Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Пермский национальный исследовательский политехнический университет»

Факультет: Прикладной математики и механики
Кафедра: Вычислительной математики, механики и биомеханики

Направление: 09.04.02 Информационные технологии и системная инженерия
Профиль: «Информационные технологии и системная инженерия»

Лабораторные работы по дисциплине: «Параллельное программирование»

Выполнил
студент гр. ИТСИ-24-1м
Лушин Андрей Петрович

Принял Преподаватель кафедры ВММБ
Истомин Денис Андреевич

Пермь 2025

Оглавление

Лабораторная работа 1

Задание

Необходимо:

1. При помощи SSE инструкций написать программу (или функцию), которая перемножает массив из 4х чисел размером 32 бита;
2. Написать аналогичную программу (или функцию), которая решает ту же задачу последовательно;
3. Сравнить производительность;
4. Проанализировать сгенерированный ассемблер: gcc -S sse.c.

Особенности реализации

В работе реализовано два способа перемножения массивов из четырёх чисел типа `float`:

1. **SSE-реализация**
   Для ускорения расчётов использовалась встроенная ассемблерная вставка с использованием инструкций SSE.
   - В функции `sse_mul` используется три инструкции:
     - `movups` для загрузки данных из массива `a` в регистр `xmm0`;
     - `movups` для загрузки данных из массива `b` в регистр `xmm1`;
     - `mulps` для параллельного перемножения четырёх пар чисел (SIMD);
     - `movups` для сохранения результата обратно в массив `c`.
   - Все данные представляются массивами по 4 элемента (это соответствует ширине SSE-регистра XMM — 128 бит).
   - Благодаря использованию SIMD-инструкций, все 4 умножения выполняются одной командой процессора.
2. **Последовательная реализация**
   Для сравнения была реализована функция `seq_mul`, которая перемножает по одному элементу из двух массивов с помощью обычного цикла `for` (4 итерации).
   - Каждая пара элементов перемножается отдельно, результат сохраняется в выходной массив.
   - Данный способ не использует векторные расширения и полностью последовательный.

Для замеров производительности использовался POSIX-таймер (`clock_gettime` с `CLOCK_MONOTONIC`).
В каждом случае функция (SSE или последовательная) вызывалась многократно (`outer` раз), чтобы исключить влияние погрешности измерения и получить достаточно длительный временной промежуток.

Результат

Результаты выполнения при одинаковых входных данных:

```
crsd@Blade:~$ ./lab1 100000
SSE:        0.000272 s   -> 5.00 12.00 21.00 32.00
Sequential: 0.001270 s   -> 5.00 12.00 21.00 32.00
```

Рисунок 1 Результат работы программы для лаб.1

Вывод

SSE-реализация справилась с задачей примерно в **4.7 раза быстрее** (0.001270 / 0.000272 ≈ 4.67).

Оба способа выдали одинаковый корректный результат по значениям элементов.

Полученное ускорение связано с тем, что SSE-инструкция `mulps` выполняет сразу четыре умножения за один такт, а последовательный цикл — по одному умножению за раз.

Лабораторная работа 2

Задание

Необходимо:

1. При помощи Pthreads написать программу (или функцию), которая создает n потоков и каждый из потоков выполняет длительную операцию;
2. Написать аналогичную программу (или функцию), которая решает ту же задачу последовательно;
3. Сравнить производительность.

Особенности реализации

Написаны две версии программы на C:

- Последовательная: главный поток поочерёдно выполняет вычисление n раз.
- Многопоточная: с помощью POSIX `threads (pthread_create)` запускаются n потоков, каждый считает свою часть работы.

Для синхронизации доступа к разделяемым данным (например, переменной `counter`, если она используется несколькими потоками) применён мьютекс (`pthread_mutex_t`).

Это важно: если несколько потоков одновременно изменяют одну переменную — возникает гонка данных (race condition), результат становится непредсказуемым. Мьютекс обеспечивает, что только один поток заходит в критическую секцию в каждый момент времени.

Для замера времени использован `clock_gettime(CLOCK_MONOTONIC, ...)`.

Для предотвращения оптимизаций компилятора результат аккумулируется в `volatile double acc`

Результат

```
crsd@Blade:~$ ./lab2_3 15
=== Последовательное выполнение ===
        Sequential task #0 started
        Sequential task #0 finished
        Sequential task #1 started
        Sequential task #1 finished
        Sequential task #2 started
        Sequential task #2 finished
        Sequential task #3 started
        Sequential task #3 finished
        Sequential task #4 started
        Sequential task #4 finished
        Sequential task #5 started
        Sequential task #5 finished
        Sequential task #6 started
        Sequential task #6 finished
        Sequential task #7 started
        Sequential task #7 finished
        Sequential task #8 started
        Sequential task #8 finished
        Sequential task #9 started
        Sequential task #9 finished
        Sequential task #10 started
        Sequential task #10 finished
        Sequential task #11 started
        Sequential task #11 finished
        Sequential task #12 started
        Sequential task #12 finished
        Sequential task #13 started
        Sequential task #13 finished
        Sequential task #14 started
        Sequential task #14 finished
Wall-time sequential: 6.082 s
```

Рисунок 2 Последовательное вычисление лаб2

```
=== Pthreads выполнение ===
MAIN: starting thread 0
MAIN: starting thread 1
        Thread #0 started
MAIN: starting thread 2
MAIN: starting thread 3
MAIN: starting thread 4
MAIN: starting thread 5
        Thread #1 started
        Thread #3 started
        Thread #2 started
MAIN: starting thread 6
MAIN: starting thread 7
MAIN: starting thread 8
MAIN: starting thread 9
MAIN: starting thread 10
MAIN: starting thread 11
MAIN: starting thread 12
MAIN: starting thread 13
MAIN: starting thread 14
        Thread #4 started
        Thread #12 started
        Thread #14 started
        Thread #13 started
        Thread #11 started
        Thread #6 started
        Thread #9 started
        Thread #5 started
        Thread #7 started
        Thread #8 started
        Thread #10 started
        Thread #4 finished
        Thread #11 finished
        Thread #9 finished
        Thread #12 finished
        Thread #0 finished
        Thread #5 finished
        Thread #14 finished
        Thread #1 finished
        Thread #8 finished
        Thread #6 finished
        Thread #10 finished
        Thread #2 finished
        Thread #3 finished
        Thread #7 finished
        Thread #13 finished
Wall-time Pthreads:   1.969 s
```

Рисунок 3 Многопоточное вычисление Pthreads лаб2

Вывод

Использование Pthreads дало ускорение примерно в 3.5 раза на 4 ядрах.

Мьютекс необходим для корректной работы при доступе к общим переменным: если его не использовать, результат вычислений может быть неверным из-за одновременных изменений памяти разными потоками.

Корректность обеспечена синхронизацией доступа и защитой от оптимизаций.

Программа масштабируется с увеличением числа ядер, но ускорение ограничивается физическими ресурсами системы.

Лабораторная работа 3

Задание

Необходимо:

1. При помощи OpenMP написать программу (или функцию), которая создает n потоков и каждый из потоков выполняет длительную операцию;
2. Сравнить с последовательной программой и программой с Pthreads из предыдущей лабораторной работы.

Особенности реализации

Используется код с прошлой лабораторной работы с дополнительной версией распараллеливания посредством OpenMP.

Перед циклом запуска «тяжелой задачи» используется директива `#pragma omp parallel for schedule(static,1)`, настройка вручную указана для того, чтобы гарантировать что каждый поток получит по одной задаче, особенно полезно при большом количестве независимых задач: все потоки равномерно загружены, не простаивают, результат выполнения максимально сбалансирован по времени.

Число потоков устанавливается через `omp_set_num_threads(n)`, где n — количество задач. Замеры времени реализованы через встроенную функцию: `omp_get_wtime()`.

Результат

Рисунок 4 многопоточное вычисление OpenMP лаб3

Вывод

В этом запуске OpenMP оказался немного эффективнее (в 3,7 раза относительно последовательной, при показателе в 3,5 у Pthreads(демонстрируется на рисунках 2 и 3)),

потому что его рантайм чуть лучше распределил задачи по ядрам, и нет накладных расходов на ручное управление потоками.

После того, как число потоков превышает число физических ядер (в моем случае 4), ускорение перестаёт расти линейно: потоки начинают конкурировать за процессорное время, и прирост эффективности замедляется.

Но даже на 15 задачах видно, что параллельные технологии дают большую выгоду, если задачи достаточно тяжёлые (затраты на создание и синхронизацию потоков не перекрывают выигрыш от распараллеливания).

Реализация программного кода OpenMP в том числе позволяет разработчику «параллелить» задачи одной строчкой не требуя ручного управления потоками и синхронизацией как у Pthread.

Лабораторная работа 4

Задание

Необходимо:

1. Написать программу, которая запускает несколько потоков.
2. В каждом потоке считывает и записывает данные в HashMap, Hashtable, synchronized HashMap, ConcurrentHashMap.
3. Модифицировать функцию чтения и записи элементов по индексу так, чтобы в многопоточном режиме использование непотокобесопасной коллекции приводило к ошибке.
4. Сравнить производительность.

Особенности реализации

Использован язык Java и пул потоков ExecutorService для запуска 50 потоков.
Все коллекции тестируются на одинаковом сценарии: 3 ключа, 100 000 инкрементов на поток (итого 5 млн операций).
Реализованы три варианта:

Стандартный read-modify-write:

```
int oldVal = map.getOrDefault(key, 0);
map.put(key, oldVal + 1);
```

Проверяется для всех коллекций.

Внешний synchronized-блок:

```
synchronized (map) {
   int oldVal = map.getOrDefault(key, 0);
   map.put(key, oldVal + 1);
}
```

Все операции внутри одного синхронизированного блока.

Атомарный инкремент (через merge):

```
map.merge(key, 1, Integer::sum);
```

После каждого теста выводится ожидаемое и фактическое значение суммы инкрементов.

Результат

```
=== Стандартный read-modify-write (get+put) ===
HashMap:
    Expected sum: 5000000
    Actual sum:   3095891
    Time: 0,326 s
    [DATA RACE!]

Hashtable:
    Expected sum: 5000000
    Actual sum:   826937
    Time: 0,763 s
    [DATA RACE!]

SynchronizedMap:
    Expected sum: 5000000
    Actual sum:   754998
    Time: 0,826 s
    [DATA RACE!]

ConcurrentHashMap:
    Expected sum: 5000000
    Actual sum:   2493915
    Time: 0,350 s
    [DATA RACE!]
```

```
=== Синхронизированный read-modify-write
HashMap:
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,485 s
    [OK]
Hashtable (synchronized block):
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,534 s
    [OK]
SynchronizedMap (synchronized block):
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,698 s
    [OK]
ConcurrentHashMap:
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,646 s
    [OK]
```

Рисунок 5 синхронизированный read-modify-write лаб4

Рисунок 6 read-modify-write лаб4

```
=== Атомарный инкремент для
HashMap:
    Expected sum: 5000000
    Actual sum:   2790006
    Time: 0,125 s
    [DATA RACE!]

Hashtable:
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,377 s
    [OK]
SynchronizedMap:
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,441 s
    [OK]
ConcurrentHashMap:
    Expected sum: 5000000
    Actual sum:   5000000
    Time: 0,386 s
    [OK]
```

Рисунок 7 Атомарный метод лаб4

Вывод

Метод **read-modify-write** не подходит для многопоточного инкремента ни в одной из коллекций. Во всех случаях наблюдаются потери данных из-за неконсистентности операций: между чтением и записью значения могут вмешаться другие потоки.

Внешняя синхронизация (**synchronized**) полностью решает проблему потерь, но существенно (вплоть до 45%) снижает производительность из-за блокировки всей коллекции для каждого инкремента.

Атомарные методы (**merge**) работают корректно и быстрее всего на **ConcurrentHashMap**, **Hashtable** и **SynchronizedMap** — обновления происходят без потерь и без полной блокировки структуры. Для **HashMap** это не работает, так как коллекция изначально не потокобезопасная и не имеет встроенной защиты.

HashMap не следует использовать в многопоточных сценариях, даже с атомарными методами. Если нужна потокобезопасность следует выбирать другие реализации.

Лабораторная работа 5
Задание

Необходимо:

1. Написать программу, которая демонстрирует работу считающего семафора
2. Написать собственную реализацию семаформа (наследование от стандартного с переопределением функций) и использовать его

Особенности реализации

Ключевые классы:

Semaphore (java.util.concurrent) — стандартный семафор.

MySemaphore — собственная реализация семафора (через наследование от Semaphore и использование ReentrantLock).

Запускается пул из 4 потоков (`THREADS = 4`)

Каждый поток получает разрешение на выполнение только при наличии доступных permits (COUNT = 2).

Каждый поток захватывает разрешение у семафора, выполняет работу (симуляция через Thread.sleep(1000)), и освобождает permit.

Выводится количество одновременно работающих потоков.

Особенности синхронизации:

В MySemaphore для управления разрешениями используется ReentrantLock + Condition.

```
lock.lock();
try {
   while (permits <= 0) {
      permitsAvailable.await();
   }
   permits--;
} finally {
   lock.unlock();
}
```

Число потоков: 4 **Число разрешений (permits):** 2

Для предотвращения потерь разрешений и гонок вся работа с permits обёрнута в lock/unlock. Количество одновременно активных потоков не превышает 2. Само тестирование проводится для обоих вариантов — стандартный семафор и собственный.

Результат

```
------------------
Regular semaphore:
------------------
Поток pool-1-thread-2 работает. Активных потоков: 2
Поток pool-1-thread-1 работает. Активных потоков: 1
Поток pool-1-thread-4 работает. Активных потоков: 2
Поток pool-1-thread-3 работает. Активных потоков: 1
Максимальное количество активных потоков: 2
-------------
My semaphore:
-------------
Поток pool-2-thread-1 работает. Активных потоков: 1
Поток pool-2-thread-2 работает. Активных потоков: 2
Поток pool-2-thread-3 работает. Активных потоков: 2
Поток pool-2-thread-4 работает. Активных потоков: 2
Максимальное количество активных потоков: 2
```

Рисунок 8 семафоры лаб5

Вывод

Оба варианта семафора корректно ограничивают число одновременно выполняющихся потоков, защищая критическую секцию от переполнения.

Реализация через ReentrantLock и Condition более гибкая, чем стандартный synchronized-блок: даёт возможность управления честностью, поддерживает несколько условий ожидания и удобна для сложной логики синхронизации.

Масштабируемость: ограничение потоков зависит только от числа permits, логика легко расширяется под большее количество разрешений или потоков.

Узкое место — только на этапе захвата и освобождения permits, остальное не блокируется.

Лабораторная работа 6
Задание

Необходимо создать клиент-серверное приложение:

1. Несколько клиентов, каждый клиент - отдельный процесс
2. Серверное приложение - отдельный процесс
3. Клиенты и сервер общаются с использоваени Socket

Необходимо релизовать функционал:

1. Клиент подключается к серверу
2. Сервер запоминает каждого клиента в java.util.concurrent.CopyOnWriteArrayList
3. Сервер читает ввод из консоли и отправляет сообщение всем подключенным клиентам

Особенности реализации

Реализован многопользовательский чат-сервер на Java, с использованием TCP-соединений через Socket API. Каждый клиент запускается в отдельном процессе, подключается к серверу, отправляет и принимает сообщения в реальном времени

Коллекция: `CopyOnWriteArrayList` для хранения активных подключений

Каждый клиент обрабатывается в отдельном потоке через реализацию `Runnable`. Сервер запускает отдельный поток для чтения консольного ввода администратора. Сервер создаёт `ServerSocket` на порту 12345 и слушает подключения.

При подключении создаётся объект `ClientHandler`, который добавляется в `CopyOnWriteArrayList`. `ClientHandler` читает сообщения от клиента и рассылает их остальным через static-метод `broadcast`.

Клиент подключается к серверу по адресу localhost:12345 и в двух потоках:

- Читает сообщения с сервера
- Считывает ввод пользователя с консоли и отсылает его

Серверный `broadcast` исключает отправителя из получателей.

Отключение клиента корректно обрабатывается через `SocketException` и удаление его из списка клиентов и имён.

Обработка консоли сервера запускается в отдельном потоке, не блокируя основной `accept()`.

При отключении клиента производится корректное закрытие всех ресурсов (`in`, `out`, `socket`). Сервер остаётся живым и продолжает работать при отключении клиентов.

Проверка работоспособности производится запуском нескольких клиентов параллельно (через разные терминалы).

Потокобезопасное добавление клиента: clients.add(clientHandler);

Рассылка сообщений: for (ClientHandler client : clients) { if (client != sender) client.sendMessage(msg); }

Корректное удаление клиента: clients.remove(this); names.remove(username);

Результат

```
PS D:\prp\lab6> java -cp target\classes Server
Server is running and waiting for connections...
New client connected: Socket[addr=/127.0.0.1,port=58369,localport=12345]
User vasya connected.
New client connected: Socket[addr=/127.0.0.1,port=58377,localport=12345]
User vladimir connected.
l33t
[vasya]: it is
[vladimir]: q
```

Рисунок 9 сервер лаб6

```
PS D:\prp\lab6> java -cp target\classes Client
Connected to the chat server!
Enter your username:
vladimir
Welcome to the chat, vladimir!
Type Your Message
[Server]: l33t
[vasya]: it is
```

Рисунок 10 клиент1 лаб6

```
PS D:\prp\lab6> java -cp target\classes Client
Connected to the chat server!
Enter your username:
vasya
Welcome to the chat, vasya!
Type Your Message
[Server]: l33t
it is
[vladimir]: q
```

Рисунок 11 клиент2 лаб6

Вывод

Приложение реализует корректную многопоточную серверную архитектуру на сокетах. Для хранения подключённых клиентов использована потокобезопасная коллекция CopyOnWriteArrayList.

Несмотря на то, что CopyOnWriteArrayList не самая эффективная структура при частых изменениях (так как при каждом добавлении или удалении создаётся копия массива), она обеспечивает безопасную итерацию без необходимости явной синхронизации.

Это особенно важно в контексте метода broadcast, который параллельно с удалением клиентов рассылает сообщения всем остальным. Потокобезопасность обеспечена на

уровне коллекции без использования внешних synchronized-блоков или явных мьютексов, что упрощает архитектуру и снижает риск ошибок синхронизации.
Такой подход удобен для задач, где операции чтения происходят гораздо чаще, чем модификации.

Лабораторная работа 7

Задание

1. Java IPC (Использование библиотеки MappedBus: запустить example)

Особенности реализации

Реализуем межпроцессное взаимодействие через общий memory-mapped файл: несколько **ObjectWriter** пишут объекты, а **ObjectReader** их читает.

Язык и библиотеки: Java 8, java.nio + внутренние API (**sun.nio.ch.FileChannelImpl, sun.misc.Unsafe**).

MappedBus делает memory-mapped file без блокировок, с высокой пропускной способностью и низкой задержкой. Ring-buffer логика.

Я не стал делать global path, поэтому прописал в терминале:
$Env:JAVA_HOME = 'C:\Users\crsd\.jdks\corretto-1.8.0_452'
$Env:Path   = "$Env:JAVA_HOME\bin;$Env:Path"
Затем непосредственно вызов
java -cp mappedbus.jar io.mappedbus.sample.object.ObjectWriter 0
java -cp mappedbus.jar io.mappedbus.sample.object.ObjectReader

Результат

```
Read: PriceUpdate [source=0, price=40, quantity=80], hasRecovered=false
Read: PriceUpdate [source=1, price=76, quantity=152], hasRecovered=false
Read: PriceUpdate [source=0, price=42, quantity=84], hasRecovered=true
Read: PriceUpdate [source=1, price=78, quantity=156], hasRecovered=true
Read: PriceUpdate [source=0, price=44, quantity=88], hasRecovered=true
```

Рисунок 11 результат работы ObjReader`a на данные ObjWriter 0 и 1