Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

Название: Параллельное программирование

Дисциплина: Анализ алгоритмов

Студент	ИУ7-53Б		И. О. Артемьев
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

В настоящее время компьютерные системы оперируют большими объемами данных. Над этими данными проводится большой объем различного рода вычислений. Для того, чтобы они выполнялись быстрее, было придумано параллельное программирование.

Его суть заключается в том, чтобы относительно равномерно разделять нагрузку между потоками ядра. Каждое из ядер процессора может обрабатывать по одному потоку, поэтому когда количество потоков на ядро становится больше, происходит квантование времени. Это означает, что на каждый процесс выделяется фиксированная величина времени (квант), после чего в течение кванта обрабатывается следующий процесс. Таким образом создается видимость параллельности. Тем не менее, данная оптимизация может сильно ускорить вычисления.

Целью данной лабораторной работы является получение навыков параллельного программирования. Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучить последовательный и параллельный варианты выбранного алгоритма;
- составить схемы данных алгоритмов;
- реализовать разработанные версии алгоритма;
- провести сравнительный анализ реализаций по затрачиваемым ресурсам (время и память);
- описать и обосновать полученные результаты.

1. Аналитический раздел

В данном разделе будут представлены описания последовательного и параллельного вариантов алгоритма сложения матриц.

1.1 Алгоритм нахождения суммы двух матриц

Суммой матриц A и B одинаковой размерности называется матрица той же размерности, обозначаемая A + B, каждый элемент которой равен сумме соответственных элементов матриц A и B, т. е. если A = $(a_{ij})_{m\times n}$ и B = $(b_{ij})_{m\times n}$, то A + B = $(a_{ij} + b_{ij})_{m\times n}$, где i = 1, 2, ..., m; j = 1, 2, ..., n

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \tag{1.1}$$

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \tag{1.2}$$

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{pmatrix} \tag{1.3}$$

1.2 Параллельный алгоритм нахождения суммы двух матриц

Поскольку можно обрабатывать строки матриц раздельно, то логично взять и разбить вычисление суммы определенных строк на потоки, нужно будет лишь правильно определить какой поток за какие строки отвечает.

1.3 Вывод

В данном разделе были рассмотренны принципы работы последовательного и параллельного алгоритмов нахождения суммы матриц. Полученных знаний достаточно для разработки выбранных алгоритмов.

На вход алгоритмам будут подаваться результирующая, левая и правая матрицы, а также их размерности.

Реализуемое ПО будет работать в пользовательском режиме (вывод результата вычисления суммы двух матриц), а также в экспериментальном (проведение замеров времени выполнения алгоритмов).

2. Конструкторский раздел

В данном разделе будут спроектированы схемы алгоритмов, описаны используемые типы данных, а также произведена оценка памяти и описана структура ПО.

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.3 представлены схемы рассматриваемых алгоритмов.

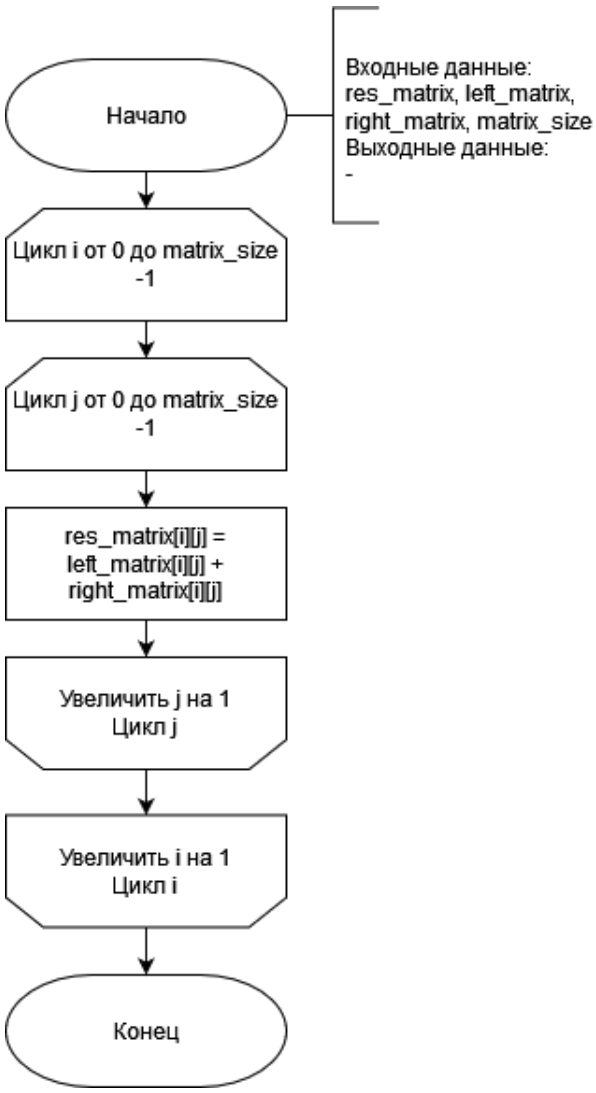

Рисунок 2.1 – Схема алгоритма последовательного вычисления суммы двух матриц

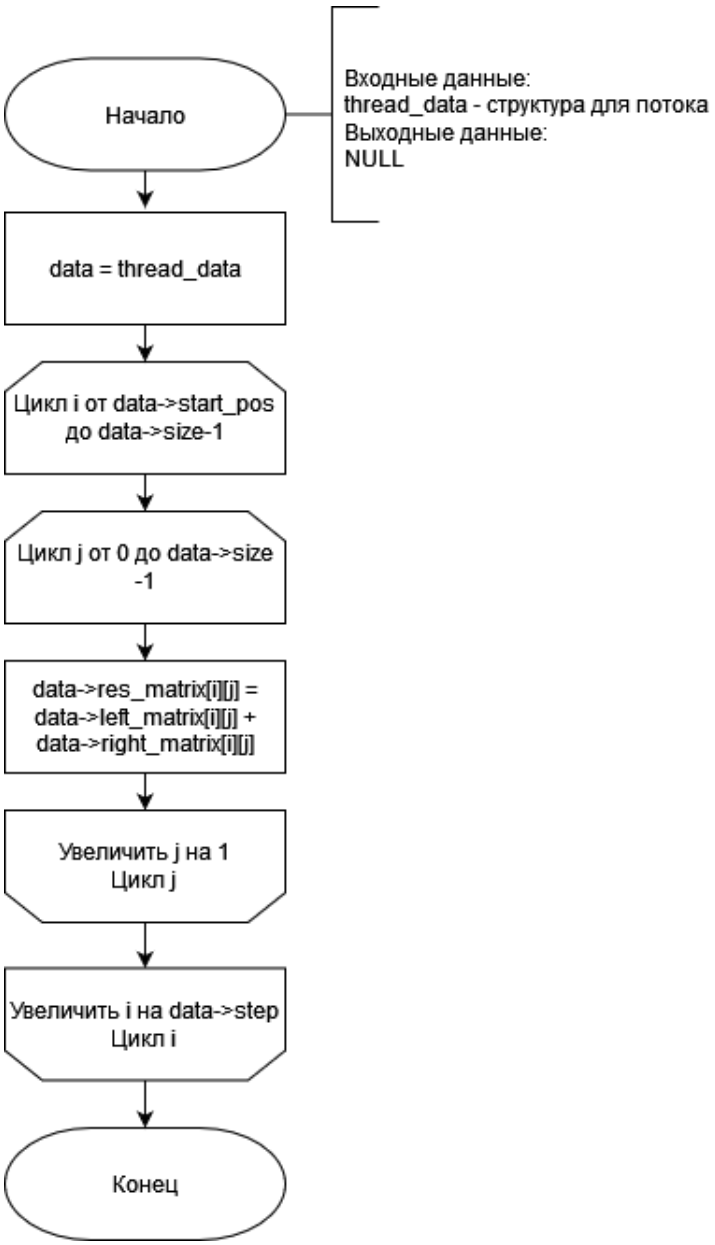

Рисунок 2.2 – Схема алгоритма вычисления суммы определенных строк матриц для параллельного выполнения операций

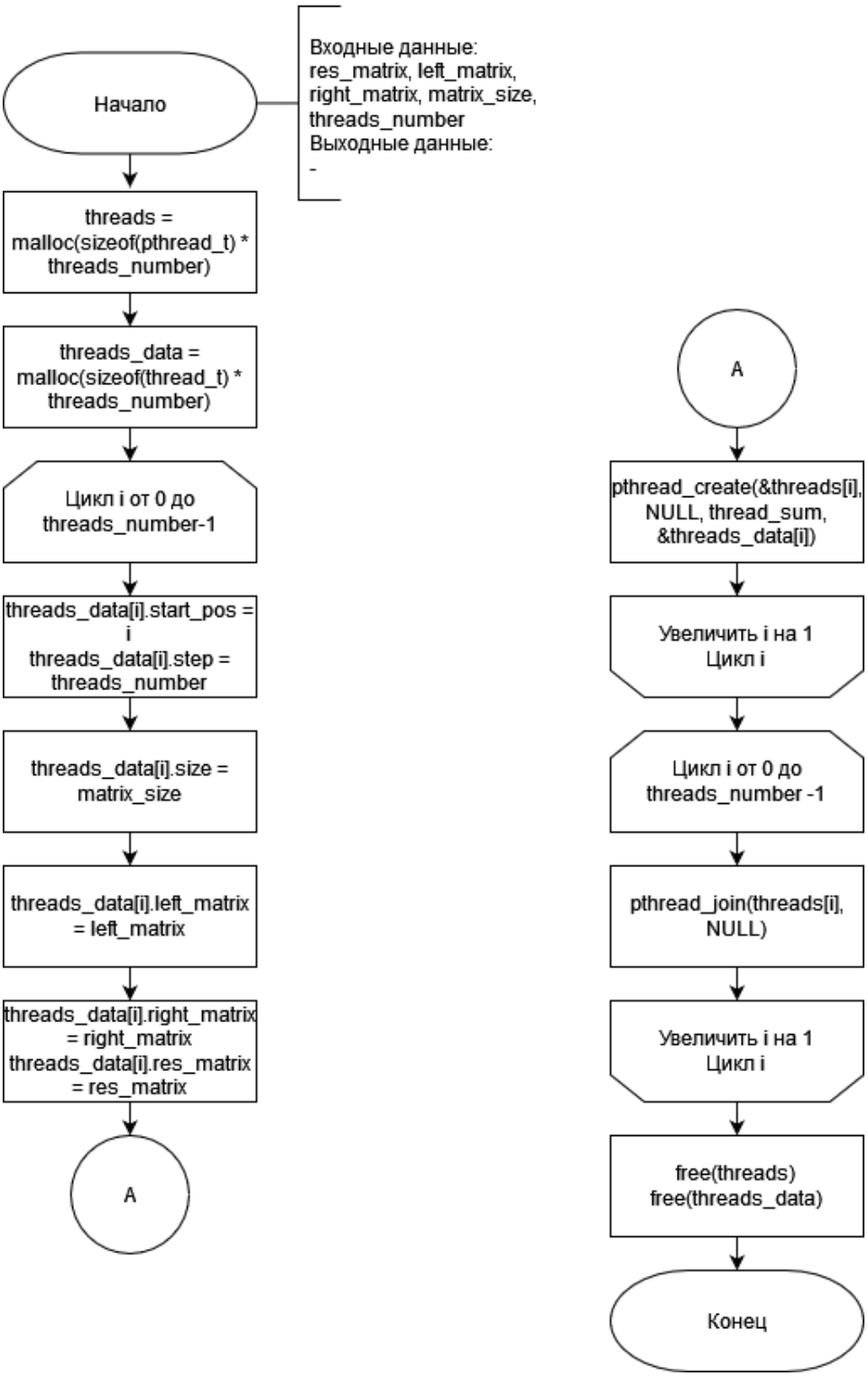

Рисунок 2.3 – Схема алгоритма параллельного вычисления суммы двух матриц

2.2 Используемые типы данных

При реализации алгоритмов будут использованы следующие структуры данных:

- потоки - массив типа pthread_t;
- данные потоков - массив типа thread_t.

Листинг 2.1 – Тип данных потока

```
typedef struct
{
    size_t start_pos;
    size_t step;

    size_t size;
    int **left_matrix;
    int **right_matrix;
    int **res_matrix;
} thread_t;
```

2.3 Оценка памяти

Рассмотрим затрачиваемый объем памяти для рассмотренных алгоритмов.

При последовательном вычислении память используется на:

- входные параметры функции - sizeof(int) + 3 * sizeof(int **);
- вспомогательные переменные - sizeof(size_t);

В случае параллельного вычисления для каждого потока помимо описанной выше используемой памяти выделяется структура thread_t, размер которой равен 3 * sizeof(size_t) + 3 * sizeof(int **). Также внутри функции, создающей потоки выделяется память:

- массив потоков - threads_number * sizeof($pthread\_t$);
- вспомогательные переменные - 2 * sizeof(size_t).

2.4 Структура ПО

ПО будет состоять из следующих модулей:

- главный модуль - из него будет осуществляться запуск программы и выбор соответствующего режима работы;
- модуль интерфейса - в нем будет описана реализация режимов работы программы;
- модуль, содержащий реализации алгоритмов.

2.5 Вывод

На основе полученных в аналитическом разделе знаний об алгоритмах были спроектированы схемы алгоритмов, выбраны используемые типы данных, проведена оценка затрачиваемого объема памяти, а также описана структура ПО.

3. Технологический раздел

В данном разделе будут приведены требования к ПО, средства его реализации и листинга кода алгоритмов, а также рассмотрены тестовые случаи.

3.1 Средства реализации

Для реализации программ я выбрал язык программирования C, так как этот язык наиболее точно покажет разницу по времени, а еще я очень хорошо знаком с этим языком и пишу на нем давно.

3.2 Листинги кода

Листинги 1 - 3 демонстрируют реализацию алгоритмов.

Листинг 3.1 – Алгоритм последовательного вычисления значения определенного интеграла

```
void default_sum(int **const res_matrix, int **const left_matrix,
int **const right_matrix, const int matrix_size)
{
    for (size_t i = 0; i < matrix_size; ++i)
    {
        for (size_t j = 0; j < matrix_size; ++j)
        {
            res_matrix[i][j] = left_matrix[i][j] + right_matrix[i][j];
        }
    }
}
```

Листинг 3.2 – Алгоритм вычисления значения определенного интеграла

```
void *thread_sum(void *thread_data)
{
    thread_t *data = (thread_t *)thread_data;

    for (size_t i = data->start_pos; i < data->size; i += data->step)
    {
        for (size_t j = 0; j < data->size; ++j)
        {
```

```
            data->res_matrix[i][j] = data->left_matrix[i][j] +
                                     data->right_matrix[i][j];
        }
    }

    return NULL;
}
```

Листинг 3.3 – Алгоритм параллельного вычисления значения определенного интеграла

```
void threads_matrix_sum(int **const res_matrix, int **const left_matrix,
int **const right_matrix, const int matrix_size, const int threads_number)
{
    pthread_t *threads = (pthread_t *)malloc(sizeof(pthread_t) *
    threads_number);
    thread_t *threads_data = (thread_t *)malloc(sizeof(thread_t) *
    threads_number);

    for (size_t i = 0; i < threads_number; ++i)
    {
        threads_data[i].start_pos = i;
        threads_data[i].step = threads_number;

        threads_data[i].size = matrix_size;
        threads_data[i].left_matrix = left_matrix;
        threads_data[i].right_matrix = right_matrix;
        threads_data[i].res_matrix = res_matrix;

        pthread_create(&threads[i], NULL, thread_sum, &threads_data[i]);
    }

    for (size_t i = 0; i < threads_number; ++i)
        pthread_join(threads[i], NULL);

    free(threads);
    free(threads_data);
}
```

3.3 Тестирование ПО

В таблице 3.1 приведены тестовые случаи для алгоритма сложения матриц.

Таблица 3.1 – Тестовые случаи

N	Матрица 1	Матрица 2	Результат
1	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$
2	(3)	(3)	(6)
3	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 8 & 10 & 12 \\ 8 & 10 & 12 \\ 8 & 10 & 12 \end{pmatrix}$

3.4 Вывод

На основе схем из конструкторского раздела были написаны реализации требуемых алгоритмов, а также проведено их тестирование.

4. Исследовательский раздел

В данном разделе будет проведен сравнительный анализ алгоритмов по времени и затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее.

- Операционная система: macOS Catalina версия 10.15.6;
- Память: 8 GB 1600 MHz DDR3;
- Процессор: 1,8 GHz 2-ядерный процессор Intel Core i5.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола.

Замеры времени осуществлялись с помощью функция clock_gettime из time.h.

4.2 Оценка времени работы алгоритмов

В таблице 4.1 представлены замеры времени (в нс) работы алгоритмов на размерностях квадратных матриц от 1000 до 3000. Каждое значение было получено усреднением по 100 замерам. Максимальное количество запускаемых потоков было выбрано 8, так как процессор содержит 2 логических ядра.

Таблица 4.1 – Временные замеры работы алгоритмов

Размер	Последоват.	1 п.	2 п.	4 п.	8 п.
1000	5718559	7244901	4822434	3364424	3474179
1500	14189482	16170494	10369529	7251459	8184497
2000	22699222	26970522	16873255	11781718	13092998
2500	35500930	43180334	25758164	19568871	19866884
3000	52606071	61008999	37990068	26939517	26126144

4.3 Вывод

По результатам исследования можно сделать вывод, что самым быстрым для процессора является использование 4 потоков. Если количество потоков увеличить до 8 время увеличивается, так как за квант процессорного времени процессы не успевают обработаться, из-за чего увеличивается объем "распаковок"и "запаковок"используемых процессором данных.

Заключение

В ходе выполнения данной лабораторной работы были выполнены следующие задачи:

- изучены последовательный и параллельный варианты алгоритма нахождения суммы двух матриц;
- составлены схемы данных алгоритмов;
- реализованны разработанные версии алгоритмов;
- проведен сравнительный анализ алгоритмов по затрачиваемым ресурсам (время и память);
- описаны и обоснованы полученные результаты.

В результате исследований можно прийти к выводу, что использование параллельного алгоритма нахождения суммы двух матриц по времени является в 2 раза оптимальнее последовательного (при использовании 4 потоков). По памяти же многопоточная реализация оказывается более затратной, так как выделяется память под массив указателей на потоки и массив с данными для каждого потока отдельно.

Список литературы

1. Язык программирования Си [Электронный ресурс] Режим доступа: https://ikt.muctr.ru/images/info/Brian_Kernighan_Dennis_Ritchie-The_C_Programming_Language-RU.pdf, (дата обращения: 10.11.2021)

2. clock_gettime [Электронный ресурс] Режим доступа: http://all-ht.ru/inf/prog/c/func/clock_gettime.html, (дата обращения: 10.11.2021)

3. Сложение матриц [Электронный ресурс] Режим доступа: https://zaochnik.com/spravochnik/matematika/matritsy/slozhenie-i-vychitanie/, (дата обращения: 10.11.2021)

4. Центральный процессор [Электронный ресурс] Режим доступа: https://mediapure.ru/matchast/chto-takoe-centralnyj-processor/, (дата обращения: 10.11.2021)

5. Потоки и работа с ними [Электронный ресурс] Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/standard/threading/threads-and-threading, (дата обращения: 10.11.2021)