МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ 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: «Фундаментальная информатика и информационные

технологии»

Магистерская программа: «Инженерия программного обеспечения»

**ОТЧЕТ**

по лабораторной работе

**Выполнил:** студент группы382006-1м

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.А. Солуянов

**Проверил:**

к.ф.-м. н., доц., доцент каф. МОСТ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_К.А. Баркалов

Нижний Новгород

2021

Оглавление

[Введение 3](#_Toc72267624)

[Постановка задачи 4](#_Toc72267625)

[Описание метода 4](#_Toc72267626)

[Задача лабораторной работы 4](#_Toc72267627)

[Алгоритм решения 5](#_Toc72267628)

[Трудоемкость 5](#_Toc72267629)

[Параллельный алгоритм 6](#_Toc72267630)

[Реализация 6](#_Toc72267631)

[Эксперименты 10](#_Toc72267632)

[Тестовая инфраструктура 10](#_Toc72267633)

[Результаты 10](#_Toc72267634)

# Введение

Решение систем линейных алгебраических уравнений (СЛАУ) является достаточно важной вычислительной задачей (примерно 75% всех расчетных математических задач приходится на их решение). С решением СЛАУ связаны такие задачи, как вычисление определителей, обращение матриц, вычисление собственных значений и собственных векторов матриц, интерполирование, аппроксимация по методу наименьших квадратов, решение систем дифференциальных уравнений и многие другие. Современная вычислительная математика располагает большим арсеналом методов решения СЛАУ, а математическое обеспечение ЭВМ – многими пакетами программ и программными системами, позволяющими решать СЛАУ.

Методы решения систем линейных алгебраических уравнений классифицируют на прямые (точные) и итерационные. Прямые методы основаны на выполнении конечного числа арифметических операций (метод обратной матрицы, метод Гаусса, метод разложения Холецкого и т.д.). В основе итерационных методов лежит последовательность приближений, позволяющая получить решение системы, определяемое необходимой точностью. Общими словами, такие методы устанавливают процедуру уточнения определённого начального приближения к решению. При выполнении условий сходимости они позволяют достичь любой точности путем повторения итераций. Преимущество этих методов в том, что часто они позволяют достичь решения с заранее заданной точностью быстрее прямых методов, а также позволяют решать большие системы уравнений. К итерационным методам относятся: метод сопряженных градиентов, метод бисопряженных градиентов, методы Зейделя и Якоби и т.д.

В данной работе рассматривается метод бисопряженных градиентов решения СЛАУ (итерационный метод) c разреженной матрицей A и плотными векторами x и b.

# Постановка задачи

## Описание метода

Метод бисопряженных градиентов является обобщением метода сопряженных градиентов на случае линейной системы с произвольной квадратной невырожденной матрицей . Хорошо известно, что матрица – симметрична и положительно определена, а система эквивалентная исходной. Для решения данной системы мы не можем применить метод сопряженных градиентов, так как обусловленность матрицы много хуже обусловленности матрицы , а именно .

С другой стороны, доказано, что с помощью рекуррентного соотношения небольшого порядка добиться попарной ортогонализации невязок в случае произвольной матрицы невозможно. В методе бисопряженных градиентов реализована другая идея. Последовательность ортогональных невязок заменяется на две биортогональные последовательности и , а последовательность сопряженных направлений заменена на две бисопряженные и .

Системы векторов и называются биортогональными, если при .

## Задача лабораторной работы

В данной лабораторной работе требуется реализовать метод бисопряженных градиентов решения СЛАУ с разреженной матрицей и плотными векторами . Для реализации параллельных вычислений необходимо использовать технологию OpenMP.

Необходимо произвести замеры времени выполнения и сравнить эффективность параллельных вычислений с последовательной реализацией.

## Алгоритм решения

1. Подготовка перед итерационным процессом:
   1. Задается начальное приближение
   2. – вектор невязки
2. На каждой -ой итерации метода вычисляются:

|  |  |  |
| --- | --- | --- |
| a. | (1) | |
| b. | (2) |
| c. | (3) |
| d. | (4) |
| e. | (5) |

1. Критерий остановки

|  |  |
| --- | --- |
| или | (6) |

1. Если критерий не выполняется, то вычисляются , и начинается следующая итерация:

|  |  |
| --- | --- |
| f. | (7) |
| g. | (8) |

## Трудоемкость

Подсчитаем количество операций, которое требуется для данного метода.

1. На каждой итерации:

* Две операции умножения матрицы на вектор () - операций.
* Четыре операции скалярного произведения - операций.
* Десять операций над векторами (сложение, разность векторов, умножение константы на вектор, взятие нормы) - операций

1. Общая трудоемкость одной итерации:

|  |  |
| --- | --- |
|  | (9) |

1. Для выполнения K итераций метода необходимо:

|  |  |
| --- | --- |
|  | (10) |

## Параллельный алгоритм

Выполнение итераций метода осуществляется последовательно, так как каждая следующая итерация уточняет решение, полученное на предыдущей итерации.

Анализ последовательного алгоритма позволяет сделать следующие выводы:

1. Основные вычисления метода приходятся на умножение и . При использовании алгоритмов параллельного умножения матрицы на вектор, можно повысить эффективность выполнения итерации исходного алгоритма.
2. Дополнительные вычисления по обработке векторов (скалярное произведение, сложение и вычитание, умножение на скаляр, взятие нормы) так же могут быть выполнены параллельно в силу наличия независимых шагов.

Поэтому эффективность последовательного алгоритма может быть повышена за счет применения параллельных вычислений в ходе выполнения отдельных итераций.

# Реализация

В данной задаче используются разреженные матрицы. Для описания матрицы необходимо знать:

* n – число строк в матрице
* m – число столбцов в матрице
* nz – число ненулевых элементов
* Позиции элементов.

Существует несколько способов представления таких матриц в памяти компьютера. В данной лабораторной работе будет использован разреженный строчный формат (CRS – Compressed Row Storage). Данный способ хранения (Листинг 1) требует разделения матрицы на 3 массива:

1. Массив значений value, который содержит ненулевые значения матрицы, записанные построчно сверху вниз.
2. Массив номеров столбцов colIndex.
3. Массив индексов начала строк rowPtr.

Способ требует объем памяти равный

|  |  |
| --- | --- |
|  | (11) |

Листинг 1. Структура хранения разреженной матрицы

**struct** CRSMatrix

{

 int n; // Число строк в матрице

 int m; // Число столбцов в матрице

 int nz; // Число ненулевых элементов в разреженной матрице

 vector val; // Массив значений матрицы по строкам

 vector colIndex; // Массив номеров столбцов

 vector rowPtr; // Массив индексов начала строк

};

Внутри одной итерации необходимо несколько раз просчитывать операции с матрицами и векторами, поэтому имеет смысл вынести их в отдельные функции (Листинг 2).

Листинг 2. Дополнительные операции

//Умножение матрицы на вектора

**void** Multiplication(CRSMatrix& A, **double**\* v, **double**\* res) {

#**pragma omp parallel for**

**for** (**int** i = 0; i < A.n; i++) {

res[i] = 0.0;

**for** (**int** j = A.rowPtr[i]; j < A.rowPtr[i + 1]; j++) {

res[i] += A.val[j] \* v[A.colIndex[j]];

}

}

}

//Вычисление скалярного произведения векторов

**double** Scalar(**double**\* a, **double**\* b, **int** & size) {

**double** res = 0;

#**pragma omp parallel for reduction(+: res)**

**for** (**int** i = 0; i < size; i++) {

res += a[i] \* b[i];

}

**return** res;

}

//Вычисление коэффициента альфа

**void** GetAlpha(**double**\* r, **double**\* r\_, CRSMatrix& A, **double**\* p, **double**\* p\_, **double**\* tmp, **double** &alpha) {

Multiplication(A, p, tmp);

alpha = Scalar(r, r\_, A.n) / Scalar(tmp, p\_, A.n);

}

//Вычисление коэффициента бетта

**void** GetBeta(**double**\* r, **double**\* r\_, **double**\* r\_prev, **double**\* r\_prev\_, **double**& beta, **int** size) {

beta = Scalar(r, r\_, size) / Scalar(r\_prev, r\_prev\_, size);

}

//Сложение векторов с некоторым коэффициентом

**void** Addition(**double**\* l, **double**\* r, **double**\* res, **double** coef, **int** size) {

#**pragma omp parallel for**

**for** (**int** i = 0; i < size; i++) {

res[i] = l[i] + coef \* r[i];

}

}

//Вычисление нормы вектора

**double** Norma(**double**\* a, **int** size) {

**double** sum = 0.0;

#**pragma omp parallel for reduction(+: sum)**

**for** (**int** i = 0; i < size; i++) {

sum += a[i] \* a[i];

}

**return** **sqrt**(sum);

}

В функции SLE\_Solver\_CRS\_BICG реализовано основное тело алгоритма с тремя критериями остановки: по максимальному числу итераций и двум критериям точности (Листинг 3).

Листинг 3. Функция решения СЛАУ методом бисопряженных градиентов

**void** SLE\_Solver\_CRS\_BICG(CRSMatrix& A, **double**\* b, **double** eps, **int** max\_iter, **double**\* x, **int**& **count**) {

**int** n = A.n;

**double**\* r = **new** **double**[n];

**double**\* r\_prev = **new** **double**[n];

**double**\* r\_ = **new** **double**[n];

**double**\* r\_prev\_ = **new** **double**[n];

**double**\* p = **new** **double**[n];

**double**\* p\_ = **new** **double**[n];

**double**\* tmp = **new** **double**[n];

**double** alpha, beta;

GenerateStartSolution(n, x);

GetR0(A, x, b, r);

CopyArray(r, r\_, n);

CopyArray(r, p, n);

CopyArray(r, p\_, n);

CRSMatrix AT = Transponse(A);

**while** (**true**) {

**count**++;

GetAlpha(r, r\_, A, p, p\_, tmp, alpha);

Addition(x, p, x, alpha, n);

Multiplication(A, p, tmp);

CopyArray(r, r\_prev, n);

Addition(r, tmp, r, -alpha, A.n);

Multiplication(AT, p\_, tmp);

CopyArray(r\_, r\_prev\_, n);

Addition(r\_, tmp, r\_, -alpha, n);

GetBeta(r, r\_, r\_prev, r\_prev\_, beta, n);

**if** (**abs**(beta) < 1e-10) {

**break**;

}

**if** (Norma(r, n) < eps) {

**break**;

}

**if** (**count** >= max\_iter) {

**break**;

}

Addition(r, p, p, beta, n);

Addition(r\_, p\_, p\_, beta, n);

}

**delete**[] r;

**delete**[] r\_prev;

**delete**[] r\_;

**delete**[] r\_prev\_;

**delete**[] p;

**delete**[] p\_;

**delete**[] tmp;

}

# Эксперименты

## Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (Таблица 1).

|  |  |
| --- | --- |
| Процессор | Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz |
| Память | 8,00 ГБ |
| Операционная система | Windows 10 |
| Среда разработки | Visual Studio 2019 |
| Компилятор | Intel(R) oneAPI DPC++/C++ Compiler 2021.1 |
| Библиотеки | OpenMP |

Таблица 1. Тестовая инфраструктура

## Результаты

Заключение

Литература

Приложение