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 _____ Люлин Д.В.

Преподаватель _____ Сергеева Е.И.

Санкт-Петербург

2023

Цель работы.

Изучить основы работы с процессами и потоками.

Здание.

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).

В отчёте:

Исследовать зависимость между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы. Сформулировать ограничения.

Выполнение работы.

1. Умножение матриц с помощью 3 процессов.

В качестве средства межпроцессного взаимодействия были использованы сокеты UNIX. Благодаря данному средству процессы могут быть запущены независимо друг от друга. Для использования сокетов в стиле ООП был написан класс *Socket*:

```cpp
/// @brief RAII socket wrapper.
class Socket
{
public:
    explicit Socket( int sockfd );
    Socket( Socket&& other );
    Socket();
    virtual ~Socket();

    Socket& operator=( Socket&& rhs );

    Socket( const Socket& )            = delete;
    Socket& operator=( const Socket& ) = delete;

    bool is_valid() const;

    template < typename T >
    bool read( T* buffer, std::size_t nmemb )
    {
        std::size_t bytes_size = nmemb * sizeof( T );
        return bytes_size == ::read( sockfd_, buffer, bytes_size );
    }

    template < typename T >
    bool write( const T* buffer, std::size_t nmemb )
    {
        std::size_t bytes_size = nmemb * sizeof( T );
        return bytes_size == ::write( sockfd_, buffer, bytes_size );
    }

protected:
    int sockfd_;
};
```

Данный класс позволяет читать и записывать данные в произвольный сокет, а также закрывает сокет при уничтожении объекта.

Для использования UNIX domain-сокетов был написан класс *UnixSocket*, наследующийся от *Socket*:

```cpp
/// @brief Unix domain socket for IPC.
class UnixSocket : public Socket
{
public:
    explicit UnixSocket( const std::string& sock_file );

    ~UnixSocket();

    bool setup_server( size_t queue_size = 1 );
    bool setup_client();
    Socket accept_connection();

private:
    struct sockaddr_un sockaddr_;
    bool is_server_;
};
```

Данный класс позволяет устанавливать соединения с помощью файла сокета. Объект *UnixSocket* может быть либо клиентом, либо сервером.

Также был написан класс *Matrix*, реализующий все необходимые операции с матрицами. Для выполнения умножения матриц было создано 3 программы: *in_proc*, *calc_proc* и *out_proc*. Они обмениваются данными по сокетам */tmp/in_socket* и */tmp/out_socket* (файлы сокетов удаляются при закрытии серверов).

in_proc является сервером на сокете */tmp/in_socket*. Эта программа генерирует случайные матрицы и отправляет их по сокету. *calc_proc* является клиентом на */tmp/in_socket*, с которого она считывает матрицы. *calc_proc* также является сервером на сокете */tmp/out_socket*, по которому она передаёт вычисленное произведение полученных матриц. *out_proc* является клиентом на */tmp/out_socket* и выводит полученные матрицы.

2. Умножение матриц с помощью 3 потоков.

Для умножения матриц с помозью 3 потоков была создана программа *thread_proc*. В ней создаются 3 потока: *in_thread*, *calc_thread* и *out_thread*, выполняющие функции, аналогичные процессам:

```
// thread 1
std::thread in_thread( input, std::ref( lhs ), std::ref( rhs ), sizes );
// thread 2
std::thread calc_thread( calculate, std::cref( lhs ), std::cref( rhs ), std::ref( result ) );
// thread 3
std::thread out_thread( output, std::cref( lhs ), std::cref( rhs ), std::cref( result ) );

in_sync.notify();

in_thread.join();
calc_thread.join();
out_thread.join();
```

Так как потоки зависят друг от друга, необходимо производить синхронизацию между ними. Без синхронизации возможна следующая ситуация: *calc_thread* выполнился до того, как в *in_thread* были созданы матрицы. Для синхронизации использовался механизм *condition_variable*. Была написана структура *ThreadSync*:

```
struct ThreadSync
{
public:
     void wait();
     void notify();

private:
     std::condition_variable cond_;
     std::mutex mtx_;
     std::atomic_bool ready_{ false }; // flag for dealing with spurious and lost wakeups
};
```

Экземпляры данной структуры были созданы для каждого потока.

3. Выполнение параллельного умножения матриц.

Для умножения матриц с использованием *P* (число *P* – произовльное, задаётся в параметрах программы) потоков на основе программы *thread_proc* была создана программа *thread_proc_par*. В ней поток *calc_thread* запускал *P* потоков, которые считали определённые элементы результирующей матрицы. При этом число *P* не могло превышать размер результирующей матрицы.

4. Измерения производительности.

Для измерения производительности была использована команда *time* в bash, потому что для корректного сравнения скорости работы потоков и процессов необходимо учитывать время запуска и остановки программ. Измерялось реальное, системное и пользовательское время. Результаты измерений усреднены по 1000 запускам программ. Результаты приведены в табл. 1-3.

Таблица 1. Умножение матриц с помощью 3 процессов.

Размер матриц	User time, мс	System time, мс	Real time, мс
10x10	3,004	1,057	1,732
50x50	5,911	5,764	6,040
100x100	26,143	22,142	26,001
150x150	74,418	62,056	72,797

По таблице видно, что установка соединений по сокетам занимает константное время, а всё остальное время процессы обрабатывают матрицы.

Также во время измерений пользовательское и системное время складываются для 3 процессов, и, благодаря параллельной первоначальной настройке процессов, реальное время меньше.

Таблица 2. Умножение матриц с помощью 3 потоков.

Размер матриц	User time, мс	System time, мс	Real time, мс
10x10	1,226	0,578	1,777
50x50	4,654	1,575	6,315
100x100	24,242	2,165	26,578
150x150	71,859	3,118	76,226

По таблице видно, что большую часть времени программа производит расчёты (user time), а меньшую – ожидает в блокировках (system time). При этом, реальное время на 1-3% больше, чем при использовании процессов, потому что все потоки создаются и уничтожаются синхронно, а не параллельно.

На рис. 1 приведена зависимость реального времени умножения матриц 50x50 от количества (P) потоков. Измерения проводились на системах с 4 и 8 процессорами.

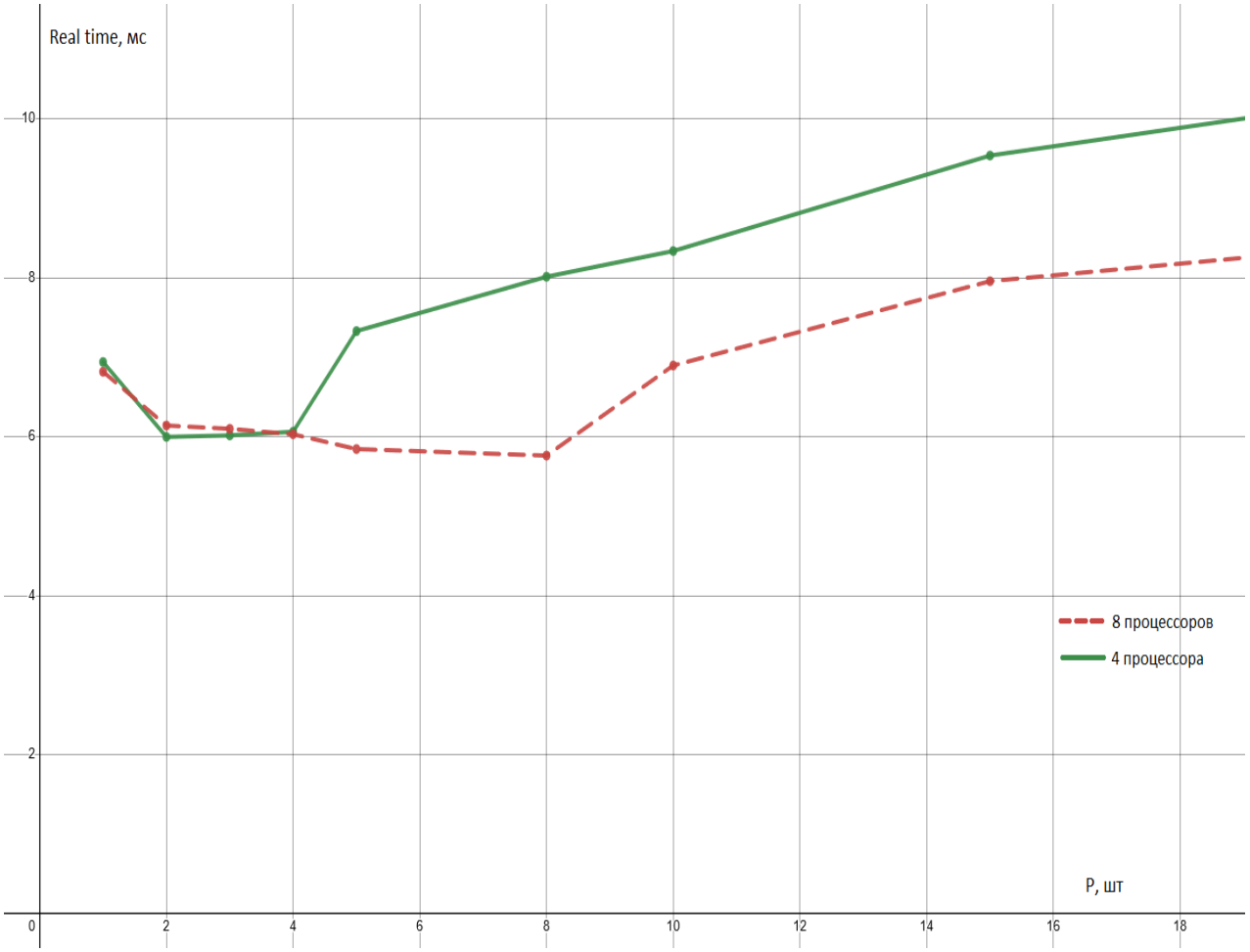

Рисунок 1. Графики зависимости времени исполнения от количества потоков расчёта на системах с 4 и 8 процессорами.

По графикам видно, что наименьшее время исполнения достигается, когда используется столько потоков, сколько процессоров в системе (*std::thread::hardware_concurrency*). Если потоков меньше, то не все процессоры заняты, а если больше – то процессоры вынуждены переключаться между исполнением разных потоков, отчего время исполнения растёт.

Выводы.

В ходе работы были изучены процессы и потоки, а также исследовано время выполнения расчётов (умножения матриц) в зависимости от количества потоков, проведено сравнение потоков и процессов.

В результате работы были сделаны заключения:

1. Установка соединений по сокетам занимает константное время, и первоначальная настройка процессов может выполняться параллельно.
2. Создание и уничтожение потока выполняется быстрее, чем создание и уничтожение процесса, но, благодаря асинхронному созданию процессов, процессы выполнялись быстрее на 1-3%.
3. При проведении параллельных вычислений с помощью потоков наименьшее время исполнения достигается, когда используется столько потоков, сколько процессоров в системе (*std::thread::hardware_concurrency*). Если потоков меньше, то не все процессоры заняты, а если больше – то процессоры вынуждены переключаться между исполнением разных потоков, отчего время исполнения растёт.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Исходный код программы доступен в репозитории https://github.com/Astana-Mirza/parallel_algo/tree/master.