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 _____ Асташёнок М.С.

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 (генерация, вычисление и логирование) был создан файл MatrixUtils.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 генерирует матрицу заданного размера со случайными значениями в ячейках и порождает дополнительный процесс для расчёта произведения матрицы, который в свою очередь порождает ещё один дополнительный процесс для вывода итоговой матрицы в консоль. Таким образом, по итогу будут образованы 3 процесса (на генерацию, вычисление и логирование 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 генерацию, 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 были сгенерированы данные, после матрицы были перемножены в другом потоке и выведены в консоль 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 генерацию, P потоков на вычисление и 1 поток на вывод).

Каждый запуск программы в конце оканчивался выводом времени (в микросекундах) работы программы от начала генерации данных до окончания вывода данных в консоль.

Анализ

Перед переходом к вычислениям стоит упомянуть, что работа выполнялась на машине с количеством ядер процессора равным 2 и максимальным числом потоков равным 4.

Для дальнейшего анализа было рассчитано время выполнения задачи с помощью процессов, 1 потока на вычисления и 5 потоков на вычисления. За результат бралось среднее значение из 10 запусков программы в одинаковых условиях. Результат приведён в таблице 1.

Таблица 1. Результаты запусков программы с разными размерами матрицы.

n	**Процессы**	**1 Поток**	**3 Потока**
3	1157	431	492
9	1264	505	555
15	1329	607	646
21	1701	678	732

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.

Однако мы можем заметить, что увеличение количества потоков не уменьшило время исполнения умножения матриц. Это может быть связано со слишком малым размером матрицы, при котором разделение умножения на потоки только замедляет работу программы.

Далее было проанализировано время работы программы при разных размерах матриц и разном количестве потоков, выделяемых под вычисление произведения. За результат бралось среднее значение из 10 запусков программы в одинаковых условиях. 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	650	5151	54791	422629
5	868	4387	33495	256233
10	1098	3892	32009	238439
15	1322	3469	30285	234413
20	1446	4354	32784	233403
40	2457	4582	34363	232242
80	4077	6192	33499	235278

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 учитывать, что время на выделение новых потоков может замедлить скорость работы программы. Для того, чтобы этого не происходило, нужно поддерживать баланс между количеством потоков и объемом решаемой задачи.