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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Паралельнные алгоритмы»

Тема: Основы работы с процессами и потоками

Студент гр. 0304 _____ Максименко Е.М.

Преподаватель _____ Сергеева Е.И

Санкт-Петербург

2023

Цель работы.

Исследование способов организации межпроцессного и межпоточного взаимодействия. Исследование зависимости между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы.

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 строкам-столбцам).

Выполнение работы.

1. Для выполнения операций над матрицами реализован класс матрицы *Matrix*. Конструктор класса *Matrix* принимает размеры создаваемой матрицы. Поддерживаются операции получения размеров матрицы, получения

отдельных элементов либо массива всех элементов матрицы. Поддерживается ввод и вывод матрицы из потоков ввода/вывода и из сокета. Поддерживается вспомогательная операция умножения, которая вычисляет элемент матрицы — результата умножения, которая принимает две матрицы, а также номер строки и столбца. Объявление класса матрицы *Matrix* см. на рис. 1.

```
namespace common
{

class Matrix final
{
public:
    Matrix( size_t rows, size_t columns );

    inline size_t   rows() const { return rows_; }
    inline size_t   columns() const { return columns_; }
    inline float*   data() { return data_.data(); }
    inline const float* data() const { return data_.data(); }

    float           get( size_t row, size_t column ) const;
    void            set( size_t row, size_t column, float el );

    static bool     read( Matrix& matrix, std::istream& stream );
    static bool     read( Matrix& matrix, ipc::Socket& socket );

    static bool     write( const Matrix& matrix, std::ostream& stream );
    static bool     write( const Matrix& matrix, ipc::Socket& socket );

    static float    multiplyRC( const Matrix& lhs, const Matrix& rhs, size_t row, size_t column );

private:
    size_t rows_;               ///< Количество строк в матрице.
    size_t columns_;            ///< Количество столбцов в матрице.
    std::vector< float > data_; ///< Элементы матрицы.
}; // class Matrix

} // namespace common
```

Рисунок 1. Класс матрицы *Matrix*

2. Для работы с сокетами реализован класс *Socket*, который является оберткой для стандартного сокета. Конструктор класса принимает дескриптор созданного сокета, далее операции производятся с данным дескриптором. Основное предназначение данного класса — обертывание операций чтения и записи через сокет. Объявление класса *Socket* см. на рис. 2.

```
namespace ipc
{

class Socket
{
public:
    Socket( const int fd ) noexcept;
    virtual ~Socket();

    inline bool isValid() const { return isValid_; }
    inline const std::string& getErrMsg() const { return errMsg_; }

    template< typename T >
    ssize_t read( T* buffer, const size_t countElements = 1 )
    {
        if ( !isValid() )
        {
            errMsg_ = "Socket is invalid";
            return -1;
        }

        const size_t bufferSize = sizeof( T ) * countElements;
        return ::recv( fd_, reinterpret_cast< void* >( buffer ), bufferSize, 0 );
    }

    template< typename T >
    ssize_t write( T* buffer, const size_t countElements = 1 )
    {
        if ( !isValid() )
        {
            errMsg_ = "Socket is invalid";
            return -1;
        }

        const size_t bufferSize = sizeof( T ) * countElements;
        return ::send( fd_, reinterpret_cast< const void* >( buffer ), bufferSize, MSG_NOSIGNAL );
    }

    virtual void close();

protected:
    int fd_;                            ///< Файловый дескриптор сокета.
    bool isValid_ = true;               ///< Валидность сокета и соединения.
    std::string errMsg_;                ///< Сообщение об ошибке.

}; // class Socket

} // namespace ipc
```

Рисунок 2. Класс *Socket*

3. Реализована программа для выполнения задачи с помощью процессов. Создание процессов происходит с помощью вызова функции *fork* библиотеки *unistd*. Межпроцессное взаимодействие организовано с использованием сокетов, которые создаются с помощью вызова функции *socketpair* библиотеки *sys/socket*. Для корректного завершения процессов используется вызов *waitpid* библиотеки *unistd*.

Умножение матриц реализовано отдельной функцией *multiplyMatrices* из пространства имен *ipc*, которая принимает на вход две матрицы и возвращает результирующую матрицу.

4. Реализована программа для выполнения задачи с помощью потоков. Для инкапсуляции функций обработки, потоков и флагов готовности данных реализован класс *Tasks*. Конструктор класса *Tasks* принимает на вход количество потоков, выполняющих умножение матриц. Если количество потоков равно 1, то умножение производится в 1 потоке, иначе — распараллеливается на заданное количество потоков. Межпоточное взаимодействие синхронизируется с помощью атомарных флагов, сигнализирующих о готовности данных. Состояние флага проверяется в цикле без блокировок. Объявление класса *Tasks* см. на рис. 3.

```
namespace threads
{

class Tasks
{
public:
    Tasks( size_t threadsCount = 1 );

    bool isValid() const;

    void readMatrices( common::Matrix& lhs, common::Matrix& rhs );
    void multiplyMatricesSerial( const common::Matrix& lhs, const common::Matrix& rhs, common::Matrix& result );
    void multiplyMatricesParallel( const common::Matrix& lhs, const common::Matrix& rhs
        , common::Matrix& result, size_t threadsCount );
    void writeResultMatrix( const common::Matrix& result );

private:
    void invalidate();
    void storeAtomicBool( std::atomic< bool >& atomic, bool value );
    void waitForAtomicBool( const std::atomic< bool >& atomic, bool value );

private:
    common::Matrix              lhs_;               ///< Левая умножаемая матрица.
    common::Matrix              rhs_;               ///< Правая умножаемая матрица.
    common::Matrix              result_;            ///< Результирующая матрица.

    std::vector< std::thread >  threads_;           ///< Рабочие потоки.

    std::atomic< bool > isValid_;                   ///< Валидность программы.
    std::atomic< bool > calcAllowed_;               ///< Разрешение на чтение данных для второго потока.
    std::atomic< bool > writeAllowed_;              ///< Разрешение на чтение данных для третьего потока.

}; // class Tasks

}; // namespace threads
```

Рисунок 3. Класс *Tasks*

Запуск потоков реализован в функции *main*.

5. Реализована программа для выполнения задачи с помощью потоков с параллельным умножением. Реализация задач чтения, умножения и вывода реализована в классе *Tasks*. Количество потоков умножения задается в конструкторе класса *Tasks*.

6. Замеры времени выполнения программ.

Замер времени выполнения программ производится с помощью скрипта BASH и утилиты time. Программа запускается 100 раз для усреднения результатов. Данные генерируются с использованием BASH скрипта и записываются в файл.

Сведения о платформе, на которой производились замеры, приведены на рис. 4.

Рисунок 4. Сведения о платформе

Таблица 1. Зависимость времени выполнения программы для выполнения задачи с использованием процессов.

Размеры матриц	**Real Time, мс (100 замеров)**	**Sys. Time, мс (100 замеров)**
10x10, 10x10	351	171
20x20, 20x40	480	173
50x40, 40x50	831	191
100x100, 100x100	3965	290

150x150, 150x150	9097	507

Таблица 2. Зависимость времени выполнения программы для выполнения задачи с использованием потоков.

Размеры матриц	**Real Time, мс (100 замеров)**	**Sys. Time, мс (100 замеров)**
10x10, 10x10	284	156
20x20, 20x40	428	197
50x40, 40x50	744	204
100x100, 100x100	3620	270
150x150, 150x150	7900	342

Таблица 3. Зависимость времени выполнения программы для выполнения задачи с использованием потоков и параллельного умножения.

Размеры матриц	**Количество потоков**	**Real Time, мс (100 замеров)**	**Sys. Time, мс (100 замеров)**
10x10, 10x10	2	329	211
150x150, 150x150	2	6382	371
10x10, 10x10	3	332	211
150x150, 150x150	3	5277	366
10x10, 10x10	6	361	261
150x150, 150x150	6	4446	337
10x10, 10x10	12	451	307
150x150, 150x150	12	3795	421
10x10, 10x10	24	542	380
150x150, 150x150	24	3337	351

10x10, 10x10	48	577	407
150x150, 150x150	48	3293	565
10x10, 10x10	96	873	733
150x150, 150x150	96	3465	1052
10x10, 10x10	256	1543	1467
150x150, 150x150	256	4049	1709

Как видно по табл. 1 и табл. 2, реальное время выполнения (Real Time) многопоточной программы немного меньше времени выполнения многопроцессной программы. Это связано с тем, что создание процессов — процесс более «тяжелый» для системы, чем создание потоков.

По табл. 3 был составлен график зависимости реального времени исполнения (Real Time) от количества потоков (рис. 5). График составлен с использованием логарифмической шкалы для осей.

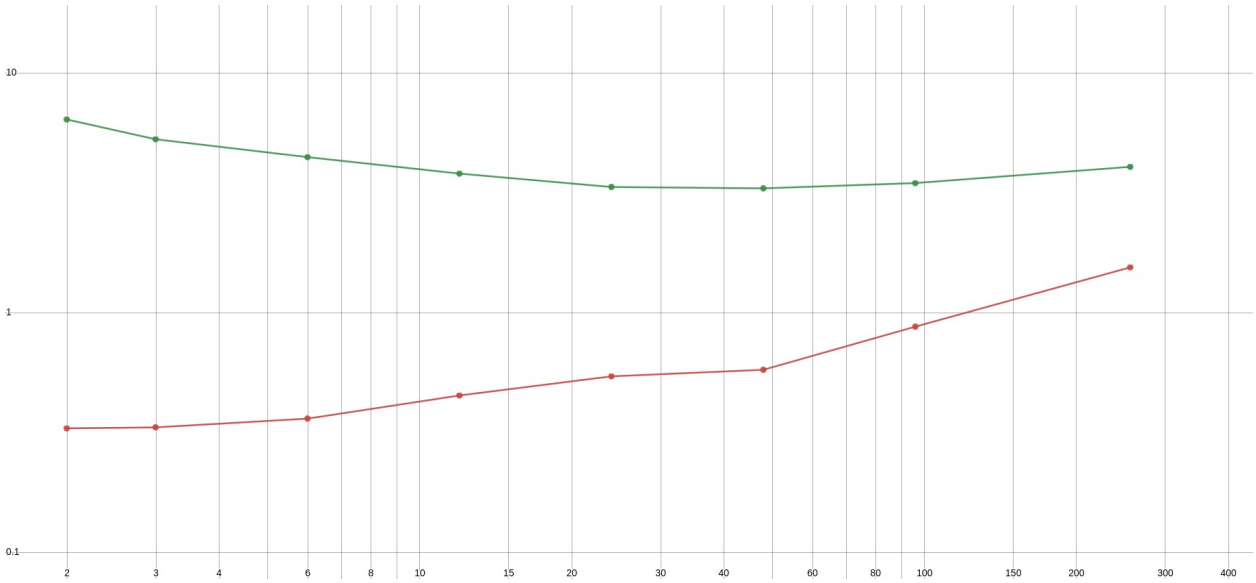

Рисунок 5. Зависимость среднего реального времени выполнения (сек) от количества потоков. Красный график — для матриц 10x10, 10x10. Зеленый график — для матриц 150x150, 150x150.

Как видно по графикам на рис. 5, зависимость времени выполнения присутствует как от количества потоков, так и от размеров матрицы.

Зависимость от размера матрицы связана с тем, что при небольших размерах матрицы время выполнения каждым потоком своей задачи сопоставимо со временем создания потока. При больших размерах матрицы время создания потоков оправдано временем выполнения задач каждым потоком.

Зависимость от количества потоков (для больших матриц) связана с тем, что большее количество потоков, выполняющих задачи параллельно, позволяет сократить общее время выполнения программы. Однако, так как задача умножения матрицы требует много процессорного времени (по сравнению с задачей чтения/записи), большое количество потоков приведет к частой смене контекста выполнения (см. Sys. Time в табл. 3), что приводит к возрастанию общего времени выполнения программы. Оптимальным с точки зрения теории является использование количества потоков, равное количеству аппаратных потоков. Для задачи умножения матриц 150x150 и 150x150 время переключения контекста оказалось не столь значимым (из-за относительно небольшого объема вычислений), как количество параллельных потоков, поэтому оптимальным числом потоков стало 24. В табл. 4 описаны дополнительные замеры для более сложных задач.

Таблица 4. Зависимость времени выполнения программы для выполнения задачи с использованием потоков и параллельного умножения (более сложные задачи).

Real Time, сек (1 замер)	**Количество потоков**	**6**	**12**	**24**
Размер матриц	-	-	-	-
300x300, 300x300	-	20.016	17.414	16.206
500x500, 500x500	-	68.006	59.296	57.130

1000x1000, 1000x1000	**-**	447.909	343.365	347.690

Как видно по таблице 4, с возрастанием объема вычислений использование большего количества потоков, чем имеется аппаратных потоков, не дает прироста производительности. Использование меньшего количества потоков, чем имеется аппаратных, не позволяет достичь максимального уровня производительности. Наилучший результат для большого объема вычислений — распараллеливание на количество потоков, равно количеству аппаратных потоков.

Выводы.

В ходе лабораторной работы были исследованы способы организации межпроцессного и межпоточного взаимодействия. Было проведено исследование зависимости между количеством потоков, размерами входных данных и параметрами целевой вычислительной системы.

В ходе исследования было установлено, что многопоточная программа работает в среднем быстрее, чем многопроцессная. Это связано с тем, что создание процесса занимает большее количество времени, чем создание потока. Также была установлена зависимость оптимального количества потоков, выполняющих сложную вычислительную задачу, от параметров аппаратной платформы. Так, оптимальное количество потоков оказалось равно количеству аппаратных потоков.