Министерство образования и науки Российской Федерации

Государственное образовательное учреждение высшего профессионального образования «Нижегородский государственный университет   
им. Н.И. Лобачевского»

#### **Институт информационных технологий, математики и механики**

#### **Кафедра: Программная инженерия**

Специальность (направление): Программная инженерия

**Отчет**

по лабораторной работе

по дисциплине «Параллельное программирование»

тема:

**«Умножение разреженных матриц. Элементы типа double. Формат хранения матрицы – столбцовый (CCS)»**

**Выполнил:**

студент группы 381508

Назаров А.А.

\_\_\_\_\_\_\_Подпись

Нижний Новгород  
2018

**Постановка задачи**

Необходимо реализовать алгоритм умножения CCS (Compressed Column Storage) разреженных матриц в трех версиях: последовательная, параллельная с использованием технологии OpenMP и параллельная с использованием технологии TBB.

Структура хранение CCS матрицы представлена следующим образом:

Массив A содержащий значения в ячейках матрицы, массив IA содержащий номер строки, и массив JA содержащий индекс, с которого начинается столбец в массивах IA и A. В случае, если столбец полностью заполнен нулями, то следующее звено массива JA равно предыдущему. Размер массива JA равен N + 1, где N размер матрицы. Последняя ячейка массива JA хранит размер массива A и массива IA. Массив IA должен быть отсортирован в порядке убывания по примеру указанному ниже. Пример:

Матрица 4x4:

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 0 | 0 | 1 |
| 6 | 0 | 8 | 0 |
| 5 | 0 | 0 | 4 |
| 3 | 0 | 9 | 7 |

Структура хранения:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 2 | 6 | 5 | 3 | 8 | 9 | 1 | 4 | 7 |
| IA | 0 | 1 | 2 | 3 | 1 | 3 | 0 | 2 | 3 |
| JA | 0 | 4 | 4 | 6 | 9 |

1. **Разработка тестовой версии программы**

В данной части будет приведено описание последовательной реализации алгоритма умножения CCS матриц, а также программ для автоматической генерации тестов и проверки правильности проведенной сортировки.

1. **Последовательный алгоритм умножения CCS матриц**

Идея реализации:

Пусть на вход функции умножения поступают три матрицы в формате CCS. Матрица A умножается на матрицу B и в результате получаем матрицу C. Матрица A состоит из массивов A, IA, JA, матрица B из массивов B, IB, JB, матрица C из массивов C, IC, JC. Так как наша матрица является разреженной, то стандартные алгоритмы умножения матриц не подходят. Главная идея умножения CCS матриц заключается в том, что мы, вместо того, что бы высчитывать каждый элемент матрицы C по формуле , мы берем ненулевые элементы из матрицы A, высчитываем то, с какими ненулевыми элементами из матрицы B она может быть умножена, и высчитываем то, с каким элементом из матрицы C надо сложить результат. Благодаря такому подходу, наш алгоритм зависит не от количества всех элементов матрицы C, а от количества не нулевых элементов в каждой из двух умножаемых матриц. Это очень важная особенность алгоритма умножения разреженных матриц, так как наша матрица может иметь колоссальные размеры, и при этом малое количество ненулевых элементов, поэтому зависимость скорости работы от количества всех элементов матрицы недопустима. Если посмотреть на формулу, то видно, что каждый элемент матрицы может быть умножен только на элементы , и результат умножения складывается с элементом матрицы . Реализация алгоритма:

Функция взятия значения элемента (i,j) матрицы:

double CCSMatrGet(double \* A, int \* IA, int \* JA, int N, int i, int j) {

double Value = 0;

int N1 = JA[j];

int N2 = JA[j + 1];

for (int k = N1; k < N2; k++) {

if (IA[k] == i) {

Value = A[k];

break;

}

}

return Value;

}

Функция установки значения элемента (i,j) матрицы:

void CCSMatrPut(double\* &A, int\* &IA, int\* &JA, int N, int i, int j, double val) {

bool zero\_flag = true;

int N1 = JA[j];

int N2 = JA[j + 1];

int k;

for (k = N1; k < N2; k++) {

if (IA[k] == i) {

A[k] = val;

zero\_flag = false;

break;

}

}

if (zero\_flag) {

double \* B = new double[JA[N] + 1];

int \* IB = new int[JA[N] + 1];

int \* JB = new int[N + 1];

int f;

for (f = 0; f < j + 1; f++)

JB[f] = JA[f];

for (f = j + 1; f < N + 1; f++)

JB[f] = JA[f] + 1;

for (f = 0; f < JB[j]; f++) {

B[f] = A[f];

IB[f] = IA[f];

}

for (f = JA[j]; f < JA[j + 1]; f++) {

if (IA[f] < i) {

B[f] = A[f];

IB[f] = IA[f];

}

else {

B[f] = val;

IB[f] = i;

break;

}

}

if (f == JA[j + 1]) {

B[f] = val;

IB[f] = i;

}

else {

f++;

for (; f < JA[j + 1] + 1; f++) {

B[f] = A[f - 1];

IB[f] = IA[f - 1];

}

}

for (f = JA[j + 1]; f < JA[N]; f++) {

B[f + 1] = A[f];

IB[f + 1] = IA[f];

}

delete[] A;

delete[] IA;

delete[] JA;

A = B;

IA = IB;

JA = JB;

}

}

Функция поиска индекса в массиве по значению. Необходима для перебора всей строки матрицы B. Возвращает -1 если такого значения в массиве нет. Start – индекс, с которого нужно начинать поиск.

int ArrIndexFind(int \*arr, int size, int start, int num) {

for (int i = start; i < size; i++)

if (arr[i] == num) return i;

return -1;

}

Функция поиска номера столбца по индексу в массиве IA. Позволяет узнать, к какому столбцу принадлежит элемент массива IA. Реализация:

int ColSearch(int \* JA, int i) {

int f;

for (f = 0; !(JA[f + 1] > i); f++);

return f;

}

Функция умножения разряженных CCS матриц:

void CCSMatrMult(double \* A, int \* IA, int \* JA, double \* B, int \* IB, int \* JB, double \* &C, int \* &IC, int \* &JC, int N) {

if (CCSMatrMultCheck(IB, JB, N)) {

if (C != nullptr)

delete[] C;

if (IC != nullptr)

delete[] IC;

if (JC != nullptr)

delete[] JC;

JC = new int[N + 1];

JC[N] = 0;

for (int i = 0; i < N; i++) JC[i] = 0;

for (int i = 0; i < JA[N]; i++) {

int buf, start = 0;

while ((buf = ArrIndexFind(IB, JB[N], start, ColSearch(JA,i))) != -1) {

CCSMatrPut(C, IC, JC, N, IA[i], ColSearch(JB, buf), CCSMatrGet(C, IC, JC, N, IA[i], ColSearch(JB, buf)) + A[i] \* B[buf]);

start = buf + 1;

}

}

}

else

throw "matrix can not be multiple";

}

1. **Генератор тестовых данных.**

Генератор тестов реализован исходя из особенностей структуры хранения CCS матрицы. Сначала генерируется массив JA, потом массив IA, а уже потом числа в массиве A.  
 Конфигурация тестов задается в двумерном массиве n\_tests, где первое значение – размер матрицы, второе – процент заполненности. Если не указать в качестве аргумента номер теста, генератор автоматически сгенерирует все тесты, представленные в массиве. Реализация:

#include <iostream>

#include <cstdio>

#include <random>

#include <ctime>

#include <chrono>

#define TESTS\_NUM 17

using namespace std;

// 1 число - размер матрицы, 2 число - процент "заполненности" матрицы

int n\_tests[][2] = { { 6, 1},{ 10, 1 },{ 100, 1 },{ 1000, 1 },{ 6, 5 },{ 10, 5 },{ 100, 5 },{ 1000, 5 },

{ 6, 25 },{ 10, 25 },{ 100, 25 },{ 1000,25 } ,{ 6, 50 },{ 10, 50 },{ 100, 50 },{ 1000, 50 }, {40000, 1} };

void test\_creator(int i) {

if (i < 0 || i > 20)

throw "wrong test number";

char buff[3];

freopen(itoa(i, buff, 10), "wb", stdout);

default\_random\_engine generator(chrono::system\_clock::now().time\_since\_epoch().count());

uniform\_real\_distribution <double> val\_distribution(-10, 10);

int n = n\_tests[i][0];

int proc = n\_tests[i][1];

double \*A, \*B;

int \*IA, \*IB, \*JA, \*JB;

JA = new int[n + 1];

JB = new int[n + 1];

int num\_of\_val = static\_cast<int>(static\_cast<double>(static\_cast<double>(n\*n) / 100.0)\*n\_tests[i][1]);

if (num\_of\_val == 0)

num\_of\_val = 1;

IA = new int[num\_of\_val];

IB = new int[num\_of\_val];

A = new double[num\_of\_val];

B = new double[num\_of\_val];

int A\_begin = 0, B\_begin = 0;

int A\_end = ((num\_of\_val / n) == 0) ? 1 : num\_of\_val / n;

int B\_end = A\_end;

JA[0] = 0; JA[n] = num\_of\_val;

JB[0] = 0; JB[n] = num\_of\_val;

for (int f = 1; f < n; f++) {

uniform\_int\_distribution <int> JA\_distribution(A\_begin, (A\_end \* f > num\_of\_val) ? num\_of\_val : A\_end\*f);

uniform\_int\_distribution <int> JB\_distribution(B\_begin, (B\_end \* f > num\_of\_val) ? num\_of\_val : B\_end\*f);

A\_begin = JA\_distribution(generator);

B\_begin = JB\_distribution(generator);

JA[f] = A\_begin;

JB[f] = B\_begin;

}

for (int f = n; f > 0; f--) {

if (JA[f] - JA[f - 1] > n)

JA[f - 1] = JA[f] - n;

if (JB[f] - JB[f - 1] > n)

JB[f - 1] = JB[f] - n;

}

for (int f = 0; f < n; f++) {

int A\_size = JA[f + 1] - JA[f];

int B\_size = JB[f + 1] - JB[f];

for (int g = JA[f], begin = 0; g < JA[f + 1]; g++, begin++) {

uniform\_int\_distribution <int> IA\_distribution(begin, n - (JA[f + 1] - g));

begin = IA\_distribution(generator);

IA[g] = begin;

}

for (int g = JB[f], begin = 0; g < JB[f + 1]; g++, begin++) {

uniform\_int\_distribution <int> IB\_distribution(begin, n - (JB[f + 1] - g));

begin = IB\_distribution(generator);

IB[g] = begin;

}

}

for (int f = 0; f < num\_of\_val; f++) {

A[f] = val\_distribution(generator);

B[f] = val\_distribution(generator);

}

fwrite(&n, sizeof(n), 1, stdout);

fwrite(JA, sizeof(\*JA), n + 1, stdout);

fwrite(JB, sizeof(\*JB), n + 1, stdout);

fwrite(IA, sizeof(\*IA), num\_of\_val, stdout);

fwrite(IB, sizeof(\*IB), num\_of\_val, stdout);

fwrite(A, sizeof(\*A), num\_of\_val, stdout);

fwrite(B, sizeof(\*B), num\_of\_val, stdout);

delete[] JA;

delete[] JB;

delete[] IA;

delete[] IB;

delete[] A;

delete[] B;

}

int main(int argc, char\* argv[]) {

if (argc == 1) {

for (int i = 0; i < TESTS\_NUM; i++) {

test\_creator(i);

}

}

else {

test\_creator(atoi(argv[1]));

}

return 0;

}

1. **Проверка корректности**

Сначала проверяется размер матрицы на корректность, потом все три массива матрицы-ответа сравниваются с массивами матрицы-результата.

Запуск: checker <файл исходного теста> <файл результат> <файл ответ>

Реализация:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <cstdio>

#include <string>

using namespace std;

enum verdict {NO = 1, AC, WA, CE, ML, TL, RE, IL, PE, DE };

class result {

private:

FILE \* bur;

public:

enum ext\_cls { NO = 1, VERDICT, MESSAGE, TIME, MEMORY };

result(bool read = false) {

if (read) bur = fopen("result.txt", "r"); else bur = fopen("result.txt", "w");

}

~result() { fclose(bur); }

void write\_type(ext\_cls t) { fwrite(&t, sizeof(t), 1, bur); }

void write\_verdict(verdict v) {

write\_type(ext\_cls::VERDICT);

fwrite(&v, sizeof(v), 1, bur);

}

void write\_message(string str) {

cout << str << endl;

write\_type(ext\_cls::MESSAGE);

int l = str.size();

fwrite(&l, sizeof(l), 1, bur);

fwrite(&str[0], sizeof(str[0]), 1, bur);

}

void write\_time(long long x) {

cout << "time: " << x << endl;

write\_type(ext\_cls::TIME);

fwrite(&x, sizeof(x), 1, bur);

}

void write\_memory(unsigned long long x) {

write\_type(ext\_cls::MEMORY);

fwrite(&x, sizeof(x), 1, bur);

}

} checker\_result;

int main(int argc,char\* argv[]) {

if (argc != 4) {

std::cout << "use '" << argv[0] << "[source file] [result file] [answer file]'" << std::endl;

return 1;

}

FILE \* file\_source = fopen(argv[1], "rb");

FILE \* file\_res = fopen(argv[2], "rb");

FILE \* file\_ans = fopen(argv[3], "rb");

int N\_source, N\_ans, N\_res;

double res\_time, ans\_time;

fread(&N\_source, sizeof(N\_source), 1, file\_source);

fread(&res\_time, sizeof(res\_time), 1, file\_res);

fread(&ans\_time, sizeof(ans\_time), 1, file\_ans);

fread(&N\_ans, sizeof(N\_ans), 1, file\_ans);

fread(&N\_res, sizeof(N\_res), 1, file\_res);

if (N\_source != N\_ans) {

checker\_result.write\_message("PE. size of source matrix is not equal size of answer matrix");

checker\_result.write\_verdict(verdict::PE);

return 0;

}

if (N\_source != N\_res) {

checker\_result.write\_message("PE. size of source matrix is not equal size of result matrix");

checker\_result.write\_verdict(verdict::PE);

return 0;

}

int \* J\_ans = new int[N\_source + 1];

int \* J\_res = new int[N\_source + 1];

fread(J\_ans, sizeof(\*J\_ans), N\_source + 1, file\_ans);

fread(J\_res, sizeof(\*J\_res), N\_source + 1, file\_res);

int \*I\_ans = new int[J\_ans[N\_source]];

int \*I\_res = new int[J\_res[N\_source]];

double \*ans = new double[J\_ans[N\_source]];

double \*res = new double[J\_res[N\_source]];

fread(I\_ans, sizeof(\*I\_ans), J\_ans[N\_source], file\_ans);

fread(ans, sizeof(\*ans), J\_ans[N\_source], file\_ans);

fread(I\_res, sizeof(\*I\_res), J\_res[N\_source], file\_res);

fread(res, sizeof(\*res), J\_res[N\_source], file\_res);

for (int i = 0; i < N\_source + 1; i++) {

if (J\_ans[i] != J\_res[i]) {

checker\_result.write\_message("WA. J array is not equal");

checker\_result.write\_verdict(verdict::WA);

return 0;

}

}

for (int i = 0; i < J\_res[N\_source]; i++) {

if (I\_ans[i] != I\_res[i]) {

checker\_result.write\_message("WA. I array is not equal");

checker\_result.write\_verdict(verdict::WA);

return 0;

}

}

double diff = 0.0;

for (int i = 0; i < J\_ans[N\_source]; i++)

diff += (ans[i] - res[i]) \* (ans[i] - res[i]);

if (diff < 1e-6) {

checker\_result.write\_message("AC. Numbers are equal.");

checker\_result.write\_verdict(verdict::AC);

}

else {

checker\_result.write\_message("WA. Output is not correct");

checker\_result.write\_verdict(verdict::WA);

}

checker\_result.write\_time(res\_time \* 1e7);

fclose(file\_res);

fclose(file\_ans);

fclose(file\_source);

return 0;

}

1. **Параллельная реализация алгоритма умножения CCS матриц с использованием технологии OpenMP**

На данном этапе необходимо реализовать параллельную версию алгоритма умножения CCS матриц при помощи технологии OpenMP. Ниже представлена реализация алгоритма и результаты тестирования этого алгоритма.

1. **Параллельная реализация**

Идея реализации:

Реализация алгоритма сильно отличается от реализации последовательного алгоритма. Параллельный алгоритм делится на три этапа:

1. Вычисление размера всех массивов матрицы C.

Сначала нам необходимо рассчитать размер всех массивов матрицы C. Это позволит нам значительно сократить время работы алгоритма, по сравнению с предыдущем алгоритмом умножения, так как память будет выделятся единожды, а также поможет нам избежать конфликтов между потоками во следующем этапе. Идея реализации:

Если посмотреть на формулу, то видно, что для того, что бы узнать, сколько будет элементов в столбце j матрицы C, нам необходимо узнать количество ненулевых комбинации , исключая комбинации, повторно приводящие к одному и тому же элементу в матрице C. Алгоритм:

Мы создаем массив счетчиков, который хранит размер каждого столбца матрицы C. При помощи директивы: ” #pragma omp for “ распараллеливаем перебор всех столбцов матрицы B. Так как столбцы в матрице могут быть не равномерно-заполненными, то для большей эффективности алгоритма создается 100 порции вычислений. Распределение порций по потокам происходит динамически. Каждый поток получает свой false массив булевых переменных размера N, где N – размер матриц. Этот массив и будет работать защитой от комбинации, приводящих к уже вычисленному элементу матрицы C. Алгоритм просматривает элементы столбца r матрицы B, и для каждого элемента проверяет столбец i матрицы A. Если элемент матрицы A ненулевой, то в массив булевых переменных по индексу j записывается true. Далее, вычисляется количество true значении массива, и записывается в массив счетчиков по индексу столбца матрицы B. При выходе из параллельной зоны мы, исходя из данных массива счетчиков, выделяем память под массивы C и IC и заполняем массив JC. Массив C мы заполняем нулями, а массив IC мы заполняем -1. Это пригодится нам на следующем этапе вычислений.

2. Вычисление элементов матрицы C.

Идея схожа с предыдущем алгоритмом. Различие заключается в том, что, на этот раз, мы проходимся по столбцам матрицы B, и, в свою очередь, каждый элемент матрицы B столбца j мы умножаем на каждый элемент столбца i матрицы A, а результат складываем с элементом матрицы C. Распараллеливание происходит по такому же принципу, как и в первом пункте. Такой подход позволяет нам полностью избежать “гонки данных” у потоков, так как каждый поток вычисляет свой собственный столбец матрицы C. Одним из минусов данного алгоритма заключается в том, что итоговый массив IC получается неотсортированным, из-за чего нам необходим третий этап вычислений.

3. Сортировка по убыванию массива IC.

Так как в предыдущем этапе вычислений мы получили неотсортированный массив IC, то выполняем сортировку по столбцам в массиве IC. Распараллеливание идентично первому этапу вычислений.

Реализация алгоритма:

int i, j, k, s, sum = 0, chunk = (N / 100 < 1) ? 1 : N / 100;

int I\_buff;

double val\_buff;

int \*col\_val = new int[N];

bool \*vec;

#pragma omp parallel shared(col\_val, chunk, IA, IB, JA, JB, N, sum) private(i, j, k, vec) num\_threads(threads)

{

vec = new bool[N];

#pragma omp for schedule(static, chunk)

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

col\_val[i] = 0;

#pragma omp for schedule(dynamic, chunk)

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

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

vec[j] = false;

for (j = JB[i]; j < JB[i + 1]; j++) {

for (k = JA[IB[j]]; k < JA[IB[j] + 1]; k++)

vec[IA[k]] = true;

}

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

if (vec[j]) col\_val[i]++;

}

delete[] vec;

}

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

sum = sum + col\_val[i];

C = new double[sum];

IC = new int[sum];

JC = new int[N + 1];

JC[0] = 0;

for (i = 1; i < N + 1; i++) {

JC[i] = JC[i - 1] + col\_val[i - 1];

}

#pragma omp parallel for shared(C, IC, sum) private(i) schedule(static, chunk)

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

C[i] = 0.0;

IC[i] = -1;

}

#pragma omp parallel shared(chunk, A, B, C, IA, IB, IC, JA, JB, JC, N) private(i, j, k, s, I\_buff, val\_buff) num\_threads(threads)

{

#pragma omp for schedule(dynamic, chunk)

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

for (j = JB[i]; j < JB[i + 1]; j++) {

for (k = JA[IB[j]]; k < JA[IB[j] + 1]; k++) {

for (s = JC[i]; s < JC[i + 1]; s++) {

if (IC[s] == IA[k]) {

C[s] += B[j] \* A[k];

break;

}

if (IC[s] == -1) {

IC[s] = IA[k];

C[s] = A[k] \* B[j];

break;

}

}

}

}

}

#pragma omp for schedule(dynamic, chunk)

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

for (j = JC[i]; j < JC[i + 1]; j++) {

for (k = j + 1; k < JC[i + 1]; k++) {

if (IC[j] > IC[k]) {

I\_buff = IC[j];

val\_buff = C[j];

IC[j] = IC[k];

C[j] = C[k];

IC[k] = I\_buff;

C[k] = val\_buff;

}

}

}

}

}

1. **Результаты тестирования**

Тестирование проводилось на компьютере с процессором Core i5-7600k 3.80GHz (4 ядра, 4 потока) на операционной системе Windows. Для проверки эффективности алгоритма был взят 16 тест из генератора, который, в свою очередь, является самым большим тестом в тестирующей системе. Размер матриц 40000, матрицы заполнены на 1%, т.е. в них 16 миллионов элементов, размер двух таких матриц ~ 375 мегабайт, размер матрицы результата ~ 1308 мегабайт.

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 1711 с. | - |
| 2 | 866 с. | 1.97 |
| 3 | 599 с. | 2.86 |
| 4 | 453 с. | 3.78 |

Исходя из представленных данных, мы можем сделать вывод, что параллельный алгоритм дает ускорение близкое к двукратному при увеличении числа потоков, что говорит о высокой эффективности данного алгоритма.

1. **Параллельная реализация алгоритма умножения CCS матриц с использованием технологии TBB**

Идея реализации параллельного алгоритма умножения разреженных CCS матриц при помощи технологии TBB (Intel Threading Building Blocks) полностью повторяет параллельный алгоритм OpenMP версии, с поправкой на особенности TBB системы. Реализация:

struct CCSmatrix {

double\* Vals;

int\* I;

int\* J;

int N;

};

void tbb\_ccsmult(CCSmatrix A, CCSmatrix B, CCSmatrix &C, int threads) {

int grainSize = (A.N / 100 < 1) ? 1 : A.N / 100;

int \*col\_val = new int[A.N];

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

col\_val[i] = 0;

task\_scheduler\_init init(threads);

parallel\_for(blocked\_range<int>(0, A.N, grainSize), [&](blocked\_range<int> range) {

bool \*vec = new bool[A.N];

int i, j, k;

for (i = range.begin(); i < range.end(); i++) {

for (j = 0; j < A.N; j++)

vec[j] = false;

for (j = B.J[i]; j < B.J[i + 1];j++)

for (k = A.J[B.I[j]]; k < A.J[B.I[j] + 1]; k++)

vec[A.I[k]] = true;

for (j = 0; j < A.N; j++)

if (vec[j]) col\_val[i]++;

}

delete[] vec;

});

int sum = 0;

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

sum += col\_val[i];

C.Vals = new double[sum];

C.I = new int[sum];

C.J = new int[C.N + 1];

C.J[0] = 0;

for (int i = 1; i < C.N + 1; i++)

C.J[i] = C.J[i - 1] + col\_val[i - 1];

delete[] col\_val;

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

C.Vals[i] = 0.0;

C.I[i] = -1;

}

parallel\_for(blocked\_range<int>(0, C.N, grainSize), [&](blocked\_range<int> range) {

int i, j, k, s;

for (i = range.begin(); i < range.end(); i++)

for (j = B.J[i]; j < B.J[i + 1]; j++)

for(k = A.J[B.I[j]]; k < A.J[B.I[j] + 1]; k++)

for (s = C.J[i]; s < C.J[i + 1]; s++) {

if (C.I[s] == A.I[k]) {

C.Vals[s] += B.Vals[j] \* A.Vals[k];

break;

}

if (C.I[s] == -1) {

C.I[s] = A.I[k];

C.Vals[s] = A.Vals[k] \* B.Vals[j];

break;

}

}

});

parallel\_for(blocked\_range<int>(0, C.N, grainSize), [&](blocked\_range<int> range) {

int i, j, k;

int I\_buff;

double val\_buff;

for (i = range.begin(); i < range.end(); i++)

for (j = C.J[i]; j < C.J[i + 1]; j++)

for (k = j + 1; k < C.J[i + 1]; k++)

if (C.I[j] > C.I[k]) {

I\_buff = C.I[j];

val\_buff = C.Vals[j];

C.I[j] = C.I[k];

C.Vals[j] = C.Vals[k];

C.I[k] = I\_buff;

C.Vals[k] = val\_buff;

}

});

init.terminate();

}

Тестирование TBB версии проводилось аналогично OpenMP версии, с одинаковыми матрицами в качестве теста.

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 1952 с. | - |
| 2 | 985 с. | 1.98 |
| 3 | 671 с. | 2.91 |
| 4 | 509 с. | 3.83 |

Результаты тестирования показали примерно ту же эффективность ускорения при увеличении числа потоков, что и на OpenMP версии алгоритма, при чем, TBB версия имеет немного больший множитель при увеличении числа потоков, но при этом немного проигрывает в общей скорости работы алгоритма. Для более четкого понимания ниже представлены графики:

**Заключение**

В процессе выполнения работы были реализованы три версии алгоритма умножения разреженных CCS матриц: последовательная, параллельная с использованием OpenMP и параллельная с использованием TBB. Все три реализации верно решают поставленную задачу. Результаты тестирования показали, что OpenMP и TBB реализации схожи по эффективности и позволяют значительно ускорить время работы алгоритма. Все поставленные задачи успешно выполнены.