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. 0304 _____ Шквиря Е.В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург
2023

Цели работы.

На практике освоить методы работы с процессами и потоками. Провести анализ производительности обоих методов.

Постановка задачи.

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 строкам-столбцам).

Основные теоретические положения.

Поток – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Для работы с потоками используется библиотека `<thread>`. В ней располагается большое количество полезных функций для работы с потоками.

Для создания нового потока требуется создать объект класса `std::thread`. Конструктор принимает в себя функцию, которую будет поток исполнять, и список параметров для ее работы.

В C++ чтобы создать новый процесс используется функция

```
#include<unistd.h>
auto pid = fork();
```

Исходный процесс называется родительским. Созданный – дочерним. Родительский процесс получает на выход из функции `pid` нового процесса. Дочерний же процесс получает 0.

Дочерний процесс начинает свою работу со следующей строки, относительно расположения функции `fork()`.

В теории, т.к. создание нового процесса – это более затратная операция, чем создание нового потока, – многопоточное приложение должно выигрывать по производительности, относительного такого же приложения, но на процессах.

Также многопоточное приложение, где трудно вычислимая задача разбита на множество потоков, должна с увеличением количества потоков становиться быстрее. С одним нюансом: если количество потоков в приложении превысит количество физических потоков в компьютере, процессор начнет добавлять на физические потоки новые виртуальные. Это достижимо за счет переключения процессора между задачами в одном физическом потоке. Проблема в том, что на переключение процессору требуется некоторое время, которое становится весомо с большим количеством потоков.

Выполнение работы.

Перед переходом к замерам скорости работы разных подходов в разных ситуациях, был реализован алгоритм решения задачи с помощью разделения процессов и с помощью потоков (с одним или несколькими).

Для работы с матрицами (чтение, вычисление и запись) был создан файл Utils.h, в котором хранятся алгоритмы работы с технической частью ввода/вывода и вычисления итоговой матрицы. Также был добавлен метод log,

с помощью которого осуществляется логирование для лучшего понимания происходящего в процессе работы.

Для реализации алгоритма с помощью разделения процессов было необходимо выбрать механизм передачи данных между процессами. В данной работе был выбран механизм Shared Memory. Этот подход довольно прост для реализации в коде, но при этом имеет ограниченный запас вместимости, за счёт чего в него нельзя сохранить матрицы больших размеров. После этого был написал алгоритм, который считывает матрицу из файла и порождает дополнительный процесс для расчёта произведения матрицы, который в свою очередь порождает ещё один дополнительный процесс для сохранения итоговой матрицы в файл. Таким образом, по итогу будут образованы 3 процесса (на чтение, вычисление и запись результата), каждый из которых ждёт выполнения предыдущего. Как уже было упомянуто, данные между процессами передаются через Shared Memory.

Для выполнения алгоритма с помощью потоков было задействовано 3 потока (на чтение, на вычисление и на вывод). Так как потоки не создают новый процесс, то они могут работать с общей памятью приложения, за счёт чего не нужно было использовать дополнительные механизмы хранения данных. С помощью одного потока данные считались из файла, после матрицы были перемножены в другом потоке и записаны в файл в ещё одном потоке. Каждый последующий поток ждал завершения работы предыдущего.

Для выполнения алгоритма с разделением умножения матриц на несколько потоков – исходный алгоритм вычисления произведения был модифицирован. Каждому потоку для вычисления выделялось количество ячеек, равное $\frac{n}{P}$, где n –количество ячеек, а P – количество потоков. С помощью этого достигался баланс загруженности каждого потока и равномерное вычисление. Таким образом, задача решалась с использованием $2 + P$ потоков (1 поток на чтение, P потоков на вычисление и 1 поток на запись).

Каждый запуск приложения в конце оканчивался выводом времени (в микросекундах) работы программы от начала чтения данных из файла до окончания записи данных в файл. Время генерации матрицы и её первоначальной записи в файл для последующего чтения не учитывается.

Анализ

Перед переходом к вычислениям стоит упомянуть, что работа выполнялась на машине с количеством ядер процессора равным 6 и максимальным числом потоков равным 12.

Для дальнейшего анализа было рассчитано время выполнения задачи с помощью процессов, 1 потока на вычисления и 4 потоков на вычисления. За результат бралось среднее значение из 10 запусков программы в одинаковых условиях. Расчёт вёлся с квадратными матрицами размера 3, 7, 15 и 20. Результат приведён в таблице 1.

Таблица 1. Результаты запусков программы с разными размерами матрицы.

n	Процессы	1 Поток	4 Потока
3	1269	670	647
7	1292	729	713
15	1661	840	749
20	1274	766	751

Можно заметить, что при любых размерах матрицы вычисления в потоках было всегда быстрее. Это связано с тем, что создание процесса более трудоёмкая задача, чем создание потока или переключение между ними. Также по таблице видно, что разделение вычисления на несколько потоков уменьшило время, требуемое на получение конечного результата.

Далее было проанализировано время работы программы при разных размерах матриц и разном количестве потоков, выделяемых под вычисление произведения. За результат бралось среднее значение из 7 запусков программы

в одинаковых условиях. Расчёт вёлся с квадратными матрицами размера 20, 80, 200 и 400, причём каждая матрица обрабатывалась 1, 4, 10, 12, 14, 20, 40 или 80 потоками. Результат приведён в таблице 2.

Таблица 2. Результаты запусков программы с разными размерами матрицы (n) и количествами потоков (P).

P/n	20	80	200	400
1	1229	5543	61234	424918
4	1302	4780	30137	140285
10	1185	4412	26775	119473
12	1106	4229	24462	103869
14	1561	4426	25076	109804
20	1688	4615	26936	112108
40	1940	5269	27188	112550
80	3328	7188	29121	131210

Исходя из результатов, приведённых в таблице, можно сделать вывод, что при малых размерах матрицы куда быстрее работает программа с малым количеством потоков, выделенных на вычисления (так как не нужно тратить время на создание потоков и переключения процессора между ними). С другой стороны, когда размеры матрицы становятся большими – вычисление происходит быстрее при большом количестве потоков (так как происходит распараллеливание большого количества вычислений на некоторое число маленьких). Но в случае с превышением количества допустимых потоков вычисление матрицы происходит долго что при малых размерах, что при больших (так как происходит создание виртуальных потоков).

Таким образом, можно сделать вывод, что при разбиении решения задачи на несколько потоков – важно сохранить баланс между уменьшением времени работы за счёт распараллеливания вычислений и увеличением времени на поддержания этого распараллеливания.

Выводы.

В ходе выполнения лабораторной работы были разобраны методы работы с процессами и потоками с помощью возможностей C++. Была проанализирована скорость выполнения задачи программой с помощью процессов, одного потока и нескольких потоков.

Было выявлено, что выполнение задачи с помощью разделения процессов занимает больше времени, чем её разбиение на потоки. Также было выявлено, что время решения задачи с помощью нескольких потоков зависит от того, на какое количество распараллеливаются вычисления, и что при выборе большого количества потоков нужно думать о том, не пойдёт ли оно во вред производительности.