**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 |  | Аристархов И.Е. |
| Преподаватель |  | Сергеева Е.И. |

Санкт-Петербург

2023

**Цель работы.**

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

**Поставленные 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.

Примечание: ссылки на объекты рекомендуется вставлять через std::ref(..).

Также данная библиотека позволяет получить количество физических потоков в компьютере:

std::thread::hardware\_concurrency()

Процесс – выполняющаяся программа и все её элементы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и так далее.

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.**

Для начала реализации задачи 1.1 требовалось определить способ передачи информации между процессами. Существует как минимум 3 подхода: shared memory, named Pipe и Unix Cache. Был выбран второй вариант, как наиболее простой для понимания и реализации. У данного варианта есть ограничения на размер передаваемой информации (не более 64 Кб) и на направление передачи (односторонний). Т.к. для анализа эти ограничения не существенны, было решено остановиться на этом варианте.

Дополнительным ограничением оказалось невозможность передавать и получать динамически аллоцированные данные. Поэтому все массивы, содержащие матрицы, статически аллоцированны. Размер матриц задается через макрос.

Кроме того, стояла задача замера времени. Для этого использовалась команда std::chrono::high\_resolution\_clock::now(), которая возвращала текущее время. Время замирялось в начале программы до создания каналов и в конце, когда последний процесс записал ответ в файл.

Остается рассмотреть саму структуру процессов. По задачи требовалось создать 3 процесса, один из которых генерирует 2 матрицы, другой их умножает, а третий выводит результат в файл. Т.к. для проведения замеров требуется получить момент после записи файла с ответом, было решено, что основной процесс и будет этим заниматься.

Передача была выстроена так, чтобы генератор передавал данные на умножение, а с умножения результат передавался на запись в файл. Т.е. потребовалось создать 2 pipe.

Далее была реализована задача 1.2. В ней процессы заменены потоками и убран весь код передачи данных, т.к. потоки имеют общую память. Также все функции для потоков дополнительно получают данные матриц, с которыми им работать.

Важно отметить, что если задача 1.2.1 еще может быть проанализирована в полуавтоматическом режиме значениями размера матрицы до 80, то 1.2.2 уже потребует полной автоматизации, т.к. выборка различных значений должна быть больше.

Поэтому было решено перейти от статического аллоцирования матриц к динамическому через двойные указатели (сперва использовался std::vector, но оказалось, что он сильно уступает по скорости статическому выделению, что не позволило получать ожидаемые результаты).

В 1.2.1 часть с умножением производилась строго в один поток, но 1.2.2 уже требовал выставления определенного числа потоков на эту операцию. Тем не менее количество потоков не должно было превышать количество ячеек в результирующей матрице, иначе некоторые потоки были бы бесполезны.

Задача умножения матрицы разбивалась на потоки за счет выделения потоку определенного промежутка, который этот поток должен рассчитать. Очевидно, что невозможно равномерно распределить эту задачу на любое число потоков. Простой пример – разбить 3 операции и 2 потока. Один поток заберет 1 операцию, а второму придется забрать все, что осталось. Именно так и действует распределение здесь.

Переключение между 1.2.1 и 1.2.2 реализовано через макрос.

**Анализ.**

Компьютер, на котором производились замеры, имел процессор 4 физическими ядрами и 8 потоками (по 2 потокам на ядро).

Все замеры производительности производились за счет запуска приложения 150 раз и сохранения минимального времени исполнения.

В первую очередь были сравнены задачи 1.1 и 1.2.1. В качестве изменяемого параметра был размер матрицы, используемый при расчетах.

Таблица 1. Результаты запусков программ с различными размерами матрицы.

|  |  |  |
| --- | --- | --- |
| Размер матрицы | Время исполнения, микросекунд | |
| Процессы (1.1) | Потоки (1.2.1) |
| 5 | 275 | 156 |
| 10 | 281 | 168 |
| 20 | 341 | 226 |
| 40 | 649 | 547 |
| 80 | 2439 | 2338 |

Из таблицы 1 видно, что потоки выигрывают по скорости во всех случаях.

Далее были произведены замеры времени работы задачи 1.2.1 при 2 разных размерах матрицы и 6 вариантов количества потоков для умножения.

Таблица 2. Результаты запусков программ с различным количеством потоков для умножения.

|  |  |  |
| --- | --- | --- |
| Количество потоков | Время исполнения, микросекунд | |
| Матрицы 40x40 | Матрицы 400x400 |
| 1 | 527 | 210758 |
| 2 | 539 | 210193 |
| 7 | 620 | 208407 |
| 15 | 783 | 205169 |
| 40 | 1282 |  |
| 400 |  | 77042 |
| 1600 | 43010 |  |
| 1000 |  | 92868 |

Сперва может показаться, что в таблице 2 нет закономерности. Тем не менее данная таблица показывает, что при выборе количества потоков требуется выдержать баланс между приростом производительности за счет одновременных вычислений и спадом производительности из-за частых переключений процессора.

В случае матрицы 40x40 умножения производились настолько быстро, что даже многопоточное умножение не ускоряло данный процесс. Точнее прирост производительности полностью перебивался затратами времени на переключения процессора.

При умножении матрицы 400x400 уже требуется гораздо больше времени и вычислительной мощности. Поэтому при увеличении количества потоков четко видно, что время работы уменьшается вплоть до значения 400, т.е. каждый поток берет на себя ровно по 400 умножений. Если же дать каждому потоку умножать по одной ячейке, то производительность станет ниже, но все еще лучше, чем если бы это делал один поток.

**Выводы.**

В результате выполнения лабораторной работы были разработаны две программы производящие умножение матриц. Одна программа использовала процессы и Named Pipe для передачи данных между процессами, а другая работала за счет многопоточности.

Были сравнены производительности обеих программ с целью выявить зависимость между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы.

Сперва было доказано, что работа с процессами медленнее, чем с потоками, т.к. на создание процесса требуется больше времени, чем на создание потока. Кроме того, было видно, что с увеличением размера матрицы обеим программам требовалось все больше и больше времени на расчеты.

Далее исследовалась зависимость времени работы приложения от количества потоков на операцию умножения матриц. Анализ показал, что прирост производительности от добавления очередного потока зависит от того, сможет ли прирост в скорости вычислений превысить дополнительное время на переключение процессора между виртуальными потоками. Точное количество требуемых потоков зависит от задачи и требует анализа для получения.