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 «Метод бисопряженных градиентов для решения СЛАУ с разреженной матрицей»

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, Якоби, сопряженных градиентов и бисопряженных градинетов.

В данной работе будет рассмотрено метод бисопряженных градиентов.

2. Постановка задачи

Условие:
Реализовать метод бисопряженных градиентов для решения СЛАУ с разреженной матрицей, используя технологию OpenMP:
$Ax = b$, где A – разреженная квадратная положительно определённая матрица, x, b – плотные векторы.

Требования:
Программа на языке C++ должна реализовывать функцию со следующим заголовком: **SLE_Solver_CRS_BICG(CRSMatrix & A, double * b, double eps, int max_iter, double * x, int & count);**
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
};

Формат входа:
Функция получает в аргументах следующие переменные:
A – указатель на структуру CRSMatrix, в которой хранится матрица A в CRS формате размера $n \times n$
b – указатель на массив, в котором по строкам хранится столбец b размера $n \times 1$
eps – критерий остановки: $||x_k - x_{k+1}||_2 < eps$
max_iter – критерий остановки: число итераций больше max_iter
$count$ – число выполненных итераций алгоритмом

Формат выхода:
x – указатель на массив, в который по строкам необходимо записать столбец x размера $n \times 1$
Ответ считается корректным, если

$$\frac{||Ax - b||_2}{||A||_2} < 10^{-7}$$

Ограничения на размер задачи:
Размерность матрицы $n \leq 10000$, число ненулевых элементов $nz \leq 10^7$.

3. Теоретическая часть

3.1. Описание метода

Метод бисопряженных градиентов является обобщением метода сопряженных градиентов в случае линейной системы $Ax = b$ с произвольной квадратной невырожденной матрицей A. Известно, что матрица A^TA – симметрична и положительно определена, а система $A^TAx = A^Tb$ эквивалентная исходной. Для решения данной системы нельзя применять метод сопряженных градиентов, так как обусловленность A^TA много хуже обусловленности матрицы A, то есть $cond(A^TA) \approx cond(A)^2$.

Однако известно, что с помощью рекуррентного соотношения малого порядка невозможно добиться попарной ортогонализации невязок r_k для произвольной матрицы A. Но в методе реализована другая идея: последовательность ортогональных невязок r_k заменяется на две биортогональные последовательности r_k и $\widetilde{r_k}$, а последовательность сопряженных направлений заменена на две бисопряженные – p_k и $\widetilde{p_k}$.

Вектора $\{v_1, v_2, ..., v_m\}$ и $\{w_1, w_2, ..., w_m\}$ называются биортогональными, если выполняется следующее выражение $(v_i, w_j) = 0$, $i \neq j$.

Алгоритм будет выглядеть следующим образом:

1. Определение основных векторов:
   (a) Задаем вектор начального приближения x_0
   (b) Считаем начальную невязку $r_0 = Ax_0 - b$
   (c) Присваиваем $p_0 = r_0$
2. Запускаем основной цикл по k:
   (a) Вычисляем $\alpha_k = \frac{(r_k, \widetilde{r_k})}{(Ap_k, \widetilde{p_k})}$
   (b) Находим следующее приближение $x_{k+1} = x_k + \alpha_k p_k$
   (c) Считаем невязку $r_{k+1} = r_k - \alpha_k Ap_k$
   (d) Считаем невязку $\widetilde{r}_{k+1} = \widetilde{r_k} - \alpha_k A^T\widetilde{p_k}$
   (e) Вычисляем $\beta_k = \frac{(r_{k+1}, \widetilde{r}_{k+1})}{(r_k, \widetilde{r_k})}$
   (f) Если $||r_{k+1}|| < \varepsilon$ или $\beta_k = 0$, то завершаем цикл, иначе продолжаем
   (g) Вычисляем $p_{k+1} = r_{k+1} + \beta_k p_k$
   (h) Вычисляем $\widetilde{p}_{k+1} = \widetilde{r}_{k+1} + \beta_k\widetilde{p_k}$

3.2. Вычислительная трудоемкость

Подсчитаем общее число операций, которое потребуется для метода. На каждой итерации необходимо:
2 операции умножения матрицы на вектор – $n^2 + n^2 = 2n^2$,
4 операции скалярного произведения – $4n$,
5 операций умножения матрицы на скаляр – $5n$,
6 операций сложения матриц/векторов – $6n$ K – число итераций метода.

В конечном итоге: $T = K(2n^2 + 15n)$

4. Практическая часть

Для проведения дальнейших численных экспериментов были написаны 2 версии алгоритма: последовательный и параллельный. 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 для матриц размера 5000, 10000, 20000, 30000. Программно создается обыкновенная двумерная матрица и заполняется рандомными значениями, если значение попадает в отрезок $[0, 0.95]$ то на его месте ставится 0, иначе само значение. В итоге получается, что количество ненулевых элементов для каждой матрицы составляет порядка 5%. Точность $\varepsilon = 0.00001$, число итераций 10000.

4.2. Параллельный алгоритм

Сделаем замеры алгоритма при разном числе потоков Рис. 1.

Параллельный алгоритм от числа потоков				
Число потоков	Размер матрицы			
	5000	10000	20000	30000
1	2,519	89,953	375,16	973,118
2	1,412	53,893	220,865	537,269
3	0,93	36,192	216,798	435,527
4	0,771	30,21	173,216	539,283

Рис. 1: Время работы параллельного алгоритма в секундах.

Видно что увеличение числа потоков уменьшает время работы, однако в самой крупной задаче (30000), это наблюдение нарушается и для нее оптимальным будет 3 потока, в остальных 4.

Взглянем для наглядности на таблицу ускорения на Рис. 2.

Ускорение параллельного алгоритма от числа потоков				
Число потоков	Размер матрицы			
	5000	10000	20000	30000
1	1	1	1	1
2	1,783994	1,669104	1,698594	1,811231
3	2,708602	2,485439	1,730459	2,234346
4	3,267185	2,97759	2,165851	1,804466

Рис. 2: Ускорение параллельного алгоритма.

На Рис. 1 и Рис. 2 видно, что параллельный алгоритм работает исправно, не наблюдается никаких аномалий, за исключением спада производительности для большой задачи, что не удивительно по причине слишком большого числа ненулевых элементов массива ≈ 45 млн. Так же в подтверждении моих слов видно, что с ростом размерности падает прирост на 4 потоках (от 3.27 до 1.8). Сверхлинейное расширение не наблюдается. Алгоритм работает правильно.

5. Заключение

В конечном итоге в данной лабораторной работе с помощью языка C++ и технологии OpenMP был реализован метод бисопряженных градиентов для решения СЛАУ с разреженной матрицей A и плотными векторами x и b.

Был изучен теоретический материал задачи, анализ трудоемкости вычислений. Был написан последовательный алгоритм, и в дальнейшем модифицирован до параллельного.

В результате произведенных экспериментов было выявлено, что при использовании параллельных вычислений, есть прирост производительности, однако с увеличением размера задачи он падает, что вполне логично. Однако в остальных случая получается выигрыш по времени работы алгоритма, что говорит об эффективности распараллеливания.

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. Писсанецки С. Технология разрежённых матриц. - М.: Мир, 1988.

5. Джордж А., Лю Дж. Численное решение больших разрежённых систем уравнений. – М.: Мир, 1984.

7. Приложение А – Код перевода матрицы в CRS формат

```
    void convertToCRS(const std::vector<std::vector<double>>& A,
    CRSMatrix& crsA) {
    crsA.rowPtr.push_back(0);

    for (int i = 0; i < A.size(); ++i) {
        for (int j = 0; j < A[i].size(); ++j) {
            if (A[i][j] != 0.0) {
                crsA.val.push_back(A[i][j]);
                crsA.colIndex.push_back(j);
            }
        }
        crsA.rowPtr.push_back(crsA.val.size());
    }
}
```

8. Приложение Б – Код последовательного алгоритма

```
double scalar_product(double* a, double* b, int n, int num_thread) {
    double result = 0;
    for (int i = 0; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}


void matrix_multiplication(CRSMatrix& A, double* b, double* mul,
int num_thread) {
    for (int i = 0; i < A.n; ++i) {
        mul[i] = 0.0;
        for (int j = A.rowPtr[i]; j < A.rowPtr[i + 1]; ++j) {
            mul[i] += A.val[j] * b[A.colIndex[j]];
        }
    }
}


void SLE_Solver_CRS_BICG(CRSMatrix& A, double* b, double eps,
int max_iter, double* x, int & count, int num_thread) {
    int n_size = A.n;
    CRSMatrix AT;

    AT.n = A.m;
    AT.m = A.n;

    std::vector<std::vector<int>> index(AT.n);
    std::vector<std::vector<double>> value(AT.n);
    std::vector<int> size(AT.n);

    for (int i = 0; i < A.n; ++i) {
        for (int j = A.rowPtr[i]; j < A.rowPtr[i + 1]; ++j) {
            index[A.colIndex[j]].push_back(i);
            value[A.colIndex[j]].push_back(A.val[j]);
            size[A.colIndex[j]] += 1;
        }
    }

    AT.rowPtr.push_back(0);
    for (int i = 0; i < AT.n; ++i) {
        AT.rowPtr.push_back(AT.rowPtr[i] + size[i]);
        for (int j = 0; j < size[i]; ++j) {
            AT.val.push_back(value[i][j]);
            AT.colIndex.push_back(index[i][j]);
        }
    }
```

```
for (int i = 0; i < n_size; ++i) {
    x[i] = 1;
}

double* Ap = new double[n_size];
double* r = new double[n_size];
double* r_ = new double[n_size];

double* p = new double[n_size];
double* p_ = new double[n_size];

matrix_multiplication(A, x, Ap, num_thread);

for (int i = 0; i < n_size; ++i) {
    r[i] = b[i] - Ap[i];
    r_[i] = r[i];
    p[i] = r[i];
    p_[i] = r_[i];
}

double* Ap_ = new double[n_size];

double* r_next = new double[n_size];
double* r_next_ = new double[n_size];

double alpha = 1;
double betta = 1;

double norm = sqrt(scalar_product(b, b, n_size, num_thread));

while (true) {
    count++;
    if (sqrt(scalar_product(r, r, n_size, num_thread)) / norm < eps) {
        std::cout << "Scalar" << "\n";
        break;
    }
    if (count >= max_iter) {
        std::cout << "Iter" << "\n";
        break;
    }
    //if (abs(betta) < 1e-6) {
    //    std::cout << "Betta" << "\n";
    //    break;
    //}

    matrix_multiplication(A, p, Ap, num_thread);
    matrix_multiplication(AT, p_, Ap_, num_thread);
    double scalar_tmp = scalar_product(r, r_, n_size, num_thread);
    alpha = scalar_tmp / scalar_product(Ap, p_, n_size, num_thread);
```

```
            for (int i = 0; i < n_size; ++i) {
                x[i] += alpha * p[i];
                r_next[i] = r[i] - alpha * Ap[i];
                r_next_[i] = r_[i] - alpha * Ap_[i];
            }

            betta = scalar_product(r_next, r_next_, n_size,
            num_thread) / scalar_tmp;

            for (int i = 0; i < n_size; ++i) {
                p[i] = r_next[i] + betta * p[i];
                p_[i] = r_next_[i] + betta * p_[i];
                r[i] = r_next[i];
                r_[i] = r_next_[i];
            }
        }
        delete[] Ap;
        delete[] Ap_;

        delete[] r;
        delete[] r_;

        delete[] r_next;
        delete[] r_next_;

        delete[] p;
        delete[] p_;

    }
```

9. Приложение B – Код параллельного алгоритма

```
double scalar_product(double* a, double* b, int n, int num_thread) {
    double result = 0;
#pragma omp parallel for num_threads(num_thread) reduction(+: result)
    for (int i = 0; i < n; ++i) {
        result += a[i] * b[i];
    }
    return result;
}


void matrix_multiplication(CRSMatrix& A, double* b, double* mul,
int num_thread) {
#pragma omp parallel for num_threads(num_thread)
    for (int i = 0; i < A.n; ++i) {
        mul[i] = 0.0;
        for (int j = A.rowPtr[i]; j < A.rowPtr[i + 1]; ++j) {
            mul[i] += A.val[j] * b[A.colIndex[j]];
        }
    }
}


void SLE_Solver_CRS_BICG(CRSMatrix& A, double* b, double eps,
int max_iter, double* x, int & count, int num_thread) {
    int n_size = A.n;
    CRSMatrix AT;

    AT.n = A.m;
    AT.m = A.n;

    std::vector<std::vector<int>> index(AT.n);
    std::vector<std::vector<double>> value(AT.n);
    std::vector<int> size(AT.n);

    for (int i = 0; i < A.n; ++i) {
        for (int j = A.rowPtr[i]; j < A.rowPtr[i + 1]; ++j) {
            index[A.colIndex[j]].push_back(i);
            value[A.colIndex[j]].push_back(A.val[j]);
            size[A.colIndex[j]] += 1;
        }
    }

    AT.rowPtr.push_back(0);
    for (int i = 0; i < AT.n; ++i) {
        AT.rowPtr.push_back(AT.rowPtr[i] + size[i]);
        for (int j = 0; j < size[i]; ++j) {
            AT.val.push_back(value[i][j]);
            AT.colIndex.push_back(index[i][j]);
```

```
        }
    }

#pragma omp parallel for num_threads(num_thread)
    for (int i = 0; i < n_size; ++i) {
        x[i] = 1;
    }

    double* Ap = new double[n_size];
    double* r = new double[n_size];
    double* r_ = new double[n_size];

    double* p = new double[n_size];
    double* p_ = new double[n_size];

    matrix_multiplication(A, x, Ap, num_thread);

#pragma omp parallel for num_threads(num_thread)
    for (int i = 0; i < n_size; ++i) {
        r[i] = b[i] - Ap[i];
        r_[i] = r[i];
        p[i] = r[i];
        p_[i] = r_[i];
    }

    double* Ap_ = new double[n_size];

    double* r_next = new double[n_size];
    double* r_next_ = new double[n_size];

    double alpha = 1;
    double betta = 1;

    double norm = sqrt(scalar_product(b, b, n_size, num_thread));

    while (true) {
        count++;
        if (sqrt(scalar_product(r, r, n_size, num_thread)) / norm < eps) {
            std::cout << "Scalar" << "\n";
            break;
        }
        if (count >= max_iter) {
            std::cout << "Iter" << "\n";
            break;
        }
        //if (abs(betta) < 1e-6) {
        //    std::cout << "Betta" << "\n";
        //    break;
        //}
```

```
        matrix_multiplication(A, p, Ap, num_thread);
        matrix_multiplication(AT, p_, Ap_, num_thread);
        double scalar_tmp = scalar_product(r, r_, n_size, num_thread);
        alpha = scalar_tmp / scalar_product(Ap, p_, n_size, num_thread);

#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < n_size; ++i) {
            x[i] += alpha * p[i];
            r_next[i] = r[i] - alpha * Ap[i];
            r_next_[i] = r_[i] - alpha * Ap_[i];
        }

        betta = scalar_product(r_next, r_next_, n_size,
        num_thread) / scalar_tmp;

#pragma omp parallel for num_threads(num_thread)
        for (int i = 0; i < n_size; ++i) {
            p[i] = r_next[i] + betta * p[i];
            p_[i] = r_next_[i] + betta * p_[i];
            r[i] = r_next[i];
            r_[i] = r_next_[i];
        }
    }
    delete[] Ap;
    delete[] Ap_;

    delete[] r;
    delete[] r_;

    delete[] r_next;
    delete[] r_next_;

    delete[] p;
    delete[] p_;

}
```