МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)

Институт информационных технологий, математики и механики

Направление подготовки: «Прикладная математика и информатика»
Профиль подготовки: «Вычислительные методы и суперкомпьютерные технологии»

Отчет
по лабораторной работе
на тему «Блочное LU-разложение квадратной матрицы»

Выполнил:
студент группы 3823М1ПМвм
Бекетов Е.В.

Проверил:
д.т.н., доц., зав.каф. МОСТ
Баркалов К.А.

Нижний Новгород
2024

Оглавление

1. Введение

Решение систем линейных алгебраических уравнений (СЛАУ) является важной и популярной среди инженеров задачей. С решением СЛАУ связаны многие матричные операции: вычисление определителя, обращение матрицы, нахождение собственных чисел и собственных векторов, и др.

На текущий момент известно 2 подхода, каждый из которых имеет достаточное количество методом:

1. Прямой подход, который ищет точное решение системы, где наиболее известны методы Гаусса, LU-разложение и разложение Холецкого.

2. Итерационный подход, который ищет решение за счет приближения с каждой итерацией, где наиболее известны методы Гаусса-Зейделя и Якоби.

В данной работе будет рассмотрено блочное LU-разложение.

2. Постановка задачи

Условие:
Реализовать блочное LU-разложение для квадратной матрицы, используя технологию OpenMP, то есть представить матрицу A в виде произведения двух матриц: $A = LU$, где L — нижняя треугольная матрица, а U — верхняя треугольная матрица.

Требования:
Программа на языке C++ должна реализовывать функцию со следующим заголовком:
void LU_Decomposition(double * A, double * L, double * U, int n);

Формат входа:
Функция получает в аргументах следующие переменные:
A – указатель на массив, в котором по строкам хранится матрица A размера $n \times n$
n – размерность матрицы

Формат выхода:
L – указатель на массив, в котором по строкам необходимо записать матрицу L размера $n \times n$
U – указатель на массив, в котором по строкам необходимо записать матрицу U размера $n \times n$
Ответ считается корректным, если

$$\frac{||LU - A||_2}{||A||_2} < 0.01$$

Ограничения на размер задачи:
Размерность матрицы $n \leq 3000$

3. Теоретическая часть

3.1. Описание метода

LU-разложение – это представление матрицы A в виде $A = LU$, где L – нижнетреугольная матрица с единичной диагональю, а U – верхнетреугольная матрица. LU-разложение является модификацией метода Гаусса. LU-разложение возможно только когда матрица A обратима, а все главные миноры не вырождены.

Недостатком стандартного алгоритма LU-разложения является не эффективное использование кэш-памяти. Так как при достаточно больших размерах матрицы во время арифметических операций часто приходится обращаться к элементам, не лежащим вблизи в памяти, что как раз так и является не эффективны. В то время как правильное использование кэша может существенно (в десятки раз) повысить быстродействие вычислений.

Самый главный для LU-разложения способ – это укрупнение вычислительных операций, приводящее к последовательной обработке некоторых прямоугольных блоков матрицы A.

LU-разложение можно организовать так, что матричные операции (реализация которых допускает эффективное использование кэш-памяти) станут основными. Для этого представим матрицу A в следующем блочном виде:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

где блоки A_{11} размера $r \times r$, A_{12} размера $r \times (n-r)$, A_{21} размера $(n-r) \times r$, A_{22} размера $(n-r) \times (n-r)$. Аналогично и с L и U:

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} U = \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

где блоки L_{11}, U_{11} размера $r \times r$, L_{22}, U_{11} размера $(n-r) \times (n-r)$, L_{21} размера $r \times (n-r)$, U_{12} размера $(n-r) \times r$. Перемножим:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

Теперь блоки L_{11} и U_{11} можно найти, применив метод Гаусса, а затем, решая треугольные системы с несколькими правыми частями будут найдены L_{21} и L_{12}. Далее находим редуцированную матрицы $\hat{A_{22}}$:

$$\hat{A_{22}} = A_{22} - L_{21}U_{12} = L_{21}U_{12} + L_{22}U_{22} - L_{21}U_{12} = L_{22}U_{22}$$

LU-разложение редуцированной матрицы $\hat{A_{22}}$ совпадает с искомыми блоками L_{22}, U_{22}, и для нее можно применить тот же алгоритм, то есть у нас будет рекурсия.

Так как блочная реализация будет состоять из большого количества циклов, то удобно будет использовать принципы параллелизма конкретно к ним.

3.2. Вычислительная трудоемкость

Приведенная блочная схема требует порядка $\frac{2}{3}n^3$ операций. Оценим долю матричных операций. Пусть размер матрицы кратен размеру блока: $n = rN$, где N – размер блока. Операции, не являющиеся матричными, при разложении на L и U требуют так

же $\frac{2}{3}n^3$ операций и тогда в процессе блочного разложения потребуется решить N систем. В итоге долю матричных операций можно оценить следующим образом:

$$1 - \frac{\frac{2}{3}r^3N}{\frac{2}{3}n^3} = 1 - \frac{r^3N}{n^3} = 1 - \frac{r^3N}{r^3N^3} = 1 - \frac{1}{N^2}$$

3.3. Подтверждение корректности

Для подтверждения корректности разложения, был написан метод, который считает модуль разности LU и A, и если он меньше 0.01 то алгоритм работает корректно, иначе сообщает о его не корректности. Ниже представлена его реализация:

```
#define scalar(row, col) ((col) + (row) * size)
void Check_correct(double* A, double* L, double* U, int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j)
        {
            double sum = 0;
            for (int k = 0; k < size; ++k)
                sum += L[scalar(i, k)] * U[scalar(k, j)];
            if (abs(A[scalar(i, j)] - sum) <= 0.01) {
                continue;
            }
            else {
                std::cout << "no correct" << "\n";
            }
        }
    }
    std::cout << "correct!" << "\n";
```

4. Практическая часть

Для проведения дальнейших численных экспериментов были написаны 3 версии алгоритма: последовательный, последовательный блочный, параллельный блочный. Код каждой приведен в конце отчета в приложениях.

Эксперименты проводились с использованием следующей конфигурации:

Процессор	8 ядерный AMD Ryzen 7 5800HS (2.8 GHz)
Память, кэш	32 GB, L1 – 32 Kb, L2 – 512 Kb, L3 – 16 Mb
Операционная система	Windows 10 x64
Среда разработки	Visual Studio 2022
Библиотеки	OpenMP
Компилятор	Intel oneAPI DPC++/C++ Compiler (2024.0.2)

4.1. Эксперименты

Проведем серию экспериментов для выявления оптимального числа потоков и размера блока для матриц размера 1000, 2000, 3000, 4000.

4.1.1. Последовательный алгоритм

Сделаем замеры «написанного в лоб» алгоритма Рис. 1.

Последовательный алгоритм				
Размер матрицы	1000	2000	3000	4000
Время последовательного алгоритма	0,089	1,749	6,998	16,786

Рис. 1: Время работы последовательного алгоритма в секундах.

Очевидная ситуация, время работы растет пропорционально размеру задачи, посмотрим теперь, что будет при блочной реализации алгоритма Рис. 2

Последовательный алгоритм от размера блока				
Размер блока	Размер матрицы			
	1000	2000	3000	4000
100	0,225	1,997	6,2	15,764
200	0,236	2,129	6,617	17,282
500	0,202	1,966	6,521	15,818
1000	0,085	1,818	6,485	16,779

Рис. 2: Время работы последовательного блочного алгоритма в секундах.

Из таблицы можно сделать однозначный вывод, что размер блока равный 200, самый худший, однако блоки размера 1000 и 100 самые лучшие. Дальнейшие результаты для параллельного алгоритма я проводил с размером блока равным 1000, так как для малой матрицы он дает наибольшую разность с самым худшим размером.

4.1.2. Параллельный алгоритм

Сделаем замеры алгоритма при блоке равным 1000 и числе потоков равному 1 Рис. 3.

Параллельный алгоритм от числа потоков				
Число потоков	Размер матрицы			
	1000	2000	3000	4000
1	0,097	1,772	6,321	16,28
2	0,056	1,019	3,358	8,87
3	0,041	0,726	2,526	6,318
4	0,03	0,58	2,042	5,174

Рис. 3: Время работы параллельного алгоритма в секундах.

Видно, что наименьшее время достигается при 4 потоках, что не удивительно. Посмотрим, на ускорение алгоритма в сравнении с использованием одного потока Рис. 4

Ускорение параллельного алгоритма от числа потоков				
Число потоков	Размер матрицы			
	1000	2000	3000	4000
1	1	1	1	1
2	1,732143	1,73896	1,88237	1,8354
3	2,365854	2,440771	2,502375	2,576765
4	3,233333	3,055172	3,095495	3,146502

Рис. 4: Ускорение параллельного алгоритма.

В целом на Рис. 3 и Рис. 4 видно, что параллельный алгоритм работает исправно, не наблюдается никаких аномалий, по типу свехрлинейного ускорения, а значит алгоритм написан правильно.

5. Заключение

В данной лабораторной работе был изучен алгоритм блочного разложения квадратной матрицы – LU-разложение.

Было показано, что в отличии от стандартного, «написанного в лоб», разложения, блочный параллельный алгоритм показывает результаты по времени лучше, за счет эффективного использования кэш-памяти и принципа разделения вычислений на несколько потоков, получившиеся результаты совпадают с теоретическими предположениями, что говорит о правильности алгоритма.

6. Литература

1. Баркалов К.А. Образовательный комплекс «Параллельные численные методы». – Н.Новгород, Изд-во ННГУ 2011.

2. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989.

3. Белов С.А., Золотых Н.Ю. Численные методы линейной алгебры. – Н.Новгород, Изд-во ННГУ, 2005.

4. Вербицкий В.В., Реут В.В. Введение в численные методы алгебры: учебное пособие / В.В. Вербицкий, В.В. Реут. – Одесса: Одесский национальный университет имени И.И. Мечникова, 2015.

7. Приложение А – Код последовательного алгоритма

```
void LU_Decomposition(double* A, double* L, double* U, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
        {
            U[n * i + j] = A[n * i + j];
        }
    }
    for (int i = 0; i < n; i++) {
        L[i * n + i] = 1;
        for (int k = i + 1; k < n; k++) {
            double mu = U[k * n + i] / U[n * i + i];
            for (int j = i; j < n; j++) {
                U[k * n + j] -= mu * U[i * n + j];
            }
            L[k * n + i] = mu;
            L[i * n + k] = 0;
        }
    }
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            U[i * n + j] = 0;
        }
    }
}
```

8. Приложение Б – Код последовательного блочного алгоритма

```
void LU_Decomposition(double* A, double* L, double* U, int n, int r) {
    if (n <= r) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                U[n * i + j] = A[n * i + j];
            }
        }
        for (int i = 0; i < n; ++i) {
            L[i * n + i] = 1;
            for (int k = i + 1; k < n; ++k) {
                double mu = U[k * n + i] / U[n * i + i];
                for (int j = i; j < n; ++j) {
                    U[k * n + j] -= mu * U[i * n + j];
                }
                L[k * n + i] = mu;
                L[i * n + k] = 0;
            }
        }
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                U[i * n + j] = 0;
            }
        }
    } else {
        int l = n - r;
        double* A11 = new double[r * r];
        double* A12 = new double[r * l];
        double* A21 = new double[r * l];
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < r; ++j) {
                A11[i * r + j] = A[i * n + j];
            }
        }
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < l; ++j) {
                A12[i * l + j] = A[i * n + j + r];
            }
        }
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < r; ++j) {
                A21[i * r + j] = A[(i + r) * n + j];
            }
        }
        double* L11 = new double[r * r];
        double* U11 = new double[r * r];
        LU_Decomposition(A11, L11, U11, r, r);
```

```
delete[] A11;
double* L21 = new double[r * l];
double* U12 = new double[r * l];
for (int iter = 0; iter < l; ++iter) {
    for (int i = 0; i < r; ++i) {
        L21[r * iter + i] = A21[r * iter + i];
        for (int j = 0; j < i; ++j) {
            L21[r * iter + i] -= U11[r * j + i] * L21[r * iter + j];
        }
        L21[r * iter + i] /= U11[r * i + i];
    }
}
for (int iter = 0; iter < l; ++iter) {
    for (int i = 0; i < r; ++i) {
        U12[l * i + iter] = A12[l * i + iter];
        for (int j = 0; j < i; ++j) {
            U12[l * i + iter] -= L11[r * i + j] * U12[l * j + iter];
        }
        U12[l * i + iter] /= L11[r * i + i];
    }
}
delete[] A12;
delete[] A21;
for (int i = 0; i < r; ++i) {
    for (int j = 0; j < r; ++j) {
        L[i * n + j] = L11[i * r + j];
        U[i * n + j] = U11[i * r + j];
    }
}
delete[] L11;
delete[] U11;
double* A22 = new double[l * l];
double* L22 = new double[l * l];
double* U22 = new double[l * l];
for (int i = 0; i < l; ++i) {
    for (int j = 0; j < l; ++j) {
        A22[i * l + j] = A[(i + r) * n + j + r];

        for (int k = 0; k < r; ++k) {
            A22[i * l + j] -= L21[i * r + k] * U12[k * l + j];
        }
    }
}
LU_Decomposition(A22, L22, U22, l, r);
delete[] A22;
for (int i = 0; i < r; ++i) {
    for (int j = 0; j < l; ++j) {
        L[i * n + j + r] = 0;
        U[i * n + j + r] = U12[i * l + j];
    }
```

```
        }
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < r; ++j) {
                L[(i + r) * n + j] = L21[i * r + j];
                U[(i + r) * n + j] = 0;
            }
        }
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < l; ++j) {
                L[(i + r) * n + j + r] = L22[i * l + j];
                U[(i + r) * n + j + r] = U22[i * l + j];
            }
        }
        delete[] L21;
        delete[] L22;
        delete[] U12;
        delete[] U22;
    }
}
```

9. Приложение В – Код параллельного блочного алгоритма

```
void LU_Decomposition(double* A, double* L, double* U, int n, int r) {
    const int num_thread = 16;
    const int r = 1500;
    if (n <= r) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                U[n * i + j] = A[n * i + j];
            }
        }
        for (int i = 0; i < n; ++i) {
            L[i * n + i] = 1;
#pragma omp parallel for num_threads(num_thread)
            for (int k = i + 1; k < n; ++k) {
                double mu = U[k * n + i] / U[n * i + i];
                for (int j = i; j < n; ++j) {
                    U[k * n + j] -= mu * U[i * n + j];
                }
                L[k * n + i] = mu;
                L[i * n + k] = 0;
            }
        }
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                U[i * n + j] = 0;
            }
        }
    } else {
        int l = n - r;
        double* A11 = new double[r * r];
        double* A12 = new double[r * l];
        double* A21 = new double[r * l];
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < r; ++j) {
                A11[i * r + j] = A[i * n + j];
            }
        }
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < l; ++j) {
                A12[i * l + j] = A[i * n + j + r];
            }
        }
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < r; ++j) {
```

```
                A21[i * r + j] = A[(i + r) * n + j];
            }
        }
        double* L11 = new double[r * r];
        double* U11 = new double[r * r];
        LU_Decomposition(A11, L11, U11, r, r, num_thread);
        delete[] A11;
        double* L21 = new double[r * l];
        double* U12 = new double[r * l];
#pragma omp parallel for num_threads(num_thread)
        for (int iter = 0; iter < l; ++iter) {
            for (int i = 0; i < r; ++i) {
                L21[r * iter + i] = A21[r * iter + i];
                for (int j = 0; j < i; ++j) {
                    L21[r * iter + i] -= U11[r * j + i] * L21[r * iter + j];
                }
                L21[r * iter + i] /= U11[r * i + i];
            }
        }
#pragma omp parallel for num_threads(num_thread)
        for (int iter = 0; iter < l; ++iter) {
            for (int i = 0; i < r; ++i) {
                U12[l * i + iter] = A12[l * i + iter];
                for (int j = 0; j < i; ++j) {
                    U12[l * i + iter] -= L11[r * i + j] * U12[l * j + iter];
                }
                U12[l * i + iter] /= L11[r * i + i];
            }
        }
        delete[] A12;
        delete[] A21;
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < r; ++j) {
                L[i * n + j] = L11[i * r + j];
                U[i * n + j] = U11[i * r + j];
            }
        }
        delete[] L11;
        delete[] U11;
        double* A22 = new double[l * l];
        double* L22 = new double[l * l];
        double* U22 = new double[l * l];

#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < l; ++j) {
                A22[i * l + j] = A[(i + r) * n + j + r];

                for (int k = 0; k < r; ++k) {
```

```
                    A22[i * l + j] -= L21[i * r + k] * U12[k * l + j];
                }
            }
        }
        LU_Decomposition(A22, L22, U22, l, r, num_thread);
        delete[] A22;
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < l; ++j) {
                L[i * n + j + r] = 0;
                U[i * n + j + r] = U12[i * l + j];
            }
        }
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < r; ++j) {
                L[(i + r) * n + j] = L21[i * r + j];
                U[(i + r) * n + j] = 0;
            }
        }
#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < l; ++i) {
            for (int j = 0; j < l; ++j) {
                L[(i + r) * n + j + r] = L22[i * l + j];
                U[(i + r) * n + j + r] = U22[i * l + j];
            }
        }
        delete[] L21;
        delete[] L22;
        delete[] U12;
        delete[] U22;
    }
}
```