

**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.

Фундаментальная информатика и информационные технологии.

Отчет по лабораторной работе на тему:

**«Умножение плотных матриц. Элементы типа double.Алгоритм Штрассена с помощью OpenMP и TBB»**

**Выполнил:** студент группы *381706-4*

*Андреев Сергей Александрович*

*подпись\_\_\_\_\_*

**Проверил:**  доцент кафедры МОСТ ИТММ

*Сысоев Александр Владимирович*

*подпись\_\_\_\_\_*

Оглавление

* Введение
* Постановка задачи
* Теоретическая часть
* Методы решения
* Последовательный метод
* OpenMP
* TBB
* Программная реализация
* Необходимые функции
* Тестирование программы
* Заключение
* Литература

# Введение

Одной из самых распространенных задач в математике является перемножение матриц. Самый известный способ-это, конечно же, перемножение строчки на столбец, данный метод известен, наверное, каждому студенту. Но что, если размеры нашей матрицы больше, чем трехзначные числа. Справится с таким перемножением сложно не только человеку, но даже компьютеру. Поэтому математики придумывают разные другие методы, чтобы сократить время выполнения данного перемножения. Одним из таких математиков является Фолькер Штрассен, который придумал свой метод перемножения. В отличии от традиционного перемножения, которое работает за время, его алгоритм выполняет умножает за время, что дает выигрыш на больших плотных матрицах, начиная с размера 64х64.

# Постановка задачи

Необходимо разработать программу, которая будет выполнять умножение плотных матриц с элементами типа double, с помощью алгоритма Штрассена. Работа должна быть выполнена в версиях: последовательная, с помощью OpenMP и TBB, а также сравнить скорость их выполнения на разном количестве потоков.

# Теоретическая часть

Пусть нам даны две квадратные матрицы А и В с кольцом R и матрица С получается по формуле:

Разделим матрицы А, В и С на равные по размеру блочные матрицы:

Тогда:

Но выполняя данную процедуру нам не удалось уменьшить количество умножений, поэтому мы вводим новые переменные:

Которые затем мы используем для С, таким образом, нам требуется всего 7 перемножений, вместо обычных 8. С через Р мы вычисляем по следующим формулам:

Дальше продолжаем рекурсивный процесс до тех пор, пока размер матрицы не станет достаточно мал для использования обычного метода умножения. Обычно этот малы размер в пределах от 32 до 128.

# Методы решения

## Последовательный метод

Последовательный метод алгоритма Штрассена

* Проверяем размер матрицы, если меньше критического значения, то используем обычный метод
* Разделяем наши матрицы на 4 части
* Выполняем рекурсивное умножение с помощью вспомогательных переменных
* Находим результирующие блоки С
* Формируем результирующую матрицу

double\* Str\_alg(double\* matrix1, double\* matrix2, int N, int threshold)

{

double\* Rez;

if (N <= threshold)

Rez = defaultMult(matrix1, matrix2, N);

else

{

Rez = CreateMatrix(N);

N = N / 2;

double\* A[4]; double\* B[4]; double\* C[4]; double\* P[7];

double\* TMP1; double\* TMP2; double\* TMP3; double\* TMP4; double\* TMP5;

double\* TMP6; double\* TMP7; double\* TMP8; double\* TMP9; double\* TMP10;

for (int i = 0; i < 4; i++)

{

A[i] = CreateMatrix(N);

B[i] = CreateMatrix(N);

}

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

{

A[0][i \* N + j] = matrix1[2 \* i \* N + j];

A[1][i \* N + j] = matrix1[2 \* i \* N + j + N];

A[2][i \* N + j] = matrix1[2 \* i \* N + j + 2 \* N \* N];

A[3][i \* N + j] = matrix1[2 \* i \* N + j + 2 \* N \* N + N];

B[0][i \* N + j] = matrix2[2 \* i \* N + j];

B[1][i \* N + j] = matrix2[2 \* i \* N + j + N];

B[2][i \* N + j] = matrix2[2 \* i \* N + j + 2 \* N \* N];

B[3][i \* N + j] = matrix2[2 \* i \* N + j + 2 \* N \* N + N];

}

TMP1 = Add(A[0], A[3], N);

TMP2 = Add(B[0], B[3], N);

TMP3 = Add(A[2], A[3], N);

TMP4 = Sub(B[1], B[3], N);

TMP5 = Sub(B[2], B[0], N);

TMP6 = Add(A[0], A[1], N);

TMP7 = Sub(A[2], A[0], N);

TMP8 = Add(B[0], B[1], N);

TMP9 = Sub(A[1], A[3], N);

TMP10 = Add(B[2], B[3], N);

P[0] = Str\_alg(TMP1, TMP2, N, threshold);

P[1] = Str\_alg(TMP3, B[0], N, threshold);

P[2] = Str\_alg(A[0], TMP4, N, threshold);

P[3] = Str\_alg(A[3], TMP5, N, threshold);

P[4] = Str\_alg(TMP6, B[3], N, threshold);

P[5] = Str\_alg(TMP7, TMP8, N, threshold);

P[6] = Str\_alg(TMP9, TMP10, N, threshold);

C[0] = Sub(P[0], P[3], P[6], P[4], N);

C[1] = Add(P[2], P[4], N);

C[2] = Add(P[1], P[3], N);

C[3] = Sub(P[0], P[2], P[5], P[1], N);

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++) {

Rez[i \* 2 \* N + j] = C[0][i \* N + j];

Rez[i \* 2 \* N + j + N] = C[1][i \* N + j];

Rez[i \* 2 \* N + j + 2 \* N \* N] = C[2][i \* N + j];

Rez[i \* 2 \* N + j + 2 \* N \* N + N] = C[3][i \* N + j];

}

for (int i = 0; i < 4; i++) {

delete[] A[i];

delete[] B[i];

delete[] C[i];

}

for (int i = 0; i < 7; i++) {

delete[] P[i];

}

delete[]TMP1; delete[]TMP2; delete[]TMP3; delete[]TMP4; delete[]TMP5;

delete[]TMP6; delete[]TMP7; delete[]TMP8; delete[]TMP9; delete[]TMP10;

}

return Rez;

}

## OpenMP

Отличия от последовательной версии:

Распараллеливаем циклы с помощью pragma omp for

#pragma omp for private(i,j) schedule(static)

for (i = 0; i < N; i++) {

for (j = 0; j < N; j++) {

С помощью omp section передаем каждому потоку задачу на выполнение

#pragma omp sections

{

#pragma omp section

{

TMP1 = Add(A[0], A[3], N); // (A11 + A22)

}

#pragma omp section

{

TMP2 = Add(B[0], B[3], N); // (B11 + B22)

}

#pragma omp section

{

TMP3 = Add(A[2], A[3], N); // (A21 + A22)

}

#pragma omp section

{

TMP4 = Sub(B[1], B[3], N); // (B12 - B22)

}

#pragma omp section

{

TMP5 = Sub(B[2], B[0], N); // (B21 - B11)

}

#pragma omp section

{

TMP6 = Add(A[0], A[1], N); // (A11 + A12)

}

#pragma omp section

{

TMP7 = Sub(A[2], A[0], N); // (A21 - A11)

}

#pragma omp section

{

TMP8 = Add(B[0], B[1], N); // (B11 + B12)

}

#pragma omp section

{

TMP9 = Sub(A[1], A[3], N); // (A12 - A22)

}

#pragma omp section

{

TMP10 = Add(B[2], B[3], N); // (B21 + B22)

}

}

#pragma omp sections

{

#pragma omp section

{

P[0] = Str\_alg(TMP1, TMP2, N, threshold); //(A11+A22)\*(B11+B22)

}

#pragma omp section

{

P[1] = Str\_alg(TMP3, B[0], N, threshold); // (A21 + A22)\*B11

}

#pragma omp section

{

P[2] = Str\_alg(A[0], TMP4, N, threshold); // A11\*(B12 - B22)

}

#pragma omp section

{

P[3] = Str\_alg(A[3], TMP5, N, threshold); // A22\*(B21 - B11)

}

#pragma omp section

{

P[4] = Str\_alg(TMP6, B[3], N, threshold); // (A11 + A12)\*B22

}

#pragma omp section

{

P[5] = Str\_alg(TMP7, TMP8, N, threshold); //(A21-A11)\*(B11+B12)

}

#pragma omp section

{

P[6] = Str\_alg(TMP9, TMP10, N, threshold); //(A12-A22)\*(B21+B22)

}

}

#pragma omp sections

{

#pragma omp section

{

C[0] = Sub(P[0], P[3], P[6], P[4], N); // P1 + P4 - P5 + P7

}

#pragma omp section

{

C[1] = Add(P[2], P[4], N); // P3 + P5

}

#pragma omp section

{

C[2] = Add(P[1], P[3], N); // P2 + P4

}

#pragma omp section

{

C[3] = Sub(P[0], P[2], P[5], P[1], N); // P1 – P2 + P3 + P6

}

}

# TBB

Отличия от последовательной версии:

Распараллеливаем цикл при помощи подключения библиотек tbb и parallel for

parallel\_for(blocked\_range<size\_t>(0, N),

[=](const blocked\_range<size\_t>& r)

{

for (size\_t i = r.begin(); i != r.end(); ++i)

{

for (size\_t j = 0; j < N; ++j)

{

С помощью task\_group потоки выполняют параллельные перемножения

task\_group TMP;

TMP.run([&] { TMP1 = Add(A[0], A[3], N); });

TMP.run([&] { TMP2 = Add(B[0], B[3], N); });

TMP.run([&] { TMP3 = Add(A[2], A[3], N); });

TMP.run([&] { TMP4 = Sub(B[1], B[3], N); });

TMP.run([&] { TMP5 = Sub(B[2], B[0], N); });

TMP.run([&] { TMP6 = Add(A[0], A[1], N); });

TMP.run([&] { TMP7 = Sub(A[2], A[0], N); });

TMP.run([&] { TMP8 = Add(B[0], B[1], N); });

TMP.run([&] { TMP9 = Sub(A[1], A[3], N); });

TMP.run([&] { TMP10 = Add(B[2], B[3], N); });

TMP.wait();

task\_group p;

p.run([&] { P[0] = Str\_alg(TMP1, TMP2, N, threshold); });

p.run([&] { P[1] = Str\_alg(TMP3, B[0], N, threshold); });

p.run([&] { P[2] = Str\_alg(A[0], TMP4, N, threshold); });

p.run([&] { P[3] = Str\_alg(A[3], TMP5, N, threshold); });

p.run([&] { P[4] = Str\_alg(TMP6, B[3], N, threshold); });

p.run([&] { P[5] = Str\_alg(TMP7, TMP8, N, threshold); });

p.run([&] { P[6] = Str\_alg(TMP9, TMP10, N, threshold); });

p.wait();

task\_group c;

c.run([&] { C[0] = Sub(P[0], P[3], P[6], P[4], N); });

c.run([&] { C[1] = Add(P[2], P[4], N); });

c.run([&] { C[2] = Add(P[1], P[3], N); });

c.run([&] { C[3] = Sub(P[0], P[2], P[5], P[1], N); });

c.wait();

# Программная реализация

## Необходимые функции

CreateMatrix-создаем матрицу

GenerateRandomMatrix-заполняем числами типа double

defaultMult-обычный метод перемножения

Add-сумма 2х матриц

Add-сумма 4х матриц

Sub-вычитание двух матриц

Sub-сумма 3х и вычитание 3ей матрицы

Str\_alg-алгоритм Штрассена последовательный

StrassenParallel\_omp-алгоритм Штрассена с помощью OpenMP

StrassenParallel\_tbb-алгоритм Штрассена с помощью TBB

## Тестирование программы

Тестирование проходит на компьютере с характеристиками: 8Гб оперативной памяти, процессор: AMD RYZEN 5 3550H

Матрица размером 1024 х 1024, критическое значение:64(когда применяется обычный алгоритм). Время измеряется в секундах

Данные получились следующие:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Последовательная версия** | **OpenMP версия** | **TBB версия** |
| **2 потока** | 3.52594 | 1,9665 | 1,91883 |
| **4 потока** | 3.54059 | 1,03484 | 1,06466 |
| **6 потоков** | 3,63052 | 1,03107 | 1,01911 |
| **8 потоков** | 3,5016 | 0,602947 | 0,63656 |

Также для наглядности представлена диаграмма:

# Заключение

Таким образом, нам удалось написать программу для выполнения алгоритма Штрассена на версия TBB и OpenMP. Различия в скоростях получились значительные по сравнению с последовательной версией, но небольшие по сравнению друг с другом, это связано с тем, что использовался примерно одинаковый метод подхода к распараллеливанию программы. task\_group и omp selections похожи между собой. Оба метода параллельного решения справились с задачей и показали ускорение работы программы примерно в 5.8 раза на нашем конкретном примере.

# Литература

Учебный курс «Разработка параллельных программ» А.А. Сиднев А.В.Сысоев

И.Б.Мееров

«Умножение больших целых чисел и алгоритм умножения матриц Штрассена» А.В.Левитин