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**

**Фундаментальная информатика и информационные технологии**

**Параллельное программирование**

**ОТЧЕТ**

по лабораторной работе

на тему:

**«Решение СЛАУ методом сопряженных градиентов   
с использованием OpenMP и TBB»**

**Выполнил:**

студент группы 381706-3

Балдин Алексей Александрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

**Преподаватель:**

старший преподаватель

кафедры МОСТ ИТММ

Козинов Евгений Александрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород  
2020

**Содержание**

[Введение и постановка задачи. 3](file:///C:\Users\kret9\Desktop\Отчет%20по%20ВМ.docx#_Toc26382497)

[Теоритическая часть 4](file:///C:\Users\kret9\Desktop\Отчет%20по%20ВМ.docx#_Toc26382498)

[Метод решения и схема распараллеливания 6](file:///C:\Users\kret9\Desktop\Отчет%20по%20ВМ.docx#_Toc26382499)

[Описание программной реализации 7](file:///C:\Users\kret9\Desktop\Отчет%20по%20ВМ.docx#_Toc26382500)

[Тестирование и результаты экспериментов 11](file:///C:\Users\kret9\Desktop\Отчет%20по%20ВМ.docx#_Toc26382501)

[Заключение 13](file:///C:\\Users\\kret9\\Desktop\\Отчет%20по%20ВМ.docx" \l "_Toc26382502)

**Введение и постановка задачи.**

Метод сопряжённых градиентов — итерационный метод для решения систем линейных алгебраических уравнений, выдающий ответ, сходящийся к реальному за конечное число шагов. Он является одним из наиболее эффективных методов решения СЛАУ с положительно определённой матрицей, с другими матрицами данный метод не работает. Основная проблема заключается в том, что из-за накопления погрешностей может сходимость может ухудшаться и метод не даст правильного ответа.

Постановка задачи: требуется написать программу, которая будет использовать метод сопряженных градиентов для решения СЛАУ, работающую в трех режимах: последовательном, параллельном с использованием OpenMP, параллельном с использованием TBB. Так же необходимо провести проверку корректности и временные эксперименты.

**Теоретическая часть**

Рассмотрим систему линейных уравнений , и задачу оптимизации *,* где A – симметричная, положительно определённая матрица .

Заметим, что и условие экстремума функции

эквивалентно системе . Функция Fдостигает своей нижней грани в единственной точке x_*, определяемой уравнением . Таким образом, данная задача оптимизации сводится к решению системы линейных уравнений .

Идея метода сопряжённых градиентов состоит в следующем:   
Пусть - базис в . Тогда для любой точки вектор

раскладывается по базису . Таким образом, представимо в виде: . Каждое следующее приближение вычисляется по формуле: .

**Определение.** Два вектора p и q называются сопряженными относительно симметричной матрицы B, если. Опишем способ построения базиса в методе сопряжённых градиентов. В качестве начального приближения выбираем произвольный вектор. На каждой итерации выбираются по правилу: ).

Базисные вектора вычисляются по формулам:

Коэффициенты выбираются так, чтобы векторы и были сопряжёнными относительно А.

Если обозначить за , то после нескольких упрощений получим окончательные формулы, используемые при применении метода сопряжённых градиентов на практике:

**Метод решения и схема распараллеливания**

Для решения данной задачи напишем программу на языке С++ с использованием OpenMP и TBB, в которой реализуем функционал по решению СЛАУ методом сопряженных градиентов, использующий вышеуказанные формулы.

Шаги работы алгоритма:

1. Вычисление векторов r1 и p1.
2. Следующие шаги выполняются в цикле, количество итераций равно сумме значения размерности матрицы и количества дополнительных итераций, число которых вычисляется на основе размера СЛАУ:
   1. Вычисление коэффициента alfa.
   2. Вычисление очередного приближенного решения.
   3. Вычисление очередного вектора r.
   4. Вычисление коэффициента beta.
   5. Вычисление очередного вектора p.
3. После проделанных шагов начинаем искать невязку очередного решения, и, если она меньше заданной погрешности, заканчиваем процесс решения, иначе, повторяем шаги a-e предыдущего пункта.

**Схема распараллеливания:**

Средства распараллеливания применим на 2 и 3 шаге, применив технологии OpenMP и TBB для вычисления произведений матрицы на вектор (распределение работы между потоками происходит по строкам) и скалярного произведения векторов, и для произведения матрицы на вектор в процессе проверки корректности решения. В остальных шагах использовать распараллеливание не имеет смысла, так как мы будем затрачивать больше времени на работу с потоками, чем выигрывать от ускорения вычислений.

**Описание программной реализации**

Для решения поставленной задачи мною были написаны следующие классы и функции:

* randomGenerator – класс, нужный для генерации случайных чисел для заполнения матриц и векторов
* mathVector – класс, хранящий в себе математические вектора и необходимые для решения СЛАУ операции работы с ними
* matrix – класс, хранящий в себе математические матрицы и необходимые для решения СЛАУ операции работы с ними
* Перегруженные математические функции, такие как сложение, вычитание и умножение, необходимые для упрощения работы с матрицами и векторами
* Функции gradientMethod и CheckCorrectSolution для всех вариантов реализации программы, отвечающие за решение СЛАУ методом сопряженных градиентов и проверки корректности полученного решения
* Функции, представляющие собой умножения матриц и векторов с применением технологии OpenMP
* Функторы, необходимые для умножения матриц и векторов с использованием технологии TBB

**Использование OpenMP:**

Основой распараллеливания с использованием OpenMP является директива #pragma omp parallel for.

Умножение векторов:

double VmultVOMP(const mathVector& vec1, const mathVector& vec2, const int& size, const int& numThreads)

{

double result = 0;

int i = 0;

#pragma omp parallel for private(i) reduction(+:result) num\_threads(numThreads)

for (i = 0; i < size; ++i)

{

result += vec1.data[i] \* vec2.data[i];

}

return result;

}

Распараллеливание происходит таким образом, что разные потоки производят умножение соответствующих им элементов вектора, а затем складывают результат в переменную result, с использованием коллективной операции “+”. На каждом потоке создается локальная копия result, в которую он производит сложение, а затем весь результат складывается.

Умножение матрицы на вектор происходит несколько иначе:

mathVector MmultVOMP(const matrix& matr, const mathVector& vec, const int& size, const int& numThreads)

{

mathVector result(size);

int i = 0;

int j = 0;

#pragma omp parallel for private(j) num\_threads(numThreads)

for (i = 0; i < size; ++i)

{

for (j = 0; j < size; ++j)

{

result.data[i] += matr.data[size \* i + j] \* vec.data[j];

}

}

return result;

}

Каждый поток работает с определенными строками матрицы, не мешая друг другу и складывая результат в итоговый вектор result.

**Использование TBB:**

Основой работы с TBB в данной лабораторной является распараллеливание циклов с известным числом повторений с использованием шаблонной функции tbb::parallel\_for.

Умножение матрицы на вектор:

parallel\_for(blocked\_range<int>(0, size), MmultVTBB(&matr, &vec, &result, size));

Первый аргумент представляет собой одномерное итерационное пространство, задающее количество итераций распараллеливаемого цикла, второй представляет собой функтор, в котором, через перегрузку оператора () осуществлены вычисления.

Функтор MmultVTBB и перегрузка ():

class MmultVTBB

{

private:

matrix \*matr\_;

mathVector \*vec\_;

mathVector \*res\_;

int size\_;

public:

MmultVTBB(matrix \*matr, mathVector \*vec, mathVector \*res, const int& size) : matr\_(matr), vec\_(vec), res\_(res), size\_(size) {}

void operator()(const blocked\_range<int>& r) const;

};

void MmultVTBB::operator()(const blocked\_range<int>& r) const

{

int begin = r.begin(), end = r.end();

for (int i = begin; i != end; i++)

{

res\_->data[i] = 0;

for (int j = 0; j < size\_; j++)

{

res\_->data[i] += matr\_->data[size\_ \* i + j] \* vec\_->data[j];

}

}

}

С помощью одномерного итерационного пространства каждый поток получает свои индексы начала и конца матрицы и вектора, чтобы свободно работать, не мешая друг другу, а затем складывать результат в вектор res\_.

Умножение векторов происходит с использованием редукции:

class VmultVTBB

{

private:

mathVector \*vec1\_, \*vec2\_;

double res\_;

int size\_;

public:

explicit VmultVTBB(mathVector \*vec1, mathVector \*vec2, const int& size) : vec1\_(vec1), vec2\_(vec2), size\_(size), res\_(0) {}

VmultVTBB(const VmultVTBB& vv, split) : vec1\_(vv.vec1\_), vec2\_(vv.vec2\_), size\_(vv.size\_), res\_(0) {}

void reset(mathVector \*vec1, mathVector \*vec2);

void operator()(const blocked\_range<int>& r);

void join(const VmultVTBB& vv);

double result();

};

void VmultVTBB::operator()(const blocked\_range<int>& r)

{

int begin = r.begin(), end = r.end();

for (int i = begin; i != end; i++)

res\_ += vec1\_->data[i] \* vec2\_->data[i];

}

void VmultVTBB::join(const VmultVTBB & vv)

{

res\_ += vv.res\_;

}

Каждый поток считает суммирует соответствующие ему части векторов, а затем, с помощью функции функтора join складывает локальные результаты вычислений в общий результат.

На практике, обычное умножение векторов не используется, так как удобнее в ходе метода сопряженных градиентов единственное место, где необходимо скалярное произведение(СК) – формула, где одно СК делится на другое, и поэтому две последовательных операции СК заменены на одну общую, где распараллеливаются оба СК и затем делятся одно на другое. Распараллеливание происходит таким же образом, как и в случае умножения векторов.

**Тестирование и результаты экспериментов**

Для тестирования написанных программ я использую библиотеку gtest, обладающую широкий функционал для проведения тестов и отображения их результатов.

Тесты проводились на компьютере с 16гб оперативной памяти и процессором Intel Core i5 – 3570K x64(4 ядра, 4 потока). Допустимой погрешностью было указано – 0.001.

Общий список написанных тестов:

* Тесты на корректность метода сопряженных градиентов с размерами матриц 10х10, 100х100, 500х500, 1000х1000
* Тесты на корректность метода сопряженных градиентов с применением OpenMP с размерами матриц 10х10, 100х100, 500х500, 1000х1000
* Тесты на корректность метода сопряженных градиентов с применением TBB с размерами матриц 10х10, 100х100, 500х500, 1000х1000
* Тесты на время исполнения метода сопряженных градиентов с размерами матриц 10х10, 100х100, 500х500, 1000х1000
* Тесты на время исполнения метода сопряженных градиентов с применением OpenMP с размерами матриц 10х10, 100х100, 500х500,1000х1000
* Тесты на время исполнения метода сопряженных градиентов с применением TBB с размерами матриц 10х10, 100х100, 500х500,1000х1000

**Результаты тестирования:**

Тесты на корректность показали, что все методы работают правильно в рамках заданной погрешности.

Для получения точного среднего времени выполнения программ тестирование проводилось многократно, а затем результаты были усреднены.

Тесты на время исполнения показали следующие результаты:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Размерность матрицы** | **Время решения последовательным методом СГ** | **Время решения методом СГ с OpenMP на 2 потоках** | **Время решения методом СГ с OpenMP на 4 потоках** | **Время решения методом СГ с TBB на 2 потоках** | **Время решения методом СГ с TBB на 4 потоках** |
| 10х10 | 0.0001 сек. | 0.0001 сек. | 0.0002 сек. | 0.0002 сек. | 0.0001 сек. |
| 100х100 | 0.0052 сек. | 0.004 сек. | 0.0031 сек. | 0.0056 сек. | 0.0051 сек. |
| 500х500 | 0.6246 сек. | 0.3228 сек. | 0.1753 сек. | 0.3565 сек. | 0.193 сек. |
| 1000х1000 | 8.1857 сек. | 4.6157 сек. | 2.4604 сек. | 4.2229 сек. | 2.1622 сек. |

**Заключение**

Программа, решающая СЛАУ методом сопряженных градиентов, была реализована в трех вариантах, тесты проведены, эффективность параллельной схемы была подтверждена, однако, было установлено, что при очень маленьких размерах матрицы распараллеливать по потокам не имеет смысла из-за времени, затрачиваемого на работу с ними, а также то, что OpenMP версия немного обгоняет TBB по времени при средних размерах матрицы, таких как 100х100, однако при больших, например, 1000х1000, TBB показывает себя немного лучше, но данная разница во времени существенна лишь при огромном объеме вычислений.