МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельное умножение матриц на кольце

Отчет по лабораторной работе № 3

Выполнили студенты: Прибавкин Д.Д.
Мазяйкина Д.В.

Группа: 6407-010302D

Преподаватель: Головашкин Д.Л.

Самара 2018

СОДЕРЖАНИЕ

Задание

Вариант 4

В данном варианте требуется реализовать столбцовые алгоритмы параллельного и последовательного перемножение матриц на кольце.

ВВЕДЕНИЕ

Применение параллельных вычислительных систем является перспективным направлением развития компьютерной индустрии.

В течение трех десятков лет допивались повышения производительности компьютера за счёт повышения тактовой частоты процессора. Но пришлось отказаться от этого способа повышения быстродействия из-за ограничения на потребляемую мощность. Увеличение количества вычислительных устройств послужило решением данной проблемы. На данный момент затруднительно найти компьютер с одним процессорным ядром.

Главным препятствием к использованию графических многопроцессорных ЭВМ для параллельных вычислений является отсутствие эффективных параллельных методов, учитывающих особенности их архитектуры. Это связано с тем, что алгоритмы, разработанные для последовательного выполнения, не могут быть эффективно использованы для параллельных систем. Поэтому все еще актуальна проблема повышения эффективности параллельных систем.

Таким образом, несмотря на результаты обширных исследований в области параллельных вычислений, работы в этом направлении не утрачивают своей значимости и требуют дальнейшего развития в связи с массовым распространением параллельных вычислительных систем и отсутствием должной программной составляющей.

Теоретические сведения

Пусть A и B – две квадратные матрицы размерности n.

$$D = C + AB$$

Параллельный алгоритм

Инициализация $\{\mu, p, n, r = \frac{n}{p}, col = \big((\mu - 1)r + 1 : \mu r\big), Cloc = C(:, col), Bloc = B(:, col), Aloc = A(:, col), left, right\}$

for t = 1 : p

send($Aloc$, right);

recv($Aloc$, left);

$\tau = \mu - t$;

if $\tau \leq 0$ then $\tau = \tau + p$;

$Cloc = Cloc + Aloc * Bloc((\tau - 1)r + 1 : \tau r, :)$;

end

Последовательный алгоритм

```
for (int k = 0; k < N; k++) {
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; j++) {
            res(i, j) += A(i, k) * B(k, j);
        }
    }
}
```

Цель эксперимента

Целью эксперимента является выявление зависимости времени работы параллельного и последовательного алгоритмов перемножения матриц от их размерности, получение зависимости ускорения параллельного алгоритма, выяснение преимуществ и недостатков для каждого алгоритма.

Для выявления зависимости времени работы алгоритмов от размерности матриц, проанализируем время, затраченное на выполнение программы, при перемножении матриц разных размерностей.

Использованный инструментарий

Для вычислений использовалось следующее оборудование:

- аппаратные: AMD Phenom(tm) II X4 960T 3GHz, 8Gb RAM;
- системные: Windows 10 Pro;
- программные: g++ GNU Compiler Collection version 7.3.0;
- библиотека MPI Microsoft HPC Pack 2008 R2;
- для построения графиков использовался язык Python v 3.6.5;
- язык программирования: C++ стандарта C++11.

Данное оборудование удовлетворят всем необходимым аппаратным и программным требованиям, т.к. оно позволяет выполнять высокопроизводительные вычисления с достаточной степенью точности.

Данный язык программирования удовлетворяет необходимым требованиям, т. к. он позволяет взаимодействовать с библиотекой MPI Microsoft HPC.

Параметры эксперимента

В качестве нижней границы была взята размерность матриц 50×50, поскольку при меньших значениях результаты не являются достаточно наглядными. Максимальное значение размерности матриц 1400x1400 элементов. Данный диапазон значений обусловлен увеличением времени работы программы для матриц с размерностью свыше 1400×1400. Вместе с тем результат для данной области дает объективное представление о качестве работы алгоритмов. Увеличение размерности было проведено с шагом 100. Количество процессоров для параллельного алгоритма равно 2.

Теоретическое ожидание

На основе теоретических данных и лекционных материалов мы предположили, что параллельный алгоритм будет работать быстрее последовательного из-за разделения операций между двумя потоками.

В силу закона Амдала при увеличении размерности матриц ускорение вычисления должно стремиться к двум. Стоит отметить, что при недостаточно больших размерностях матрицы возможен случай когда, параллельный алгоритм будет показывать результат хуже последовательного, что обусловлено затратами времени на пересылку данных.

При увеличении размера матрицы, количество ее элементов растет в геометрической прогрессии, поэтому время, затраченное последовательным алгоритмом**,** будет так же увеличиваться в геометрической прогрессии.

Описание результатов

В процессе выполнения работы были получены результаты, представленные в таблице 1.

Размер перемножаемых матриц	Время работы, с	
	Последовательный	Параллельный
50	0.0029	0.0036
100	0.0121	0.0113
200	0.1205	0.0726
300	0.2607	0.2564
400	1.0074	0.5745
500	2.9630	1.1357
600	2.8892	1.7065
700	4.4124	2.6910
800	5.6607	3.2550
900	7.4	4.2879
1000	10.0150	5.0218
1100	12.0729	6.0036
1200	12.105	6.01
1300	14.125	7.362
1400	14.725	7.544

Таблица 1 - Время выполнения программы, с

График зависимости ускорения вычислений (отношение времени работы последовательного алгоритма к времени работы параллельного) от размерности матрицы приведен на рисунке 2.

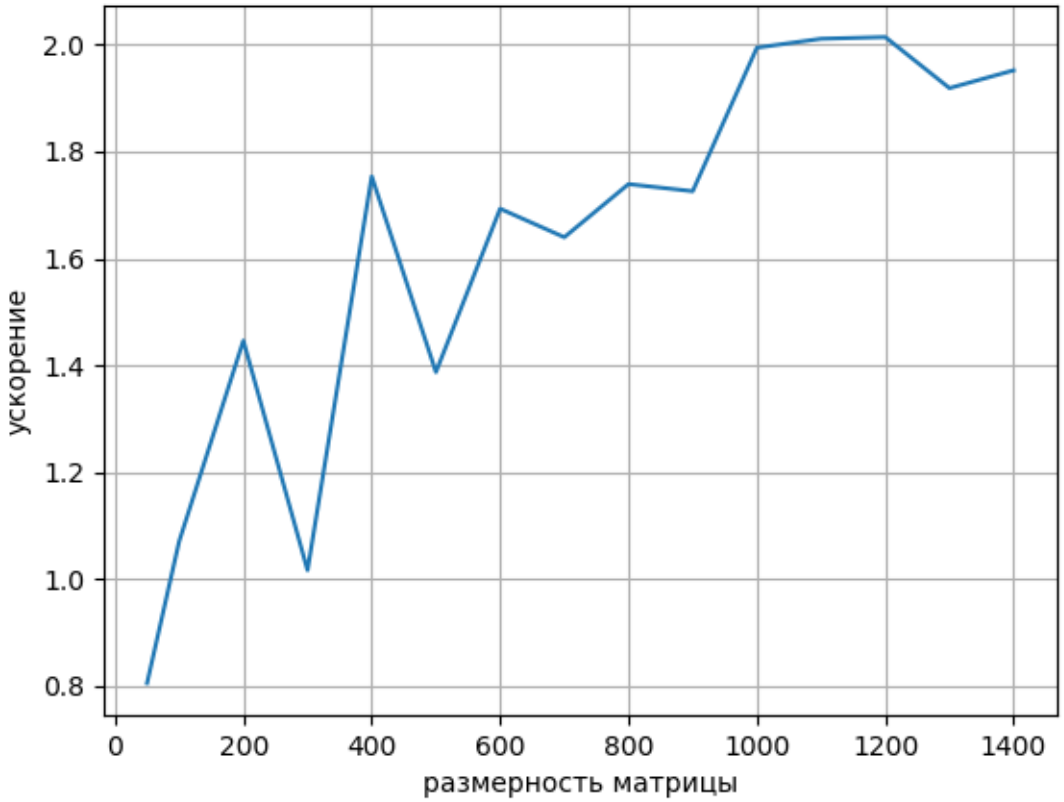

Рисунок 2 – График зависимости ускорения вычислений от размерности матрицы

Из рисунка 2 видно, что ускорение растет с увеличением размерности матрицы.

Так же, стоит отметить, что при малых размерностях матриц, последовательный алгоритм работает быстрее.

Небольшие проседания ускорения обусловлены работой операционной системы.

Анализ результатов

При малых размерностях матриц параллельный алгоритм затрачивает больше времени на инициализацию и пересылку данных, поэтому он оказывается медленнее последовательного.

Все результаты являются приблизительными, так как на времени работы алгоритма сказывается загруженность процессора в данный момент. Поэтому в разный момент времени, выполнение одного и того же действия занимает различное время.

Рост и спад ускорения зависит от доли параллельного кода, который, в свою очередь, зависит от размера матрицы.

Выполнение алгоритма на 2 процессорах быстрее, чем выполнение того же алгоритма последовательно, что соответствует ожидаемому результату.

ЗАКЛЮЧЕНИЕ

В ходе данной работы были продемонстрированы различия между последовательной и параллельной реализациями алгоритмов перемножения матриц.

В итоге проделанной работы можно прийти к выводу, что для матриц размерностью меньше 100×100 желательно применять последовательный алгоритм, а для больших размерностей стоит использовать параллельный алгоритм.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Голуб Дж. Матричные вычисления / Дж. Голуб, Ч.Ван. Лоун; пер. с англ. Ю.М. Нечепуренко, А.Ю. Романова, А.В. Собянина, Е.Е. Тыртышникова; по ред. В.В. Воеводина. – М.: Мир, 1999. – 548 с

Приложение А

Код программы

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <fstream>
#include <iostream>

using namespace std;
void MatrixMultiplication(double *&A, double *&B, double *&C, int Size) {     int   i,   j,
k;
        for (i = 0; i < Size; i++) {
                for (j = 0; j < Size; j++) {
                        C[i*Size + j] = 0;
                        for (k = 0; k < Size; k++)
                                C[i*Size + j] += A[i*Size + k] * B[j*Size + k];
                }
        }
}
int main(int argc, char* argv[]) {
        int procRank, procNum, N = 100, r, *col, i, j, left, right, t, T;
        const int NMAX = 100;
        double start_time, end_time;
        double *a_loc, *b_loc, *c_loc;
        double *a = new double[NMAX*NMAX];
        double * b= new double[NMAX*NMAX];
        double *c = new double[NMAX*NMAX];

        for (i = 0; i<N; i++) {
                for (int j = 0; j<N; j++) {
                        a[i*N + j] = 1;
                        b[i*N + j] = 2;
                        c[i*N + j] = 0.0;        }
        }
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &procNum);
        MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
        if (procRank - 1<0) left = procNum - 1;
        else left = procRank - 1;
        if (procRank == procNum - 1) right = 0;
        else right = procRank + 1;
        r = N / procNum;
        a_loc = (double *)malloc(N*r*sizeof(double));
        b_loc = (double *)malloc(N*r*sizeof(double));
        c_loc = (double *)malloc(N*r*sizeof(double));
        col = (int *)malloc(r*sizeof(int));
        for (i = 0; i<r; i++) {
                col[i] = procRank*r + i;
        }
        for (int i = 0; i<r; i++) {
```

```
                for (j = 0; j<N; j++) {
                        a_loc[i*N + j] = a[col[i] * N + j];
                        c_loc[i*N + j] = c[col[i] * N + j];
                        b_loc[i*N + j] = b[col[i] * N + j];
                }
        }
        start_time = MPI_Wtime();
        for (t = 0; t < procNum; t++) {
                MPI_Send(a_loc, N*r, MPI_DOUBLE, right, 0, MPI_COMM_WORLD);
                MPI_Recv(a_loc, N*r, MPI_DOUBLE, left, 0, MPI_COMM_WORLD,
&status);
                T = procRank - t;
                if (T <= 0) T = T + procNum;
                for (int k = 0; k < N*r; ++k) {
                        for (j = (T-1)*r; j < T*r; j++) {
                                c_loc[k] += a_loc[k] * b_loc[j];
                        }
                }
        }
        MPI_Gather(c_loc, N*r, MPI_DOUBLE, c, N*r, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    end_time = MPI_Wtime();
    end_time = end_time - start_time;
        if (procRank == 0) {
                printf("\n Time: %f\n", end_time);
        }
        MPI_Finalize();
        return 0;
```