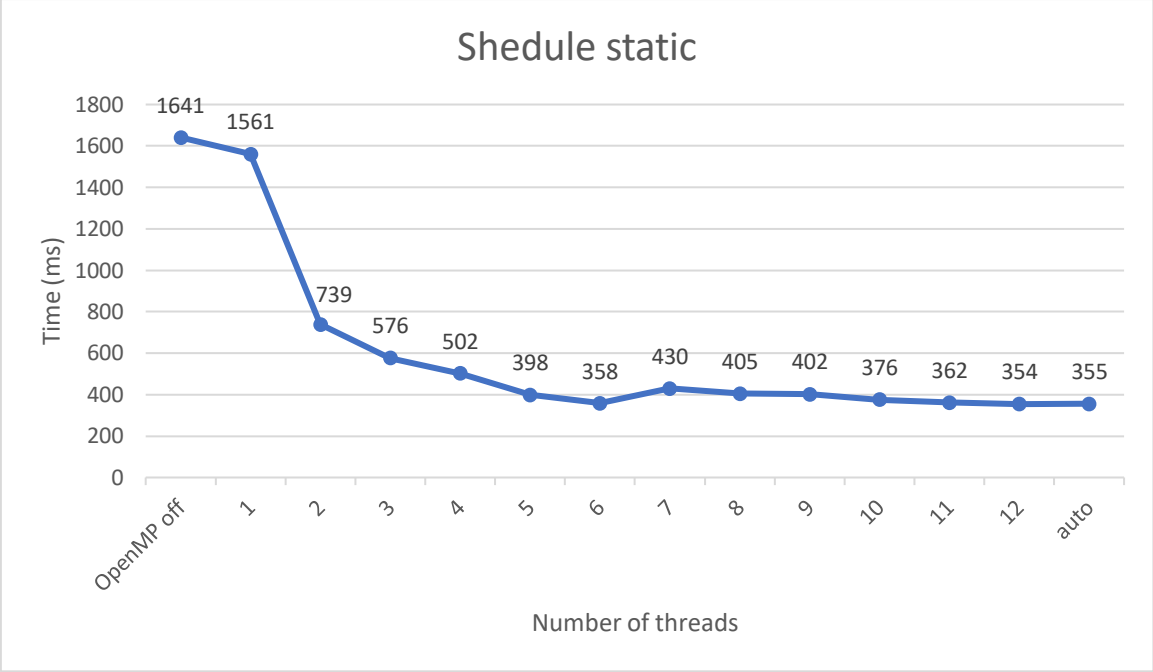

График №1 – Зависимость времени выполнения программы от числа потоков

На графике 2 изображена зависимость времени выполнения программы от метода планирования (schedule type), все тесты проводились для 12 потоков.

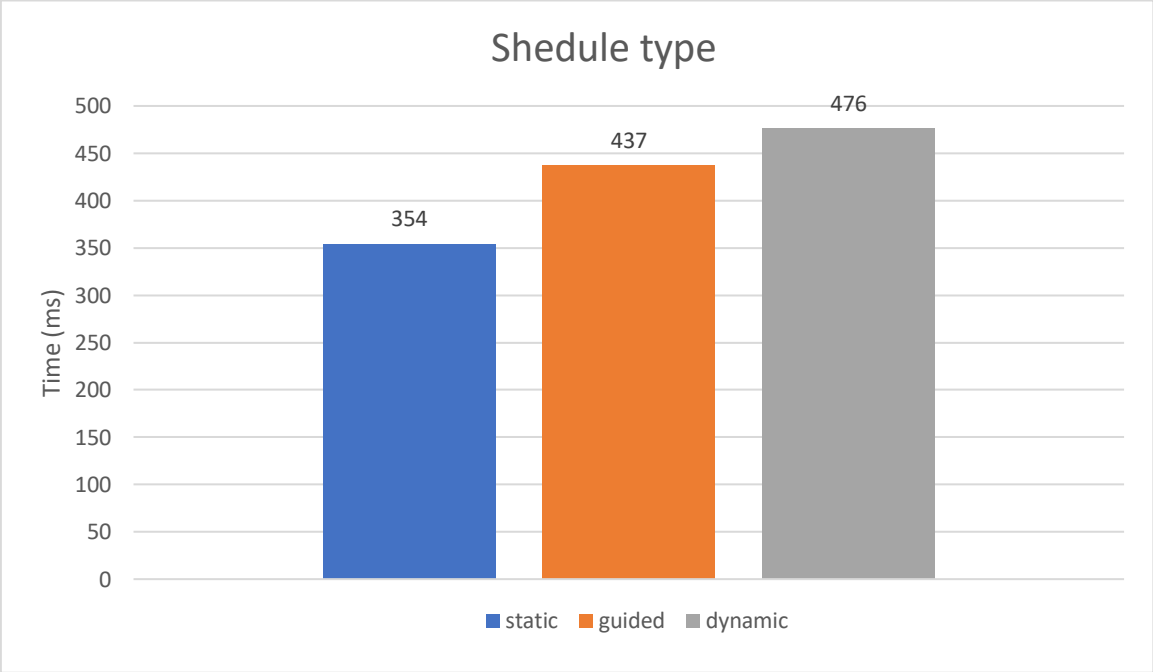

График №2 – Зависимость времени выполнения программы от метода планирования

Листинг

Компилятор Visual C++ 2019.

MatrixDeterminant.cpp

```
#include <iostream>
#include <random>
#include <cmath>
#include <ctime>
#include <omp.h>
#include <fstream>

using namespace std;

int LUPDecompose(double** a, int n, double tol);
int LUPDecomposeParallel(double** a, int n, double tol);
double LUPDeterminant(double** a, int p, int n);
void fillMatrix(double** a, int n, FILE* in);
void fillMatrixRandom(double** a, int n, int abs = 10);
double fRand(double fMin, double fMax);
void printMatrix(double** a, int n);
double** copyMatrix(double** a, int n);
void DeleteMatrix(double** a, int n);
double* testParallel(double** matrix, int n, int threads);
double* test(double** matrix, int n);
double averageTime(double** a, int n, int nthreads, int ntests, bool parallel =
true);

#define SHEDULE dynamic
#define TOL 0.0001

int main(int argc, char* argv[]) {
    if (argc < 3 || argc > 4) {
        printf("Invalid number of arguments");
        return 1;
    }
    char* p;
    int nthreads = strtol(argv[1], &p, 10);
    if (*p || nthreads < -1) {
        printf("Invalid number of threads");
        return 1;
    }

    FILE* in;
    fopen_s(&in, argv[2], "r");
    if (!in) {
        printf("Can't open file: %s", argv[2]);
        return 1;
    }
    if (feof(in)) {
        printf("Unexpected end of file");
        return 1;
    }
    int n = 0;
    fscanf_s(in, "%i", &n);

    if (n <= 0) {
```

```
        printf("Invalid matrix size");
        return 1;
    }
    double** matrix = new double* [n];
    for (int i = 0; i < n; i++) {
        matrix[i] = new double[n];
    }
    fillMatrix(matrix, n, in);
    fclose(in);
    FILE* out;
    if (argc == 4) {
        fopen_s(&out, argv[3], "w");
        if (!out) {
            printf("Can't open file: %s", argv[3]);
            return 1;
        }
    } else {
        out = stdout;
    }
    double* result;
    if (nthreads == -1) {
        result = test(matrix, n);
        fprintf(out, "Determinant: %g\n", result[1]);
        printf("\nTime (without OpenMP): %f ms\n", result[0]);
    } else {
        result = testParallel(matrix, n, nthreads);
        fprintf(out, "Determinant: %g\n", result[1]);
        printf("\nTime (%i thread(s)): %f ms\n", nthreads, result[0]);
    }
    fclose(out);
    DeleteMatrix(matrix, n);
}

double averageTime(double** a, int n, int nthreads, int ntests, bool parallel) {
    double* res = new double[ntests];
    for (int i = 0; i < ntests; i++) {
        res[i] = parallel ? testParallel(a, n, nthreads)[0] : test(a, n)[0];
    }
    double sum = 0;
    for (int i = 0; i < ntests; i++) {
        sum += res[i];
    }
    return sum / (double) ntests;
}

void fillMatrix(double** a, int n, FILE* in) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (feof(in)) {
                printf("Unexpected end of file");
                exit(1);
            }
            fscanf_s(in, "%lg", &a[i][j]);
        }
    }
}

double* testParallel(double** matrix, int n, int nthreads) {
    double** a = copyMatrix(matrix, n);
```

```
    if (nthreads != 0) {
        omp_set_num_threads(nthreads);
    }
    double start_time = omp_get_wtime();
    int p = LUPDecomposeParallel(a, n, TOL);
    double det = LUPDeterminant(a, p, n);
    double end_time = omp_get_wtime();
    DeleteMatrix(a, n);
    return new double[] {(end_time - start_time) * 1000, det};
}

double* test(double** matrix, int n) {
    double** a = copyMatrix(matrix, n);
    double start_time = omp_get_wtime();
    int p = LUPDecompose(a, n, TOL);
    double det = LUPDeterminant(a, p, n);
    double end_time = omp_get_wtime();
    DeleteMatrix(a, n);
    return new double[] {(end_time - start_time) * 1000, det};
}

void DeleteMatrix(double** a, int n) {
    for (int i = 0; i < n; i++) {
        delete[] a[i];
    }
    delete[] a;
}

double** copyMatrix(double** a, int n) {
    double** b = new double* [n];
    for (int i = 0; i < n; i++) {
        b[i] = new double[n];
        for (int j = 0; j < n; j++) {
            b[i][j] = a[i][j];
        }
    }
    return b;
}

void printMatrix(double** a, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%5.3g\t", a[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void fillMatrixRandom(double** a, int n, int abs) {
    srand(time(0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = fRand(-abs, abs);
        }
    }
    a[0][0] = 0;
    if (n > 1) {
        a[1][1] = 0;
```

```
    }
}

double fRand(double fMin, double fMax) {
    double f = (double)rand() / RAND_MAX;
    return fMin + f * (fMax - fMin);
}

int LUPDecompose(double** a, int n, double tol) {
    int i, j, k, imax;
    double maxA, * ptr, absA;
    int p = 0;

    for (i = 0; i < n; i++) {
        maxA = fabs(a[i][i]);
        imax = i;
        if (maxA < tol) {
            for (k = i; k < n; k++) {
                if ((absA = fabs(a[k][i])) > maxA) {
                    maxA = absA;
                    imax = k;
                }
            }

            if (maxA < tol) return -1;

            ptr = a[i];
            a[i] = a[imax];
            a[imax] = ptr;
            p++;
        }
        for (j = i + 1; j < n; j++) {
            a[j][i] /= a[i][i];
            for (k = i + 1; k < n; k++) {
                a[j][k] -= a[j][i] * a[i][k];
            }
        }
    }
    return p;
}

int LUPDecomposeParallel(double** a, int n, double tol) {
    int i, j, k, imax;
    double maxA, * ptr, absA;
    int p = 0;

    for (i = 0; i < n; i++) {
        maxA = fabs(a[i][i]);
        imax = i;
        if (maxA < tol) {
            for (k = i; k < n; k++) {
                if ((absA = fabs(a[k][i])) > maxA) {
                    maxA = absA;
                    imax = k;
                }
            }

            if (maxA < tol) return -1;
```

```
            ptr = a[i];
            a[i] = a[imax];
            a[imax] = ptr;
            p++;
        }
        #pragma omp parallel for schedule(SHEDULE)
        for (j = i + 1; j < n; j++) {
            a[j][i] /= a[i][i];
            for (int p = i + 1; p < n; p++) {
                a[j][p] -= a[j][i] * a[i][p];
            }
        }
    }
    return p;
}

double LUPDeterminant(double** a, int p, int n) {
    if (p == -1) {
        return 0;
    }
    double det = a[0][0];
    for (int i = 1; i < n; i++) {
        det *= a[i][i];
    }
    return p % 2 == 0 ? det : -det;
}
```