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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Параллельные алгоритмы»

ТЕМА: ОСНОВЫ РАБОТЫ С ПРОЦЕССАМИ И ПОТОКАМИ

Студент гр. 0303 Мыратгелдиев А. М.

Преподаватель Сергеева Е. И.

Санкт-Петербург
2023

Цель работы.

Изучить механизм работы процессов и потоков на примере языка C++.

Задание.

Лабораторная состоит из 3х подзадач, которые выполняют одинаковую задачу с использованием процессов или потоков.

Выполнить умножение 2х матриц.

Входные матрицы вводятся из файла (или генерируются).

Результат записывается в файл.

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 умножение

Процесс 3: выводит результат

1.2.1

Аналогично 1.1, используя потоки (std::threads)

1.2.2

Разбить умножение на P потоков (“наивным” способом по по строкам-столбцам).

Основные теоретические положения.

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут файлы, разделяемый сегмент памяти и многое другое.

Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать.

Выполнение работы.

Выполнение пункта 1.1:

В первую очередь была написана функция *generateTxtFile*, которая генерирует матрицы и записывает их в файл “matrices.txt” в основном потоке.

Чтобы процессы могли обмениваться между собой данными, была написана функция *createSharedMem*, которая создает разделяемый сегмент памяти при помощи функции std::*shmget* и возвращает ссылку нее (std::*shmat*). Размер этой памяти равен размеру матрицы, которая получается после умножения первых двух матриц.

Была реализована функция *runProcess*, которая вызывает переданную ей в качестве параметра функцию в отдельном процесс. Для каждого процесса была написана своя функция:

matrixPrepare – функция, которая читает матрицу из файла и записывает в общий сегмент памяти;

matrixCalculate – функция, которая умножает две матрицы из общего сегмента памяти и записывает туда же результат умножения;

matrixPrintToFile – функция, которая записывает результат умножения из общего сегмента памяти в файл “result1.txt”.

Выполнение пункта 1.2:

Была использована функция *generateTxtFile*, которая генерирует матрицы и записывает их в файл.

Затем, была написана функция *readMatricesFromFile*, которая читает матрицы и записывает их в переданные в качестве параметров матрицы

(двумерный вектор из *int,* которая передается по ссылке и обернут в *std::ref*). Данная функция запускается в потоке *t1*.

Далее, в новом потоке *t2* запускается функция *matrixMult*, которая вычисляет умножение двух матриц, переданных ей в качестве параметра (по ссылке и обернут в *std::ref*) и возвращает их через std::promise (который также был передан в качестве параметра функции).

В конце в новом потоке вызывается функция *printMatrixToFile*, которая выводит результат умножения в файл, название которого передан в качестве параметра.

Выполнение пункта 1.3:

Зависимость времени умножения от количчества потоков P представлена в табл. 1:

При запуске, программа ожидает ввода количества потоков и размерностей двух матриц. После вызывается функция *calculate*, которой передаются число потоков и размерности матриц.

Функция *calculate(thread_num, dim1, dim2, ms)* вызывает функцию из предыдущей программы для генерации матриц и записи их в файл (функция *generateTxtFile*), после, в отдельном потоке считывает эти матрицы в двумерные векторы и засекает время перед вызовом функции *parallelMult* (функция вызывает в новом потоке). Функция *parallelMult* принимает в качестве параметров *std::promise* (который был создан перед вызовом данной функции), две матрицы и количество потоков. Данная функция вычисляет умножение этих двух матриц используя *P* потоков и возвращает результат через переданный ей *std::promise*.

После выполнения функции *parallelMult*, вычисляется время выполнения этой функции, записывается в переменную *ms* переданной по ссылке в функцию *calculate* и результат умножения записывается в текстовый файл.

Ниже в таблице 1 представлена зависимость времени выполнения программы от размеров входных данных при фиксированном *P* количестве потоков:

Время, мс	Размер матриц	Количество потоков
744	10 x 10	3
2951	100 x 100	3
235880	500 x 500	3
2580828	1000 x 1000	3

Таблица 1. Зависимость времени выполнения от размеров матриц

Из полученных результатов можно сделать вывод, что при увеличении размера матриц также увеличивает и время выполнения программы (что очевидно).

Далее, зависимость времени выполнения от количества потоков при фиксированных размерах матриц (см. табл. 2):

Время, мс	Количество потоков	Размер матриц
6142	1	100 x 100
2093	5	100 x 100
2074	10	100 x 100
3103	20	100 x 100
2689	15	100 x 100
2425	13	100 x 100
2334	9	100 x 100
2383	11	100 x 100

Таблица 2. Зависимость времени выполнения от количества потоков

Из таблицы видно, что оптимальное количество потоков в нашем случае при размере матриц 100 x 100 равно 10. При дальнейшем увеличении количества потоков время выполнения программы увеличивается так как создание и запуск нового потока отнимает время.

Выводы.

При выполнении данной лабораторной работы, мы познакомились с механизмами работы процессов и потоков на языке C++. Было исследовано, что при увеличении числа потоков не всегда уменьшается время выполнения программы, так как после определенного числа потоков, время выполнения программы постепенно увеличивается из-за того, что создание потока - довольно затратный по времени процесс.