Федеральное государственное автономное образовательное учреждение высшего

образования «ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ» Факультет математики, механики и компьютерных наук Кафедра математического моделирования Направление подготовки 010501 —«Прикладная математика и информатика»

**Индивидуальная работа на тему: «Параллельное блочное умножение матриц»**

Студентки 4 курса: М. С. Редун

Ростов-на-Дону 2018

**Отчет по индивидуальному заданию «Блочное умножение матриц»**

**Вариант 41.**

**Постановка задачи.**

Написать программу блочного умножения двух матриц C = A\*B. Матрица A верхне-треугольная. Хранится в виде одномерного массива по блочным строкам. Матрица B симметричная, хранится как нижне-треугольная. Хранится в виде одномерного массива по блочным строкам. Распараллелить блочную программу умножения двух матриц C = A\*B с использованием технологии OpenMP двумя способами • Перемножение каждых двух блоков выполнить параллельно • В разных вычислительных ядрах одновременно перемножать разные пары блоков. Определить оптимальные размеры блоков в обоих случаях. Провести численные эксперименты и построить таблицу сравнений времени выполнения различных программных реализаций решения задачи. Определить лучшие реализации. Проверить корректность (правильность) программ.

**Описание программной реализации и результатов исполнения:**

В данной работе представлен блочный механизм умножения матриц для последовательного и параллельного случаев. Для упрощения вычислений выбираются квадратные матрицы n × n с размером блока, равного степени числа 2.

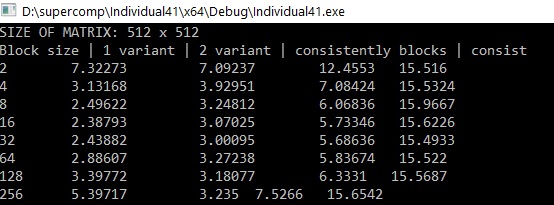
Все рассмотренные методы характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. Однако параллельные алгоритмы являются более эффективными за счет возможности равномерного распределения вычислительной нагрузки между процессорами. В отличие от последовательного, с операциями сложения и умножения, исполняющимися в 1 процессоре. Каждый из потоков отвечает за вычисление одного блока результирующей матрицы С. Задача разделяется на отдельные подзадачи, связанные с перемножением пары блоков исходных матриц (А-верхне-треугольной, В-симметричной, нижне-треугольной) и равные заданному числу потоков-4. Для 1 параллельного случая. 2 вариант позволяет контролировать число потоков за счет распределения вычислений между ядрами процессора.

Так, тестирование программы проводилось на компьютере со следующей конфигурацией:

ПРОЦЕССОР Intel Core i5 4200H ( 2 физических, 4 логических ядра, частота-2.8 Ггц), ОПЕРАТИВНАЯ ПАМЯТЬ 8Gb, ОПЕРАЦИОННАЯ СИСТЕМА Windows 10 Pro x64. Программные средства: компилятор С++ MSVC\_2017\_64 для фреймворка Qt.

**Сравнение результатов**

**Рис-1.Таблица сравнений времени выполнения для разных случаев**

****

**Вывод**

На данном изображении представлена таблица времени выполнения различных программных реализаций задачи. Если начальную матрицу выбрать порядка 512 × 512, то в качестве оптимального размера блока для всех случаев можно считать подматрицy: 32 × 32.

**Описание методов решения**

**Случай 1-параллельное перемножение пары блоков**

Выбирается фиксированное число потоков, равное 4. Задается внешний цикл для блоков матрицы результата С. Каждый поток выполняют 1 итерацию по перемножению пары блоков для данного результирующего блока матрицы С. Блок матрицы С-это перемножение блочной строки на блочный столбец исходных матриц А\*B. Ниже представлена функция перемножения пары блоков в потоке:

//for all threads variables

#pragma omp parallel shared(u, s, res, targets) num\_threads(4)

{

for (int i = 0; i < targets.size(); ++i)

#pragma omp for schedule(dynamic, 1)// перемножение пары блоков(1 итерация цикла) для каждого потока

for (int j = 0; j < targets[i].pairs.size(); ++j) //для каждого результирующего блока перемножение пар в 4 потоках

multiplyBlocks(u, s, res, targets[i].pairs[j].first, targets[i].pairs[j].second, targets[i].target\_block, block\_size);

}

}

**Случай 2-параллельное перемножение разных пар блоков в вычислительных ядрах.**

Число потоков задается равным количеству ядер процессора. Они равномерно распределяются между блоками матрицы С. Так каждая нить выполняет N/n\_threads итераций, где N=число блоков матрицы С. Блок матрицы С получается способом, аналогичным рассмотренному выше. Рассмотрим функцию перемножения пар блоков в ядрах:

//multiplication of blocks in cores

void multiplyConcurrent(const UpperTriangularMatrix & u, const SymmetricMatrix & s, vector<vector<double>>& res, int n\_threads, int block\_size) {

vector<Division> targets = divideToBlocks(res, block\_size);

//common variable list for all threads

#pragma omp parallel shared(u, s, res, targets) num\_threads(n\_threads)

{

#pragma omp for schedule(static, targets.size() / n\_threads)//static distribution of iterations between threads

for (int i = 0; i < targets.size(); ++i)

for (const auto &j : targets[i].pairs)

multiplyBlocks(u, s, res, j.first, j.second, targets[i].target\_block, block\_size);

}

}

**Случай 3-последовательное блочное умножение двух матриц C=A\*B.**

Исходные матрицы А, B разделяются на блоки, равные числу 2 в соответствующей степени. Запоминаются координаты верхнего левого угла каждого из блоков. Для получения блока матрицы Cij произведения нужно выделить i-ю строку блоков матрицы А и j-й столбец блоков матрицы B. А затем просуммировать соответствующие произведения. Индексы рассматриваемых строки и столбца-это индексы блока матрицы С результата. К примеру-блок С[0,0]=A[0,0]\*B[0,0]+ A[0,1]\*B[1,0]. Ниже представлен фрагмент кода:

//sequential multiplication blocks

void multiplyConsBlocks(const UpperTriangularMatrix & u, const SymmetricMatrix & s, vector<vector<double>>& res, const int block\_size) {

vector<Division> targets = divideToBlocks(res, block\_size); //all result blocks

for (int i = 0; i < targets.size(); ++i)

for (int j = 0; j < targets[i].pairs.size(); ++j)

multiplyBlocks(u, s, res, targets[i].pairs[j].first, targets[i].pairs[j].second, targets[i].target\_block, block\_size);

}

**Случай 4- последовательное не блочное умножение двух матриц C=A\*B.**

Матрицы А, B задаются как квадратные, согласованные, в виде одномерных массивов по строкам. При перемножении i строки верхне-треугольной матрицы А на симметричную B и суммирования элементов, получаем результирующую матрицу С. Последняя сохраняется в виде шаблона vector. Рассмотрим код ниже:

//sequential multiplication

void multiplyCons(const UpperTriangularMatrix & u, const SymmetricMatrix & s, vector<vector<double>>& res) {

for (int i = 0; i < res.size(); ++i)

for (int j = 0; j < res.size(); ++j)

{

res[i][j] = 0;

for (int k = 0; k < res.size(); ++k)

res[i][j] += u.at(i, k) \* s.at(k, j);

}

}

**Хранение матриц А и B:**

**A-верхне-треугольн**ая, **B-симметричная, нижне-треугольная**, хранятся в виде векторов по блочным строкам. Для получения искомого элемента в соответствующем столбце, делается сдвиг, равный числу элементов в предыдущей строке и номеру рассматриваемого столбца для **нижне-треугольной** матрицы. В случае с матрицей **А**-перемещение происходит по формуле, аналогичной выше, минус индекс строки(так как j>i):

**B:**

// сохранение матрицы в одну строку

for (int i = 0; i < matrix.size(); ++i)

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

data\_.push\_back(matrix[i][j]);

//create method item access by index for element

double SymmetricMatrix::at(int i\_f, int j\_f) const {

if (j\_f > i\_f) return at(j\_f, i\_f); //если получили элемент из верхнего правого угла

return data\_[sizes\_[i\_f] + j\_f]; //сместились по вектору и нашли элемент с таким индексом

}

**A:**

// сохранение матрицы в одну строку

for (int i = 0; i < matrix.size(); ++i)

for (int j = i; j < matrix[0].size(); ++j)

data\_.push\_back(matrix[i][j]);

//смещение по вектору и полчение элемента

double UpperTriangularMatrix::at(int i\_f, int j\_f) const {

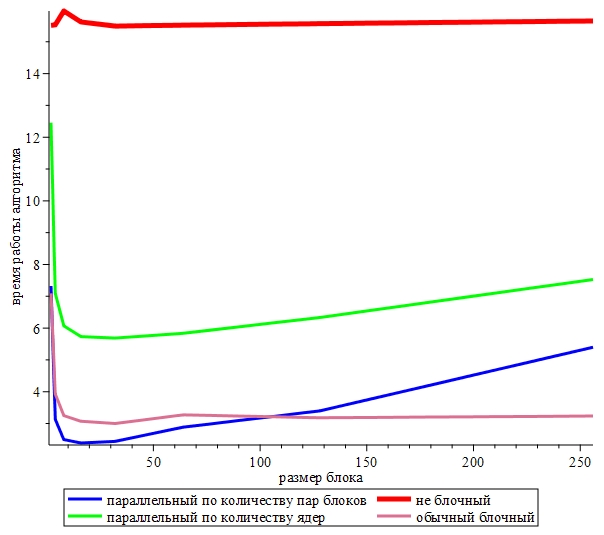
if (j\_f < i\_f) return 0;

return data\_[sizes\_[i\_f] + j\_f - i\_f];

}

**Графики**

**Рис-2.Графики зависимости времени от размера блока**



**Вывод:**

Поточный алгоритм – потоки разделяются по блочным строкам/столбцам. Было выяснено, что такое распределение по потокам работает быстрее всего.

Снимок выше иллюстрируют графическое представление зависимости времени исполнения программы от размера блока. Лучшая реализация изображена на графике “параллельный по количеству пар блоков”, так как содержит минимальное время исполнения алгоритма. А время на вычисления для параллельных случаев по сравнению с последовательным затрачивается в 2 раза меньше при выборе любого из блоков. Так же можно заметить, что для меньших размеров блоков программа работает быстрее всего.