МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Параллельные алгоритмы»

Тема: Основы работы с процессами и потоками

Студент гр. 0303 _____ Скиба В.В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург

2023

Цель работы.

Исследовать работу с потоками и процессами — механизмами операционной системы. Выявить закономерности, связанные с числом исполнителей, размером данных и времени исполнения.

Постановка задачи.

Лабораторная работа состоит из 3-х подзадач, которые решают одинаковую задачу с использованием процессов или потоков:

a) Выполнить умножение 2-х матриц.

b) Входные матрицы вводятся из файла (или генерируются).

c) Результат записывается в файл.

Описание подзадач:

1.1 Выполнить задачу, разбив её на 3 процесса. Выбрать механизм обмена данными между процессами.

Процесс 1: заполняет данными входные матрицы (читает из файла или генерирует их некоторым образом).

Процесс 2: выполняет перемножение матриц

Процесс 3: выводит результат в файл.

1.2.1 Аналогично 1.1, используя потоки (std::thread)

1.2.2 Разбить умножение на P потоков.

Выполнение задач.

Используемые инструменты:

- Язык программирования Rust
- Дополнительные зависимости (Cargo.toml):
  - rand — для генерации случайных чисел и заполнения входных матриц
  - nix — для более удобной обертки над системными вызовами в ОС Linux
  - fork — для более удобной обертки над системным вызовом fork
- ОС Linux

Общая структура программы:

Проект состоит из трех бинарных целей и файла библиотеки, общего для всех.

Файл библиотеки содержит следующие функции и типы:

1. `pub type Mat = Vec<Vec<i32>>;`

Определение типа Mat как Vec<Vec<i32>> для хранения содержимого матрицы

2. `Mat::save_to_file(&self, name: &'static str)`

Метод для типа Mat, функция третьего исполнителя, которая принимает название файла и сохраняет содержимое матрицы в текстовом человеко-читаемом формате

3. `generate_random_matrix(rows: usize, cols: usize) -> Mat`

Вспомогательная функция для создания матрицы, заполненной случайными числами от 0 до 10 заданного размера.

4. `input_matrices() -> (Mat, Mat)`

Функция первого исполнителя, которая запрашивает данные о размере матриц через стандартный подок ввода и возвращает кортеж из двух заполненных случайным образом матриц (используя функцию выше).

5. `multiply_matrices((mat1, mat2): (Mat, Mat)) -> Option<Mat>`

Функция второго исполнителя, которая принимает кортеж из двух матриц, которые необходимо перемножить, и возвращает либо ничего в случае не поддерживаемых размеров матрицы, либо готовую перемноженную матрицу.

6. `start_measure(cell: &'static OnceLock<Instant>)`
`stop_measure(cell: &'static OnceLock<Instant>) -> std::time::Duration`

Вспомогательные функции для начала и завершения изменения одной метрики времени (работает один раз за исполнение программы из-за типа OnceLock).

Во всех трех вариантах программы общая схема исполнения одинаковая, а из-за зависимости каждого этапа от предыдущего нет необходимости в

использовании примитивов синхронизации для входных/выходных данных каждого этапа.

Второй исполнитель запускается только после получения результатов первым исполнителем, третий исполнитель запускается после получения результатов вторым исполнителем.

Рассмотрим механизм создания нового потока/процесса:

1. Процессы

Для создания нового процесса используется системный вызов fork() операционной системы Linux. После его завершения, далее код начинает исполняться в двух процессах, и в двух разных ветках результата fork (он возвращает enum с вариантом родителя либо дочернего процесса). Если это родитель, то мы дожидаемся исполнения дочернего процесса системным вызовом wait(). Если это дочерний процесс, то мы выполняем код соответствующего исполнителя.

2. Потоки

Для создания нового потока используется функция стандартной библиотеки std::thread::spawn(), которая на вход принимает функцию, которая начнет свое исполнение в отдельном потоке. За созданием нового потока сразу следует join, так как этапы в данной работе строго зависят друг от друга, а также потому что если не дождаться исполнения второго исполнителя с основного потока, то его выполнение прервется (abort) в отличие от варианта с fork().

Распараллеливание умножения:

Для выполнения умножения матрицы в нескольких потоках, элементы результирующей матрицы рассматриваются как отдельные независимые задачи которые нужно решить. Следовательно, максимальное число параллельных потоков равно количеству элементов результирующей матрицы.

Первым этапом после запроса у пользователя числа исполнителей, считается сколько элементов матрицы каждый исполнитель будет вычислять. Затем для каждого такого диапазона элементов запускается отдельный поток, который вычисляет нужный диапазон элементов и получает вектор из значений этого диапазона. Вместе с индексом диапазона, этот вектор отправляется обратно в основной поток при помощи канала mpsc с примитивами синхронизации внутри, который идеально подходит для решения данной задачи. Основной поток после запуска всех исполнителей в отдельных потоках начинает синхронно принимать сообщения с кусками результирующей матрицы, пока он полностью ее не заполнит. Когда все исполнители закончили работу, канал закрылся и цикл приема сообщений завершился.

Примитив синхронизации использовался для того, чтобы избежать сложностей компиляции кода с использованием одного объекта разными потоками. Некоторые библиотеки уже решают эту проблему разделения вектора на части и предоставляют safe обертку для того чтобы избежать необходимости использования примитивов синхронизации.

Полученные результаты.

Замер времени, умножение матриц NxN на NxN, процессы

N	**400**	**500**	**600**	**700**	**800**
Запись входных матриц, мс	400	618	900	1215	1597
Умножение, мс	92	180	305	491	732
Запись результата, мс	202	312	453	613	812

Замер времени, умножение матриц NxN на NxN, потоки

N	**400**	**500**	**600**	**700**	**800**
Запись входных матриц, мс	397	634	895	1215	1605
Умножение, мс	94	182	306	492	735
Запись результата, мс	207	316	465	636	835

Замер времени, умножение матриц 400x400 на 400x400, потоки с распараллениванием умножения на P исполнителей

P	**1**	**2**	**4**	**8**
Умножение, мс	50	27	15	15

Замер времени, умножение матриц 800x800 на 800x800, потоки с распараллениванием умножения на P исполнителей

P	**1**	**2**	**4**	**8**
Умножение, мс	544	280	144	133

Из замеров времени выполнения операций, можно сделать вывод что разница между использованием процессов и потоков несущественная, однако в данной задаче они не использовались как положено: код хоть и исполнялся отдельными потоками/процессами, но все еще делал это синхронно.

Сравнивая реализацию с параллельным умножением и одним исполнителем с однопоточной реализацией наблюдается различие почти в 2 раза несмотря на несущественное различие в реализации алгоритма.

При увеличении числа параллельных исполнителей, скорость работы алгоритма близка к прямой зависимости от количества исполнителей, но после 4х время почти перестает уменьшаться. Это связано с тем, что на машине, на

которой исполняется программа установлен процессор с 4мя ядрами и 8 потоками. Небольшой прирост наблюдается на 8 исполнителях и размером 800x800.

Выводы.

В данной лабораторной работе была установлена практическим путем закономерность между количеством исполнителей и временем работы алгоритма. Также, была изучена разница между использованием потоков и процессов.