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 №4
по дисциплине «Параллельное программирование»

тема:

«Умножение плотных матриц. Элементы типа double. Блочная схема, алгоритм Фокса»

Выполнил: студент группы 381508
Грачев В.В.

_____Подпись

Нижний Новгород
2018

Оглавление

Постановка задачи 3
Метод решения 3
Схема распараллеливания 3
Программная реализация 4
OpenMP версия 4
TBB версия 4
Подтверждение корректности 5
Результаты экспериментов по оценке масштабируемости 5
Источники 6

Постановка задачи

Даны две квадратные матрицы размера $n \times n$ с элементами типа double. Требуется перемножить эти матрицы, используя P вычислительных узлов. Считается, что размер матрицы кратен корню квадратному из числа вычислительных узлов.

Метод решения

За основу параллельных вычислений для матричного умножения был принят блочный подход. При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков размера $q \times q$, где $q = n/\sqrt{P}$. Сама операция умножения может быть представлена так:

$$\begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0q-1} \\ & & \cdots & \\ A_{q-10} & A_{q-11} & \cdots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \cdots & B_{0q-1} \\ & & \cdots & \\ B_{q-10} & B_{q-11} & \cdots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \cdots & C_{0q-1} \\ & & \cdots & \\ C_{q-10} & C_{q-11} & \cdots & C_{q-1q-1} \end{pmatrix},$$

где каждый блок матрицы C определяется в соответствии с выражением: $C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj}$

Такой способ вычисления произведения формирует подзадачу, которая заключается в перемножении двух подматриц матриц A и B.

Схема распараллеливания

Алгоритм Фокса изначально ориентирован на вычислительные системы с распределённой памятью. В которых, например, используется MPI. Описанный в данном алгоритме способ распределения блоков между задачами позволяет максимально распараллелить умножение матриц и исключает обращение разными потоками к одним и тем же частям матрицы.

Однако в условии системы с общей памятью, в которых, например, используется OpenMP или TBB, потребности в пересылке данных от одного вычислителя к другому нет, поэтому классический алгоритм Фокса можно существенно упростить. Все потоки имеют доступ на чтение к обеим матрицам и на запись к разным строкам результирующей матрицы. Поэтому достаточно каждому из потоков выделить свой блок матрицы A и B размером $q \times q$, где $q = n/\sqrt{P}$. Получающиеся в результате суммы произведений части строк на часть столбцов необходимо прибавлять к результирующей матрице.

Программная реализация

OpenMP версия

```
void MatrixMult(double *A, double *B, double *C, int n, int block_size){
    int i, j, k, jj, kk;
    double temp;

    #pragma omp parallel shared(A, B, C, n) private(i, j, k, jj, kk, temp)
    {
        #pragma omp for schedule (static)
        for (jj = 0; jj < n; jj += block_size)
            for (kk = 0; kk < n; kk += block_size)
                for (i = 0; i < n; i++)
                    for (j = jj;
                        j < ((jj + block_size) > n ? n : (jj + block_size)); j++)
                    {
                        temp = 0.0;
                        for (k = kk;
                            k < ((kk + block_size) > n ? n : (kk + block_size)); k++)
                            temp += A[i*n + k] * B[k*n + j];
                        C[i*n + j] += temp;
                    }
    }
}
```

TBB версия

```
class MatrixMultiplicator{
private:
    const double *A, *B;
    double *const C;
    int const n;
public:
    MatrixMultiplicator(double *_A, double *_B, double *_C, int size) : A(_A), B(_B),
C(_C), n(size){}
    void operator()(const blocked_range2d<int, int>& r) const
    {
        int r_begin = r.rows().begin(), r_end = r.rows().end();
        int c_begin = r.cols().begin(), c_end = r.cols().end();
        double temp;

        for (int i = r_begin; i < r_end; ++i)
            for (int j = c_begin; j < c_end; j++){
                temp = 0.0;
                for (int k = 0; k < n; k++)
                    temp += A[i*n + k] * B[k * n + j];
                C[i*n + j] += temp;
            }
    }
};

void MatrixMult(double *A, double *B, double *C, int n, int block_size, int
num_threads){
    task_scheduler_init init(num_threads);
    parallel_for(blocked_range2d<int>(0, n, block_size, 0, n, block_size),
        MatrixMultiplicator(A, B, C, n));
}
```

Подтверждение корректности

Для проверки корректности вычислений используется Checker (файл "checker.cpp"). С его помощью осуществляется выставление вердикта о правильности результатов, полученных в ходе работы последовательной версии алгоритма и параллельной версии алгоритма на тестовых примерах.

В качестве последовательной версии используется следующая общепринятая реализация, корректность которой не вызывает сомнений:

```
void MatrixMult(double *A, double *B, double *C, int n){
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                C[i * n + j] += A[i * n + k] * B[k * n + j];
}
```

Для создания тестов используется Generator(файл "generator.cpp"). Всего было сгенерировано 20 тестов различной размерности (от 1 элемента до 1024). Каждый элемент матрицы создается случайным образом с помощью нормального распределения. Элементы принадлежат промежутку от -10000,0 до 10000,0. В результате сгенерированные входные данные записываться в бинарный файл.

К каждому тесту прилагается ответ, вычисленный в результате работы последовательной версии.

Checker сравнивает матрицу из файла с ответом к тесту и матрицу из выходного файла параллельной программы. Тест считается пройденным, если ошибка, накопленная после сравнения результирующих матриц, не превысила 10^{-5}.

Реализация Checker'а и Generator'а опирается на соответствующие реализации, приведенные в примере.

Все 20 подготовленных тестов были успешно пройдены каждой параллельной версией.

Результаты экспериментов по оценке масштабируемости

Были проведены эксперименты с использованием 1, 4 и 16 потоков с размером матрицы 16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024 и 2048x2048 элементов. Вычисления проводились на компьютере с процессором Intel Core i5-8250U с четырьмя физическими ядрами, 8 Гб RAM, под управлением Windows 10 Pro. Разработка программ проводилась в среде Microsoft Visual Studio 2017, для компиляции использовался стандартный компилятор, предоставляемый средой, с включенной полной оптимизацией. Получены следующие результаты:

Размер матрицы	Длительность работы				
	Последовательный алгоритм, сек	4 потока (omp), сек	4 потока (tbb), сек	16 потоков (omp), сек	16 потоков (tbb),сек
16x16	0,00001100	0,00052000	0,00086105	0,00156000	0,00258315
32x32	0,00006800	0,00060400	0,00090780	0,00124000	0,00186370
64x64	0,00055600	0,00074300	0,00124015	0,00136000	0,00226999
128x128	0,00445400	0,00263500	0,00277236	0,00328800	0,00345940
256x256	0,03420700	0,01267600	0,01328805	0,01453000	0,01523157
512x512	0,30141500	0,12309200	0,14155580	0,13993200	0,16092180
1024x1024	8,90003500	3,10761700	3,37541375	1,72330795	1,87181282
2048x2048	127,35128500	54,88769700	62,51582720	45,32156800	51,62022581

Можно вычислить ускорение для каждого из рассмотренных вариантов:

Размер матрицы	Ускорение			
	4 потока (omp), сек	4 потока (tbb), сек	16 потоков (omp), сек	16 потоков (tbb),сек
16x16	0,021153846	0,0127751	0,007051282	0,004258367
32x32	0,112582781	0,074906367	0,05483871	0,03648665
64x64	0,748317631	0,448332863	0,408823529	0,244934792
128x128	1,690322581	1,606573461	1,354622871	1,287506408
256x256	2,698564216	2,574267857	2,354232622	2,245796239
512x512	2,44869691	2,555161993	2,154010519	1,873052626
1024x1024	2,863942049	2,636724165	5,164506437	4,754767619
2048x2048	2,320215494	2,037104693	2,809948786	2,467081129

Видно, что достичь ускорения равного числу потоков не удалось. Это можно объяснить необходимостью синхронизации при вычислении элементов результирующей матрицы и накладными расходами, связанными с обслуживанием нескольких потоков. Оптимальным является использование 4 потоков.

Источники

https://bit.ly/2s1obCa
https://studfiles.net/preview/5170718/page:15/
http://www.intuit.ru/studies/courses/1156/190/lecture/4954?page=3