Государственное образовательное учреждение высшего профессионального образования
«Московский Государственный Технический Университет им Н. Э. Баумана.»

Отчет
По лабораторной работе №4
По курсу «Анализ Алгоритмов»

Тема: «Параллельные алгоритмы умножения матриц»

Студент: Губайдуллин Р.Т.
Группа: ИУ7-51

Преподаватели: Волкова Л.Л.
Строганов Ю.В.

Постановка задачи

В ходе лабораторной работы выполнить:

1. Реализовать два алгоритма параллельного умножения:
   - Базовый;
   - Модернизированный алгоритм Винограда
2. Сравнить время работы алгоритмов параллельного умножения матриц с последовательным умножением.

OpenMP (Open Multi-Processing)

Для распараллеливания операций используется OpenMP (Open Multi-Processing) - открытый стандарт для распараллеливания программ на языка Си, C++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков).

Задачи, выполняемые потоками параллельно, также как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка - прагм.

Количество создаваемых потоков может регулироваться как самой программой при помощи вызова библиотечных процедур, так и извне, при помощи переменных окружения.

Ключевыми элементами OpenMP являются

- Конструкции для создания потоков (директива parallel),
- Конструкции распределения работы между потоками (директивы do/for и section),
- Конструкции для управления работой с данными (выражения shared и private для определения класса памяти переменных),
- Конструкции для синхронизации потоков (директивы critical, atomic и barrier),
- Процедуры библиотеки поддержки выполнения (например, omp_get_thread_num),
- Переменные окружения (например, OMP_NUM_THREADS).

В данной лабораторной работе использована конструкции:

- для создания потоков (директива parallel),
- для распределения работы между потоками (директива for).

Директивой for предусмотрена барьерная синхронизация. Иначе говоря, достигнув конца региона, все потоки блокируются до тех пор, пока последний поток не завершит свою работу.

В нашем случае не требуется синхронизация потоков при записи значений в память, так как потоки не конфликтуют с адресами записываемой памяти и не используют общие переменные.

Закон Амдала

Закон Амдала - иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей. «В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента». Согласно этому закону, ускорение выполнение программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.

Математическое выражение

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля α от общего объема вычислений может быть получена только

последовательными расчётами, а, доля $1-\alpha$ может быть распараллелина идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов p). Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Описание и реализация алгоритмов

Базовый алгоритм умножения

Для вычисления произведения двух матриц A и B каждая строка матрицы A умножается на каждый столбец матрицы B. Затем сумма данных произведений записывается в соответствующую ячейку результирующей матрицы.

Пример умножения двух матриц:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} * \begin{bmatrix} A & B \\ C & D \\ E & F \end{bmatrix} = \begin{bmatrix} a*A+b*C+c*E & a*B+b*D+c*F \\ d*A+e*C+f*E & d*B+e*D+f*F \end{bmatrix}$$

Листинг. Базовый алгоритм умножения матриц (последовательный).

```
void Matrix::ClassicMultiplication(Matrix &m1, Matrix &m2, Matrix &result) {
	for (int i = 0; i < m1.col; ++i) {
		for (int j = 0; j < m2.line; ++j) {
			result.matr[i][j] = 0;
			for (int k = 0; k < m1.line; ++k)
				result.matr[i][j] = result.matr[i][j] + m1.matr[i][k] * m2.matr[k][j];
		}
	}
	return;
}
```

Листинг. Базовый алгоритм умножения (параллельный).

```
void Matrix::ClassicMultiplication(Matrix &m1, Matrix &m2, Matrix &result) {
	#if PARALLELS
	#pragma omp parallel for
	#endif
	for (int i = 0; i < m1.col; ++i) {
		for (int j = 0; j < m2.line; ++j) {
			result.matr[i][j] = 0;
			for (int k = 0; k < m1.line; ++k)
				result.matr[i][j] = result.matr[i][j] + m1.matr[i][k] * m2.matr[k][j];
		}
	}
	return;
}
```

Алгоритм Винограда

Алгоритм Винограда считается более эффективным благодаря сокращению количества операций умножения. Результат умножения двух матриц представляет собой скалярное произведение соответствующих строки и столбца. Можно заметить, что такое умножение позволяет выполнить заранее часть работы:

$U = (u_1, u_2, u_3, u_4)$

$V = (v_1, v_2, v_3, v_4)$

$U * V = u_1 * v_1 + u_2 * v_2 + u_3 * v_3 + u_4 * v_4$

Это равенство можно переписать в виде:

$U * V = (u_1 + v_2) * (v_1 + u_2) + (u_3 * v_4) * (u_3 + v_4) - u_1 * u_2 - u_3 * u_4 - v_1 * v_2 - v_3 * v_4$

При этом значения $u_1 * u_2$, $u_3 * u_4$, $v_1 * v_2$, $v_3 * v_4$ можно рассчитать заранее.

Однако под массивы данных коэффициентом требуется дополнительная память. В случае нечетного количества столбцов первой матрицы требуются дополнительные вычисления.

Листинг. Модифицированный алгоритм Винограда (последовательный).

```
void Matrix::VinogradImprovedMultiplication(Matrix &m1, Matrix &m2, Matrix &result) {
	int d = m1.line / 2;
	bool flag = m1.line % 2 == 1;

	// Вычисление rowFactors для m1
	for (int i = 0; i < m1.col; ++i) {
		m1.rowFactor[i] = m1.matr[i][0] * m1.matr[i][1];
		for (int j = 1; j < d; ++j)
			m1.rowFactor[i] += m1.matr[i][2 * j] * m1.matr[i][2 * j + 1];
	}

	// Вычисление columnFactor для m2
	for (int i = 0; i < m2.line; ++i) {
		m2.columnFactor[i] = m2.matr[0][i] * m2.matr[1][i];
		for (int j = 1; j < d; ++j)
			m2.columnFactor[i] += m2.matr[2 * j][i] * m2.matr[2 * j + 1][i];
	}

	// Вычисление матрицы result
	for (int i = 0; i < m1.col; ++i) {
		double buf = 0.0;
		for (int j = 0; j < m2.line; ++j) {
			buf = (flag ? m1.matr[i][m1.line - 1] * m2.matr[m2.line - 1][j] : 0);
			buf -= m1.rowFactor[i] + m2.columnFactor[j];

			for (int k = 0; k < d; ++k)
				buf += (m1.matr[i][2 * k] + m2.matr[2 * k + 1][j]) *
					(m1.matr[i][2 * k + 1] + m2.matr[2 * k][j]);

			result.matr[i][j] = buf;
		}
	}

	return;
}
```

Листинг. Модифицированный алгоритм Винограда (параллельный).

```
void Matrix::VinogradImprovedMultiplication(Matrix &m1, Matrix &m2, Matrix &result) {
	int d = m1.line / 2;
	bool flag = m1.line % 2 == 1;

	// Вычисление rowFactors для m1
	#if PARALLELS
	#pragma omp parallel for
	#endif
	for (int i = 0; i < m1.col; ++i) {
		m1.rowFactor[i] = m1.matr[i][0] * m1.matr[i][1];
		for (int j = 1; j < d; ++j)
			m1.rowFactor[i] += m1.matr[i][2 * j] * m1.matr[i][2 * j + 1];
	}

	// Вычисление columnFactor для m2
	#if PARALLELS
	#pragma omp parallel for
	#endif
	for (int i = 0; i < m2.line; ++i) {
		m2.columnFactor[i] = m2.matr[0][i] * m2.matr[1][i];
		for (int j = 1; j < d; ++j)
			m2.columnFactor[i] += m2.matr[2 * j][i] * m2.matr[2 * j + 1][i];
	}

	// Вычисление матрицы result
	#if PARALLELS
	#pragma omp parallel for
	#endif
	for (int i = 0; i < m1.col; ++i) {
		double buf = 0.0;
		for (int j = 0; j < m2.line; ++j) {
			buf = (flag ? m1.matr[i][m1.line - 1] * m2.matr[m2.line - 1][j] : 0);
			buf -= m1.rowFactor[i] + m2.columnFactor[j];

			for (int k = 0; k < d; ++k)
				buf += (m1.matr[i][2 * k] + m2.matr[2 * k + 1][j]) *
				    (m1.matr[i][2 * k + 1] + m2.matr[2 * k][j]);

			result.matr[i][j] = buf;
		}
	}
	return;
}
```

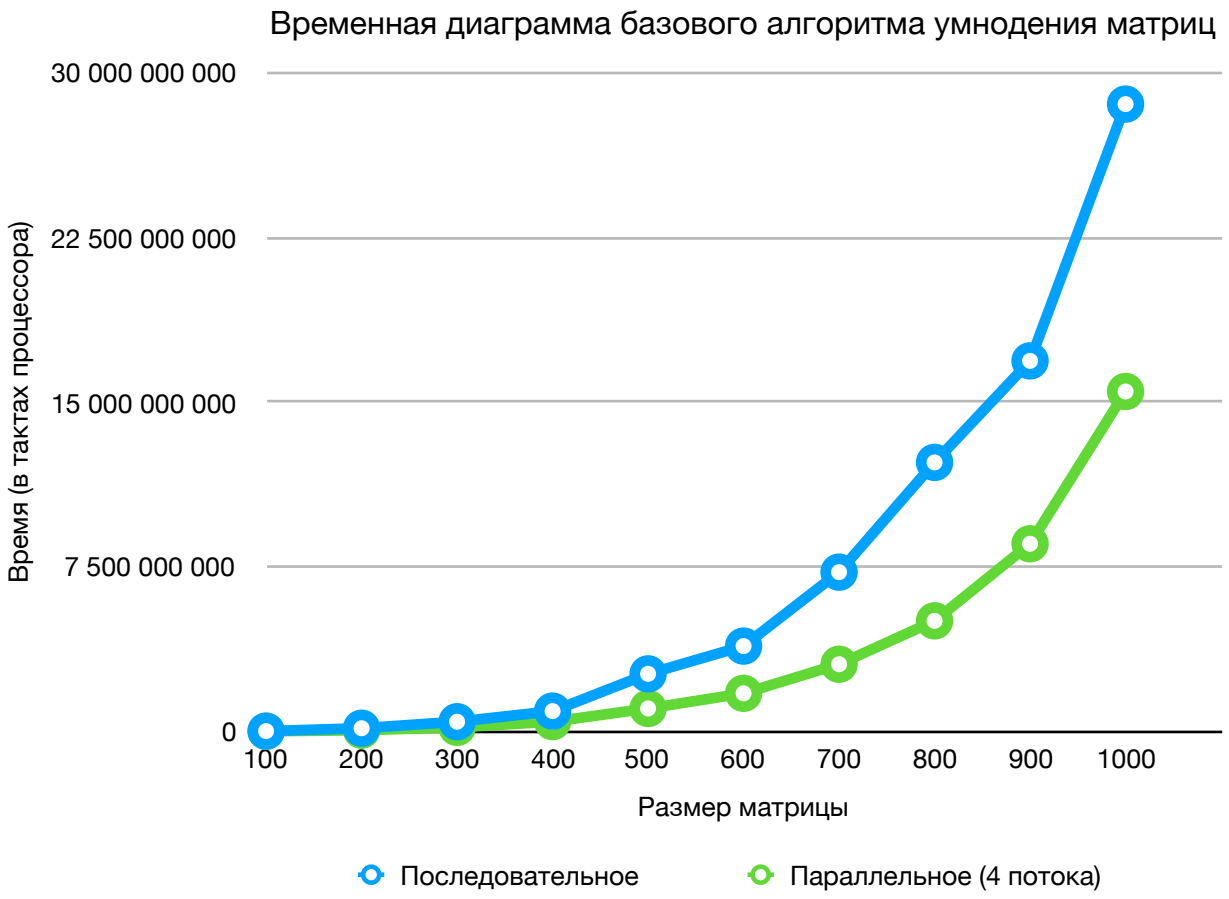
Тесты

Для сравнения алгоритмов были проведены серии тестов для последовательного и параллельного умножения.

Базовый алгоритм умножения матриц.

Размер матрицы	Последовательное	Параллельное
100x100	16 100 159	7 320 470
200x200	147 219 581	56 906 702
300x300	444 158 507	186 311 943

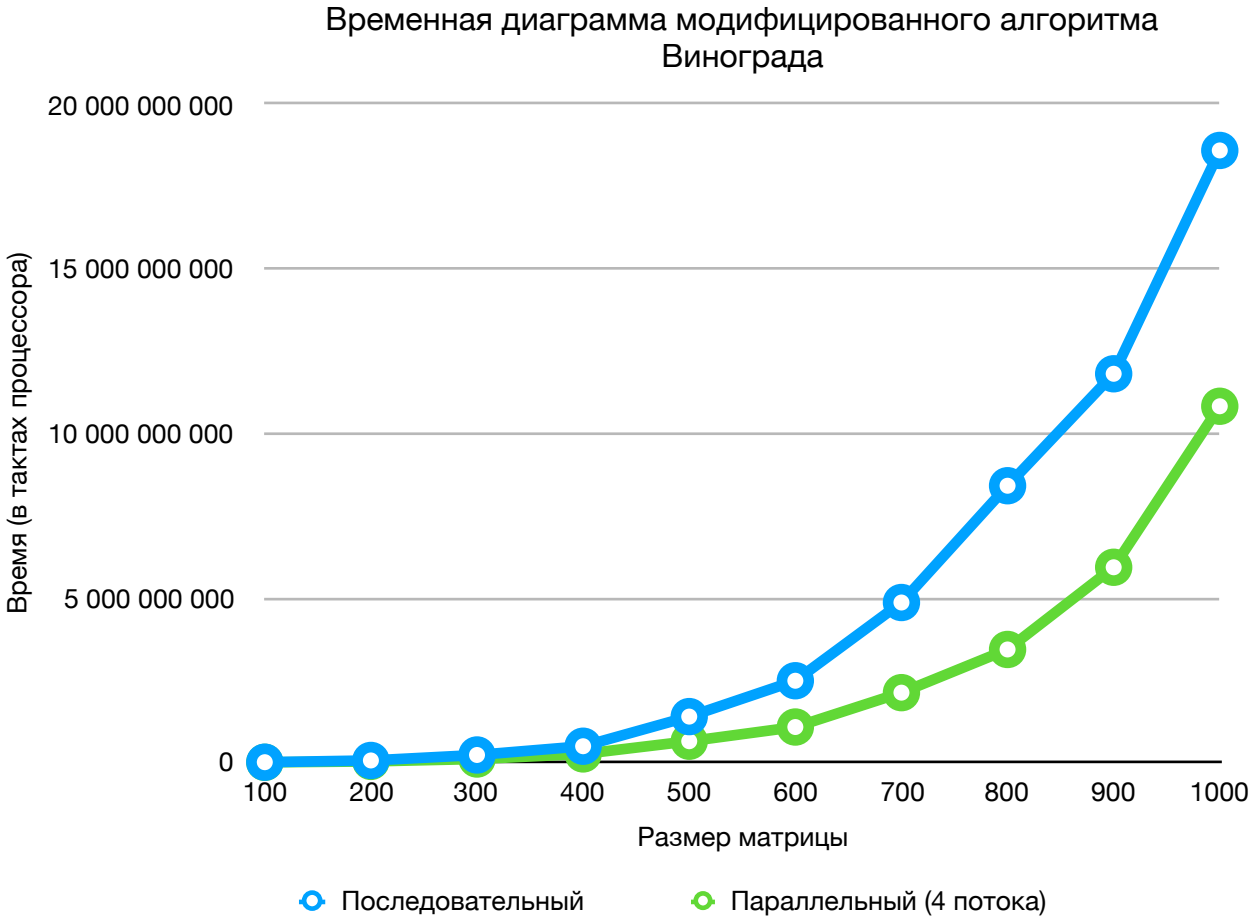
400x400	926 736 499	448 769 361
500x500	2 624 256 914	1 048 628 292
600x600	3 889 751 416	1 738 245 248
700x700	7 258 480 045	3 061 939 512
800x800	12 257 558 989	5 039 678 330
900x900	16 882 578 924	8 550 761 614
1000x1000	28 579 981 384	15 487 558 375



Модифицированный алгоритм умножения Винограда

Размер мартрицы	Последовательный	Параллельный
100x100	8 743 605	3 838 641
200x200	61 495 156	32 478 818
300x300	228 034 987	105 406 962
400x400	493 645 635	260 014 880
500x500	1 393 580 714	645 506 791
600x600	2 482 419 636	1 077 002 875

700x700	4 858 299 281	2 120 623 519
800x800	8 404 661 759	3 432 324 874
900x900	11 804 895 752	5 928 659 968
1000x1000	18 587 474 115	10 818 371 715



Вывод

Исходя из результатов тестов, а также на основе теории можно сделать вывод, что использование многопоточности позволяет сильно увеличить работу алгоритма. Прирост производительности при распараллеливании задачи ограничивается законом Амдала. При этом, как и следовало ожидать, параллельный алгоритм Винограда выигрывает у параллельного стандартного алгоритма.

Заключение

В результате лабораторной работы были изучены и реализованы версии алгоритмов умножения матриц как последовательно, так и параллельно, а именно:

- Стандартный алгоритм умножения (последовательный),
- Стандартный алгоритм умножения (параллельный),
- Модифицированный алгоритм Винограда (последовательный),
- Модифицированный алгоритм Винограда (параллельный).

Также был проведен временной анализ эффективности данных алгоритмов, построены графики для наглядности.