Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Лабораторная работа №1 «Многопоточные программы на C/C++ с использованием Threads и OpenMP»

Дисциплина: Параллельные вычисления

Выполнил студент гр. 13541/3 _____ Олейник М.А.
(подпись)

Руководитель _____ Стручков И.В.
(подпись)

Санкт – Петербург
2018

Содержание

1 Задание

В данной работе необходимо реализовать последовательный и параллельный алгоритмы умножение двух матриц.

2 Постановка задачи

Умножение матрицы **A** размера m x n и матрицы **B** размера n x l приводит к получению матрицы **C** размера l x m, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l .$$

Каждый элемент результирующей матрицы **C** есть скалярное произведение соответствующих строки матрицы **A** и столбца матрицы **B**:

$$c_{ij} = a_i, b_j^T \ , \ a_i = a_{i0,} a_{i1}, ..., a_{in-1} \ , \ b_j^T = b_{0j}, b_{1j}, ..., b_{n-1j}^{\ T} .$$

Этот алгоритм предполагает выполнение m·n·l операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера n x n количество выполненных операций имеет порядок $O(n^3)$.

3 Последовательный алгоритм

3.1 Описание алгоритма

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами (листинг 3.1):

Листинг 3.1 – Последовательный алгоритм умножения матриц.

```
int n1;
int m1 = n2;
int m2;
int matrixA[n1][m1];
int matrixB[n2][m2];
int result[n1][m2];
for (int i = 0; i < n1; i++) {
            for (int j = 0; j < m2; j++) {
                  for (int k = 0; k < m1; k++) {
                        result[i][j] += (matrix1[i][k] * matrix2[k][j]);
                  }
            }
      }
```

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы C. При выполнении одной итерации внешнего цикла (цикла по переменной i) вычисляется одна строка результирующей матрицы (рис 3.1):

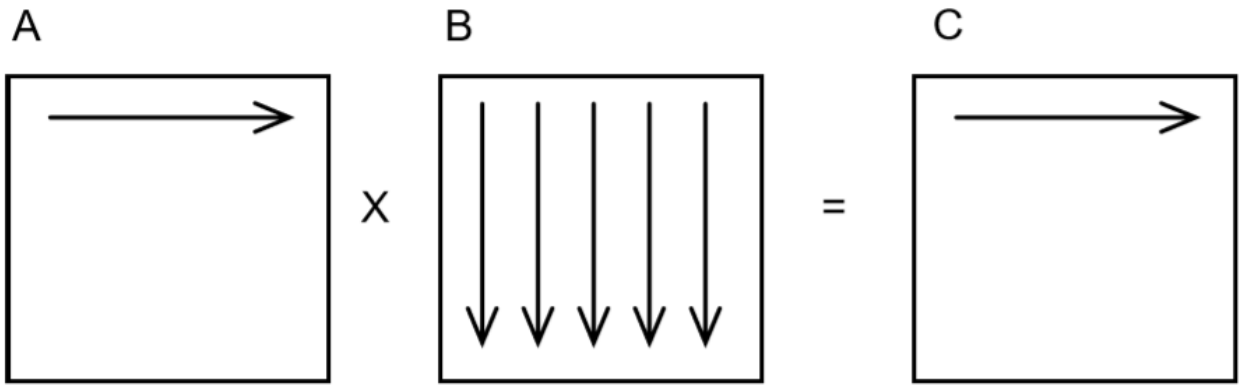

Рисунок 3.2 – Принцип умножения матриц.

3.2 Программная реализация

На языке C++ была написана следующая программа (листинг 3.2):

Листинг 3.2 – Программа последовательного умножения матриц.

```
#include <stdio.h>
#include <tchar.h>
#include <iostream>
#include <ctime>

using namespace std;

int n1 = 1000;
int m1 = 500;
int n2 = 500;
int m2 = 1200;

//Матрица результата n1 x m2
static int **result;

void randomiseMatrix(int **matrix, int n, int m) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < m; j++) {
			matrix[i][j] = rand() % 11;
		}
	}
}

//Производит умножение матрицы размером n1 x m1
//на матрицу размером n2 x m2
int** matrixMulti(int **matrix1, int n1, int m1, int **matrix2, int n2, int m2) {
	//Умножение по формуле
	for (int i = 0; i < n1; i++) {
		for (int j = 0; j < m2; j++) {
			for (int k = 0; k < m1; k++) {
				result[i][j] += (matrix1[i][k] * matrix2[k][j]);
			}
		}
	}
	return result;
}

int main(int argc, char** argv)
{
	srand(time(NULL));
```

```
	//Матрица n1 x m1
	int **matrix1;
	//Матрица n2 x m2
	int **matrix2;

	matrix1 = (int**)malloc(sizeof(int*)*n1);
	for (int i = 0; i < n1; i++) {
		matrix1[i] = (int*)malloc(sizeof(int)*m1);
	}
	matrix2 = (int**)malloc(sizeof(int*)*n2);
	for (int i = 0; i < n2; i++) {
		matrix2[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Генерируем случайные матрицы для умножения
	randomiseMatrix(matrix1, n1, m1);
	randomiseMatrix(matrix2, n2, m2);

	//Выделяем память под результат умножения
	result = (int**)malloc(sizeof(int*)*n1);
	for (int i = 0; i < n1; i++) {
		result[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Обнуляем матрицу результатов
	for (int i = 0; i < n1; i++) {
		for (int j = 0; j < m2; j++) {
			result[i][j] = 0;
		}
	}

	//Специальный тип данных из библиотеки time.h
	clock_t currentTime;
	//Берем текущее системное время
	currentTime = clock();

	matrixMulti(matrix1, n1, m1, matrix2, n2, m2);

	//Берем разницу
	currentTime = clock() - currentTime;

	cout << currentTime << " ms";

    return 0;
}
```

4 Параллельный алгоритм

4.1 Описание алгоритма

Рассмотрим параллельный алгоритм умножения матриц, в основу которого будет положено разбиение матрицы **A** на непрерывные последовательности строк (горизонтальные полосы).

Для вычисления одной строки матрицы C необходимо, чтобы в каждой подзадаче содержалась строка матрицы A и был обеспечен доступ ко всем столбцам матрицы B. Способ организации параллельных вычислений представлен на рис. 4.1.

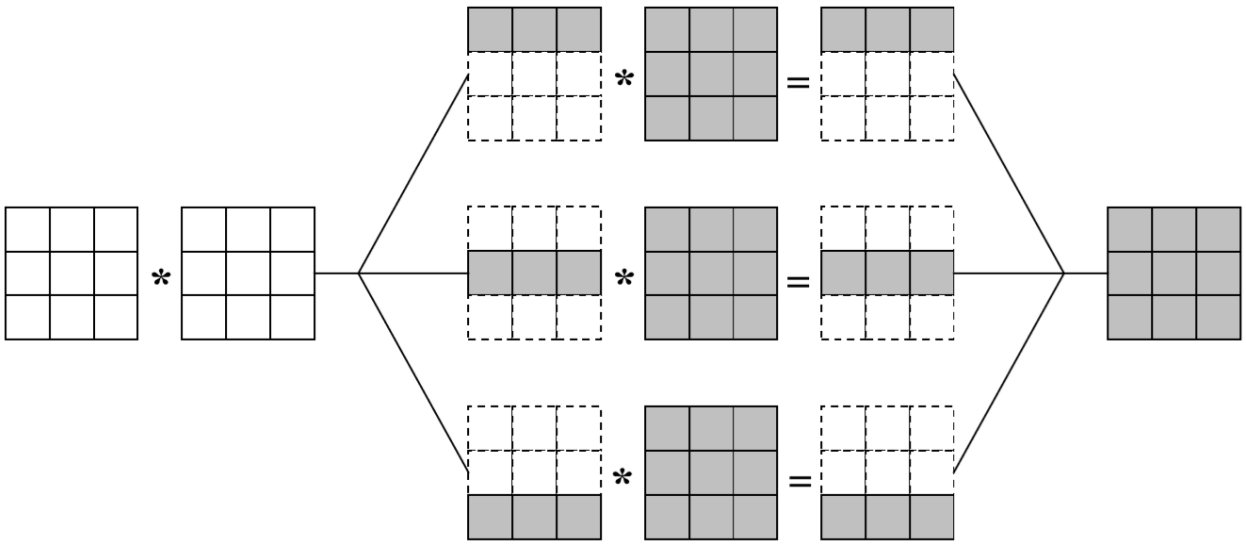

Рисунок 4.1 - Организация вычислений при выполнении параллельного алгоритма умножения матриц, основанного на разделении матриц по строкам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда размер n матриц оказывается больше, чем число p вычислительных элементов (процессоров и/или ядер), базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк матрицы. В этом случае исходная матрица **A** и матрица результат **C** разбиваются на ряд горизонтальных полос.

4.2 Программная реализация с помощью Threads

На языке C++ была написана следующая программа (листинг 4.1):

Листинг 4.1 – Программа параллельного умножения матриц с использованием Threads (потоков).

```
#include <stdio.h>
#include <tchar.h>
#include <iostream>
#include <ctime>
#include <process.h>
#include <thread>
#include <fstream>

using namespace std;

int n1 = 1000;
int m1 = 500;
int n2 = 500;
int m2 = 1200;

//Матрица n1 x m1
static int **matrix1;
//Матрица n2 x m2
static int **matrix2;
//Матрица результата n1 x m2
static int **result;

int counter = 0;
const int threads = 4;
```

```
typedef struct
{
	int from;
	int to;
} Params;

Params params[threads];

void matrixMultiThread(void *param) {
	Params *ptr = (Params*)param;
	//Умножение по формуле с строки from по строку to
	for (int i = ptr->from; i < ptr->to; i++) {
		for (int j = 0; j < m2; j++) {
			for (int k = 0; k < m1; k++) {
				result[i][j] += matrix1[i][k] * matrix2[k][j];
			}
		}
	}
	counter++;
}

void randomiseMatrix(int **matrix, int n, int m) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < m; j++) {
			matrix[i][j] = rand() % 11;
		}
	}
}

int main(int argc, char** argv)
{
	srand(time(NULL));

	//Выделяем память для матриц
	matrix1 = (int**)malloc(sizeof(int*)*n1);
	for (int i = 0; i < n1; i++) {
		matrix1[i] = (int*)malloc(sizeof(int)*m1);
	}
	matrix2 = (int**)malloc(sizeof(int*)*n2);
	for (int i = 0; i < n2; i++) {
		matrix2[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Генерируем случайные матрицы для умножения
	randomiseMatrix(matrix1, n1, m1);
	randomiseMatrix(matrix2, n2, m2);

	//Выделяем память для матрицы результатов
	result = (int**)malloc(sizeof(int*)*n1);;
	for (int i = 0; i < n1; i++) {
		result[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Обнуляем матрицу результатов
	for (int i = 0; i < n1; i++) {
		for (int j = 0; j < m2; j++) {
			result[i][j] = 0;
		}
	}

	static Params params[threads];
	//Заполняем параметры для функций потоков
	int from = 0, to;
```

```
	int q = n1 / threads,
		r = n1 % threads;

	for (int i = 0; i < threads; i++) {
		params[i].from = from;
		to = from + q + (i < r ? 1 : 0);
		params[i].to = to;
		from = to;
	}

	//Специальный тип данных из библиотеки time.h
	clock_t currentTime;
	//Берем текущее системное время
	currentTime = clock();


	for (int i = 0; i < threads - 1; i++) {
		_beginthread(matrixMultiThread, 0, (void*)&params[i]);
	}

	matrixMultiThread((void*)&params[threads - 1]);

	while (counter != threads);

	//Берем разницу
	currentTime = clock() - currentTime;
	cout << currentTime << " ms";

	return 0;
}
```

4.3 Программная реализация с помощью OpenMP

Для того, чтобы разработать параллельную программу, реализующую описанный подход при помощи технологии OpenMP, необходимо внести минимальные изменения в функцию умножения матриц последовательной версии программы. Достаточно добавить одну директиву parallel for для внешнего цикла расчета произведения матрицы и указать количество потоков.

На языке C++ была написана следующая программа (листинг 4.2):

Листинг 4.2 – Программа параллельного умножения матриц с использованием Threads (потоков).

```
#include <stdio.h>
#include <tchar.h>
#include <iostream>
#include <ctime>
#include <fstream>
#include <omp.h>

using namespace std;

const int threads = 8;

void randomiseMatrix(int **matrix, int n, int m) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < m; j++) {
			matrix[i][j] = rand() % 11;
		}
	}
```

```
}

int main(int argc, char** argv) {
	srand(time(NULL));
	int n1 = 1000;
	int m1 = 500;
	int n2 = 500;
	int m2 = 1200;

	//Матрица n1 x m1
	int **matrix1;
	//Матрица n2 x m2
	int **matrix2;

	//Выделяем память для матриц
	matrix1 = (int**)malloc(sizeof(int*)*n1);
	for (int i = 0; i < n1; i++) {
		matrix1[i] = (int*)malloc(sizeof(int)*m1);
	}
	matrix2 = (int**)malloc(sizeof(int*)*n2);
	for (int i = 0; i < n2; i++) {
		matrix2[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Генерируем случайные матрицы для умножения
	randomiseMatrix(matrix1, n1, m1);
	randomiseMatrix(matrix2, n2, m2);

	//Выделяем память для матрицы результатов
	int **result = (int**)malloc(sizeof(int*)*n1);;
	for (int i = 0; i < n1; i++) {
		result[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Обнуляем матрицу результатов
	for (int i = 0; i < n1; i++) {
		for (int j = 0; j < m2; j++) {
			result[i][j] = 0;
		}
	}

	//Запрещаем библиотеке openmp менять число потоков во время исполнения
	omp_set_dynamic(0);

	//Устанавливаем число потоков
	omp_set_num_threads(threads);

	//Специальный тип данных из библиотеки time.h
	clock_t currentTime;
	//Берем текущее системное время
	currentTime = clock();

	int i, j, k;
#pragma omp parallel for shared(matrix1, matrix2, result) private(i, j, k)
	for (i = 0; i < n1; i++) {
		for (j = 0; j < m2; j++) {
			for (k = 0; k < m1; k++) {
				result[i][j] += (matrix1[i][k] * matrix2[k][j]);
			}
		}
	}

	//Берем разницу
	currentTime = clock() - currentTime;
```

```
        cout << currentTime << " ms";
        return 0;
}
```

5 Результаты вычислительных экспериментов

Характеристики системы, на которой выполнялись вычислительные эксперименты:

Характеристика	Значение
ОС	Windows 10 Pro 64-bit
Построение	10.0, Build 16299 (16299.rs3_release.170928-1534)
Процессор	AMD Phenom(tm) II X4 B55 Processor (4 CPUs), 3.3GHz
Оперативная память	10240 MB

Разработка программ проводилась в среде Microsoft Visual Studio 2017. оптимизация компилятора была отключена.

5.1 Эксперимент 1

В данных тестах перемножались матрицы размеров (1000x500) и (500x1200) элементов, заполненные с помощью генератора случайных чисел числами в диапазоне 0-10. Все результаты указаны в миллисекундах.

Таблица 5.1 – Результаты вычислений программ последовательного (соответствует столбцу с количеством потоков - 1) и параллельного (с помощью Threads) алгоритмов. Время указано в миллисекундах.

Итерация	Кол-во потоков							
	1	2	3	4	5	6	7	8
1	7917	3834	2595	2095	2084	2104	2108	2063
2	7854	3930	2707	2056	2128	2093	2156	2126
3	7844	4023	2615	2069	2063	2091	2045	2044
4	8051	3844	2651	2208	2080	2088	2096	2025
5	8254	3939	2674	2054	2101	2041	2087	2109
6	7836	3876	2661	2127	2169	2059	2039	2120
7	7877	3917	2588	2069	2049	2084	2057	2075
8	8117	4016	2773	2101	2077	2058	2084	2042
9	7847	4720	2605	2084	2044	2047	2065	2070
10	7734	4084	2678	2237	2061	2106	2052	2108
Среднее	7933,1	4018,3	2654,7	2110	2085,6	2077,1	2078,9	2078,2

Таблица 5.2 – Результаты вычислений программы параллельного (с помощью OpenMP) алгоритма. Время указано в миллисекундах.

Итерация	Кол-во потоков							
	1	2	3	4	5	6	7	8
1	7769	3817	2502	1985	1960	1959	1982	1917
2	8048	3820	2503	1981	1957	1965	2045	2091
3	7871	3759	2488	1957	1937	1959	1982	1975

4	7825	3768	2520	2004	1980	1911	1969	1918
5	7828	3778	2498	1972	1940	2014	1940	1970
6	7825	3794	2487	1973	1958	1957	1997	1951
7	7758	3764	2503	1950	2059	1977	1934	1917
8	7778	3765	2518	2008	1997	1949	1955	1956
9	7714	3833	2524	2021	1947	1919	1955	2001
10	7896	3810	2495	1975	1980	1965	1996	2020
Среднее	7831,2	3790,8	2503,8	1982,6	1971,5	1957,5	1975,5	1971,6

Рисунок 5.1 - Относительное время работы в зависимости от количества потоков.

Таблица 5.3 – Коэффициент ускорения расчета в сравнении с временем вычисления задачи одним потоком.

Потоки	Коэффициент ускорения	
	Threads	OpenMP
1	1,00	1,00
2	1,97	2,07
3	2,99	3,13
4	3,76	3,95
5	3,80	3,97
6	3,82	4,00
7	3,82	3,97
8	3,82	3,97

5.2 Эксперимент 2

В данных тестах перемножались матрицы размеров (1000x500) и (500x1200) элементов, заполненные с помощью генератора случайных чисел числами в диапазоне 0-10. Все результаты указаны в миллисекундах.

Было произведено 100 измерений для программы с Threads с 4 потоками и 100 измерений для программы с OpenMP с 4 потоками. Полученные результаты

Таблица 5.4 – Обработка полученных результатов. Время указано в миллисекундах.

	Threads	OpenMP
Среднее значение	2172,055	2005,318
Дисперсия	1566,052	106,4942
Доверительный интервал (P=0,95)	2164 - 2179	2003 - 2007

Заключение

Можно отметить, что выполненные эксперименты показывают почти идеальное ускорение вычислений для разработанного параллельного алгоритма умножения.

Становится очевидным то, что нет особого смысла использовать больше потоков, чем ядер в процессоре вычисляющей машины – это не приносит требуемого ускорения.

По результатам сравнения OpenMP и Threads (потоков) можно сказать, что OpenMP справляется с задачей ускорения немного лучше, чем ручное разделение расчетов на потоки. Но куда более веским плюсом в пользу OpenMP является то, что его подключение является очень простой задачей – достаточно понимать, какую часть программы можно распараллелить и будет достаточно указать на это OpenMP директивой (или несколькими), чтобы программа с последовательным алгоритмом превратилась в программу с параллельными вычислениями. В данном случае, хватило всего одной строчки кода, чтобы программа с последовательным алгоритмом стала использовать параллельные вычисления.

Список использованных источников

[1] Гергель В.П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем. Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И.Лобачевского, 2010