LECTURE 4: A DESIGN PATTERN BASED APPROACH TO CONCURRENCY AND PARALLELIZATION (WITH JAVA, MPI AND OPEN MP)
Lecture Contents

• Focus: How do we get this stuff into code?
• Recap on challenges for concurrent programming
• Refresh on Java threads and mutual exclusions
• Steps in Designing concurrent/parallel programs
• Introducing Design Patterns
• The Parallel/Concurrent Design Pattern Spaces
SECTION 4.1: DESIGNING CONCURRENT/PARALLEL PROGRAMS
Parallel & Concurrent Programming
Types Discussed in this Course

• Shared Memory Machines
  – Communication is through shared variables
  – Java concurrency, Open MP

• Distributed Memory Machines (eg Clusters)
  – Needs message passing for communication
  – MPI, (Java** + RMI, web services, or REST)*

* Covered in distributed systems part of course
** or any other language
Recap on Java Threads (/1)

- Java thread is a lightweight process with its own stack & execution context, access to all variables in its scope.
- Can be programmed by extending `Thread` class or implementing `Runnable` interface.
- Both of these are part of standard `java.lang` package.
- `Thread` instance is created by:
  ```java
  Thread myProcess = new Thread();
  ```
- New thread started by executing:
  ```java
  MyProcess.start();
  ```
- `start` method invokes a `run` method in the thread.
- As `run` method is undefined as yet, code above does nothing.
Recap on Threads (/2)

- We can define the `run` method by extending the `Thread` class:

```java
class myProcess extends Thread {
    public void run () {
        System.out.println("Hello from the thread");
    }

    myProcess p = new myProcess();
    p.start();
}
```

- Best to terminate threads by letting `run` method to terminate.
- If don’t need a ref to new thread omit `p` and simply write:

```java
new myProcess().start();
```
Recap on Threads (/3)

- As well as extending `Thread` class, can create lightweight processes by implementing `Runnable` interface.
- Advantage: can make your own class, or a system-defined one, into a process.
- Avoids lack of multiple inheritance in Java with `Thread` class as Java only allows for one class at a time to be extended.
- Using the `Runnable` interface, previous example becomes:

```java
class myProcess implements Runnable {
    public void run() {
        System.out.println("Hello from the thread");
    }
}
Runnable p = new myProcess();
New Thread(p).start();
```
Recap on Threads (/4)

• If it has nothing immediate to do (e.g., it updates screen regularly) should suspend thread by putting it to sleep.
• 2 flavours of `sleep()` method (specifying different times)
• `join()` awaits specified thread finishing, giving basic synchronisation with other threads.
  – i.e., "join" start of a thread's execution to end of another thread's execution
  – thus thread will not start until other thread is done.
• If `join()` is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing:

```java
try {
    otherThread.join (1000); // wait for 1 sec
} catch (InterruptedException e) {} 
```
Recap: Concurrent Program Challenges

• Must ensure concurrent execution is safe compared to serial program, i.e.
  – No faults
  – Correct + reliable answers
  – Performance is not compromised

• Hazards
  – Data dependency: shared data changing dependent on access order
  – Name dependency: same memory location used by multiple processes
  – Control dependency: code branches enforce ordering

• Debug of these programs is much harder than serial programs
  → Need strong software engineering discipline
  → Use Design Patterns
Assuring Safety

- Multiple threads can access the same resource safely only if
  - all accesses have no effect on resource (read only). or
  - all accesses are idempotent, or
  - only one access at a time (mutual exclusion, mutex) - allows write access

- Mutex prevents multiple threads accessing a critical block of code at the same time

⇒ Introduces synchronization between threads
  ⇒ Creates potential for deadlocks

- Wider blocks create safety but induce delays as they reduce parallelism
Potential Concurrency Problems

- **Deadlock** - 2+ threads stop and wait for each other (forever)
- **Liveloop** - 2+ threads continue to execute but make no progress toward goal (spinning)
- **Starvation** - some thread gets deferred forever (especially with ordering constraints...that we used to avoid deadlock)
- **Lack of fairness** - not all threads get a turn to make progress
- **Race condition** - some possible interleaving results in an undesired computation result. This is caused by an unprotected shared object/memory.

NB Very imp to be clear on who owns data and at what time.
Race Conditions

• Application behaviour depends on sequence or timing of processes or threads eg accessing a buffer
• Non-deterministic, timing dependent
• Cause data corruption, crashes
• Difficult to detect, reproduce + eliminate
• Typically caused by:
  – Programming errors
  – Failure to apply good lock discipline
  – Scheduler controlled interleaving of threads
  – Trying to make performance improvements to code
• Avoid by appropriate use of mutual exclusion
Example: Non-thread-safe Java

```java
@NotThreadSafe
class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```

**Listing 1.1.** Non-thread-safe sequence generator.

**Figure 1.1.** Unlucky execution of UnsafeSequence.getNext.
Thread-safe Java

• If multiple threads access the same mutable state variable without appropriate synchronization, your program is broken.

• Three ways to fix:
  – Don’t share the state variable across threads; (use encapsulation)
  – Make the state variable immutable; or
  – Use synchronization whenever accessing the state variable

• Thread-safe class = it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code

• Stateless objects are always thread-safe.
• Immutable objects are always thread-safe.
• Atomic state variable updates are safe
  – If only a single state variable: can use built-in Atomic types like AtomicLong
  – Else: need to add mutex via synchronization
Mutual Exclusion in Java

- Java's supports two kinds of thread synchronization: *mutual exclusion* and *cooperation*:
  - **Mutual exclusion (AKA Locks):**
    - Supported in JVM via object locks (aka ‘mutex’),
    - Enables multiple threads to independently work on shared data without interfering with each other.
  - **Cooperation:**
    - Supported in JVM via the `wait()`, `notify()` methods of class `Object`,
    - Enables threads to work together towards a common goal.
    - Discussed later
Mutual Exclusion in Java: Synchronization

- Conceptually threads in Java execute concurrently, so could simultaneously access shared variables (aka *A Race Condition*).

- To prevent when updating a shared variable, Java provides synchronisation
  - It marks a section of code as **atomic**

- Java’s keyword **synchronized** provides mutual exclusion and can be used with a group of statements or with an entire method.

- The following class will potentially have problems if its update method is executed by several threads concurrently:

```java
class Problematic {
    private int data = 0;
    public void update() {
        data++;
    }
}
```
Mutual Exclusion in Java: Synchronization (/2)

- Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables.

```java
class ExclusionByMethod {
    private int data = 0;
    public synchronized void update (){
        data++;
    }
}
```

- To preserve state consistency, update related state variables in a single atomic operation.
- Only 1 lock per object in Java thus if a `synchronized` method is invoked the following occurs:
  - it waits to obtain the lock,
  - executes the method, and then
  - releases the lock.
- This is known as *intrinsic locking*.
- Java intrinsic locks are *reentrant*: if a thread tries to acquire a lock that it already holds, the request succeeds.
Mutual Exclusion in Java: Synchronization (/3)

- Can also have Mutual exclusion with *synchronized* statement in method’s body:

```java
class ExclusionByGroup {
    private int data = 0;
    public void update (){
        synchronized (this) { // lock this object for
            data++; // the following group of
        } // statements
    }
}
```

- A *synchronized* statement specifies that the following group of statements is executed as an atomic, non interruptible, action.

- A synchronized block has two parts:
  - A reference to an object that will serve as the *lock*,
  - A block of code to be guarded by that lock.

- A synchronized method is a shorthand for a synchronized block that spans an entire method body, and whose lock is the object on which the method is being invoked.

- Every Java object can implicitly act as a lock for purposes of synchronization
Mutual Exclusion in Java: Limitations (/4)

- At most one thread can own a mutex/intrinsic lock
  ⇒ Deadlock easy to achieve
  e.g. When thread A attempts to acquire a lock held by thread B, A must wait, or block, until B releases it. If B never releases the lock, A waits forever.
- Mutex makes code serial
  ⇒ Can lead to very poor performance
- Just synchronising every method is sometimes not enough, additional locking is required when multiple operations are combined into a compound action.
Why is this Example Bad?

Example from ref[6]

Listing 2.6. Servlet that caches last result, but with unacceptably poor concurrency. Don’t do this.

@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}

Cache of last result
To improve performance
Rationale

- Caching required shared state
- Protected it with coarse-grained mutex => safe
- If servlet is busy, new customers (threads) must wait
- Even with multiple CPUs, all threads must wait

=> try to exclude from synchronized blocks long-running operations (e.g. I/O) that do not affect shared state

Figure from ref[6]
Concurrent Solution

- Note: There is frequently a tension between simplicity and performance. When implementing a synchronization policy, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance.

Figure from ref[6]
Mutual Exclusion in Java: Guarding State (/5)

- If synchronization is used to coordinate access to a variable, it is needed *everywhere that variable is accessed*
  - E.g. not just when initialized
  - All accesses must use the same lock
    - For convenience Java creates one intrinsic lock per object so you don’t have to explicitly create lock objects
  - You must protect read access too
- Acquiring the lock associated with an object does *not* prevent other threads from accessing that object—the only thing that acquiring a lock prevents any other thread from doing is acquiring that same lock.
- It is up to you to construct *locking protocols or synchronization policies* that let you access shared state safely, and to use them consistently throughout your program
  - See Design Patterns
Java Memory Visibility

• Synchronized is not only about atomicity or demarcating “critical sections” of code
• It also determines memory visibility
• With multiple co-operating objects and threads we need to ensure that when a thread modifies the state of an object, other threads can actually see the changes that were made
• There is no guarantee that operations in one thread will be performed in the order given by the program, as long as the reordering is not detectable from within that thread—even if the reordering is apparent to other threads.
Example

```java
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Listing 3.1. Sharing variables without synchronization. Don’t do this.

Figure from ref[6]
Fixes for Stale Data

• Unless synchronization is used every time a variable is accessed, it is possible to see a stale value for that variable.

• Stale object references are very dangerous

• Solution: use synchronized get/set
**Common Steps to Parallelise a Program**

1. Start with **sequential program or algorithm**
2. Decompose into tasks
3. Assign tasks to processes
4. Decide communication pattern
5. Map to Processors/Hardware

---

**Lecture 4:** Concurrency Design Patterns

CA4006 Lecture Notes (Rob Brennan & Martin Crane 2019)
Performance

- Factors that affect performance of parallel programs are:
  - Coverage - ie extent of parallelism in algorithm
    • See Amdahl’s Law from earlier lectures
  - Granularity - how is it partitioned among processors?
  - Locality - how do you allocate it to hardware as not all processors are equal in terms of capacity + communication costs
Granularity

• Depends on how you divide up the problem
• What is the ratio of communication:computation?
• Communication used for:
  – Synchronisation/control
  – Data exchange
Fine vs Coarse Granularity

• Fine grain
  – low comp to commo ratio
  – Small amounts of comp work between communication stages
  – Less opportunity for perf enhancement
  – High communication bandwidth

• Coarse grain
  – Computation takes much longer than commo
  – More opportunity for performance increase
  – Harder to balance efficiently (load balancing)
Load Balancing Problem

• Tasks that finish first need to wait for the processor with the largest amount of work to finish => idle time
Load Balancing Solutions

- Static load balancing => programmer knows the work and assigns them a priori
  - Works well if: you understand the application, processors are homogeneous, there is an even distribution of work
- Dynamic load balancing => one processor finishes with work => it takes work from the heaviest workload, or have a work queue
Synchronisation Points

• Are there ordering constraints on execution?
  – Depends on how you cut up data or allocate tasks to processors

• Potentially all join points in an algorithm are synchronisation points
  – Can avoid by increasing granularity
  – Everything in one task = sequential program
Communication

• Expensive
• 2 main types:
  – Control messages (short)
  – Data messages (long)
• Communication Patterns
  – Point to point (one to one)
  – Broadcast (one to all) and reduce (all to one)
  – All to all
  – Scatter (one to several) and gather (several to one)
• Defer discussion to distributed memory systems later
Problem Decomposition

• Identify concurrency and decide at what level to exploit it
  – Tasks, data, pipelines

• Break computation into tasks to be divided among processors
  – Tasks may become available dynamically
  – Number of tasks may vary with time

• Make sure you have enough tasks to keep processors busy
  – i.e. number of tasks provides an upper bound on how much you can do in parallel/the available speedup
Task Assignment (decide granularity)

• Specify a mechanism to divide work among cores
  – load balance the work and minimise commo
• Structured approaches can help
  – Code inspection or understanding the application
  – **Well-known design patterns**
• Programmers really worry about partitioning first
  – ie figuring out the parts of the application that I need to compose to make the application
  – NB this is independent of hardware architecture or programming model!
  – You want to be able to keep program complexity down (so people often ignore highly complex solutions)
Orchestration and Mapping

• Orchestration = picking a communication pattern
• Try to preserve data locality
• Address scheduling
Why Use Design Patterns?

• Software problems recur - designers/engineers have to solve from first principles (although experience helps)
• Patterns provide a means of documenting and communicating successful (non-obvious) solutions to recurring problems.
• Cookbook of solution templates
  – Must be specialised for a specific problem
  – Leads to high qual solutions
• Provides a common vocabulary for communication to other programmers
What are Design Patterns?

- “a general solution to a recurring engineering problem”
- “architectural pattern” first used by architect Christopher Alexander to denote common design decision that have been used by architects and engineers to realize buildings and constructions in general
- Software Patterns popularised by the “Gang of four” in 1994 book
Four Design Spaces

- Finding Concurrency
- Algorithm Structure: Patterns to map tasks to processes
- Supporting Structures: Code and data structure patterns
- Implementation Mechanisms: Low level patterns used to write parallel progs
Finding Concurrency

Figure based on ref [3]

**Lecture 4:** Concurrency Design Patterns
Task vs Data vs Pipeline Decomposition

- Task Decomposition
- Data Decomposition
- Pipeline Decomposition
Task Decomposition

- Often harder than identifying data parallelism.
- Start with having a good understanding of the problem being solved.
- Find independent coarse-grained computations that are inherent to the algorithm.
  - Ideally pick "natural" decompositions that fall out of the processing rather than forcing some pattern on them.
  - These will form a sequence of statements that operate together as a group.
    - e.g., loops, function calls => start by examining them.
- Most tasks follow from the way the programmer thinks of a problem.
- NB: often easier to start with too many tasks and fuse them later rather than trying to split.
- Task choice will impact s/w engineering decisions and implementation.
Task Decomposition Considerations

• **Flexibility** (in terms of number + size of tasks generated)
  – Tasks should not be tied to a specific (hardware) architecture
  – Fixed tasks vs parameterised tasks
    • e.g. parameterised loop is better as more reusable for different numbers of threads

• **Efficiency**
  – Tasks need to have enough work to offset the costs of creating and maintaining them
  – Tasks should be sufficiently independent so that managing the dependencies doesn't become a bottleneck eg if each task needs to talk to others => need to amortize the commo tasks

• **Simplicity**
  – If you cannot understand the code, you cannot debug or maintain or reuse
Data Decomposition

• Key: Find where the **same operations** are applied to **different data** again and again
• Not really separate from task decomposition
  – often data decomposition is dictated by the task decomposition you select (and vice versa)
  ⇒ have to decide will you do this or task decomp first
• Data decomposition is first when:
  – Main computation is organised around manipulation of a large data structure
  – Similar operations are applied to different parts of the data structure
Common Data Decompositions

• Array data structures
  – Decompose along rows, cols, blocks
• Recursive data structure eg tree
  – Subdivide into left/right
• Need to work out how to recombine the results
• This pattern is particularly useful when the application exhibits locality of reference
  – i.e., when processors/threads can refer to their own partition only and need little or no communication with other processors/threads
Example

- Matrix-vector product $Ax = b$
- Matrix $A[] []$ is partitioned into $P$ horizontal blocks
- Each processor
  - operates on one block of $A[] []$ and on a full copy of $x[]$
  - computes a portion of the result $b[]$

Figure from ref [4]
Data Decomposition Considerations

- **Flexibility** (in terms of number + size of data chunks generated)
  - Should support a wide range of execution types

- **Efficiency**
  - Data chunks should generate around the same amount of work to process them (for load balancing)
  - E.g. (from [4]) Mandelbrot set calculation:
    - A regular partitioning can result in uneven load distribution
      - Black pixels require `maxit` iterations
      - Other pixels require fewer iterations

- **Simplicity**
  - Complex data compositions can be difficult to manage + debug
Pipeline Decomposition

• Use where data is flowing through a sequence of stages
  – Assembly line is a good analogy
  – eg instruction pipeline in modern CPUs
  – eg pipes in Linux
  – E.g. signal processing

Figure from ref [3]
Summary

• Concurrent and parallel programming has many pitfalls
• Java language gives native support for shared memory concurrency
• Design Patterns provide support for
  – Analysing and dividing up problem
  – Ways to structure solution
  – Common data structures, communication methods, task dispatchers
• Keep your design as serial and simple as possible such that it can do the job