Concurrent and Distributed Programming (CA4006)

A Design Pattern Based Approach to Concurrency and Parallelization (Part 1)

2022/2023

Graham Healy

These course slides are partly adapted from the original course slides prepared by:
Dr Martin Crane, Dr Rob Brennan and Dr Takfarinas Saber

Lecture Contents

- Focus: How do we get this stuff into code?
- Design considerations
- Steps in Designing concurrent/parallel programs
- The Parallel/Concurrent Design Pattern Spaces
- Java concurrency support

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 (e.g., Clusters)
  - Needs message passing for communication
  - MPI, (Java + RMI, web services, or REST)

Writing Concurrent Code

1. Identify concurrency in task
    - Do this on a piece of paper
2. Expose the concurrency when writing the task
    - Choose a programming model and language that allow you to express this concurrency
3. Exploit the concurrency
    - Carefully choose a language & hardware that facilitate taking advantage of the concurrency

Value of a programming model is judged on

- **Generality**: how well a range of different problems can be expressed for a variety of different architectures
- **Performance**: how efficiently compiled programs can execute on these architectures

Concurrent Program Challenges

- Must ensure concurrent execution is safe compared to serial program, i.e.
  - Correct
  - Performance is not compromised

- Debug of these programs is much harder than serial programs
  - Need strong software engineering discipline
  - Use Design Patterns

A concurrent program must be correct under all possible interleavings.

Challenges: Assuring Safety

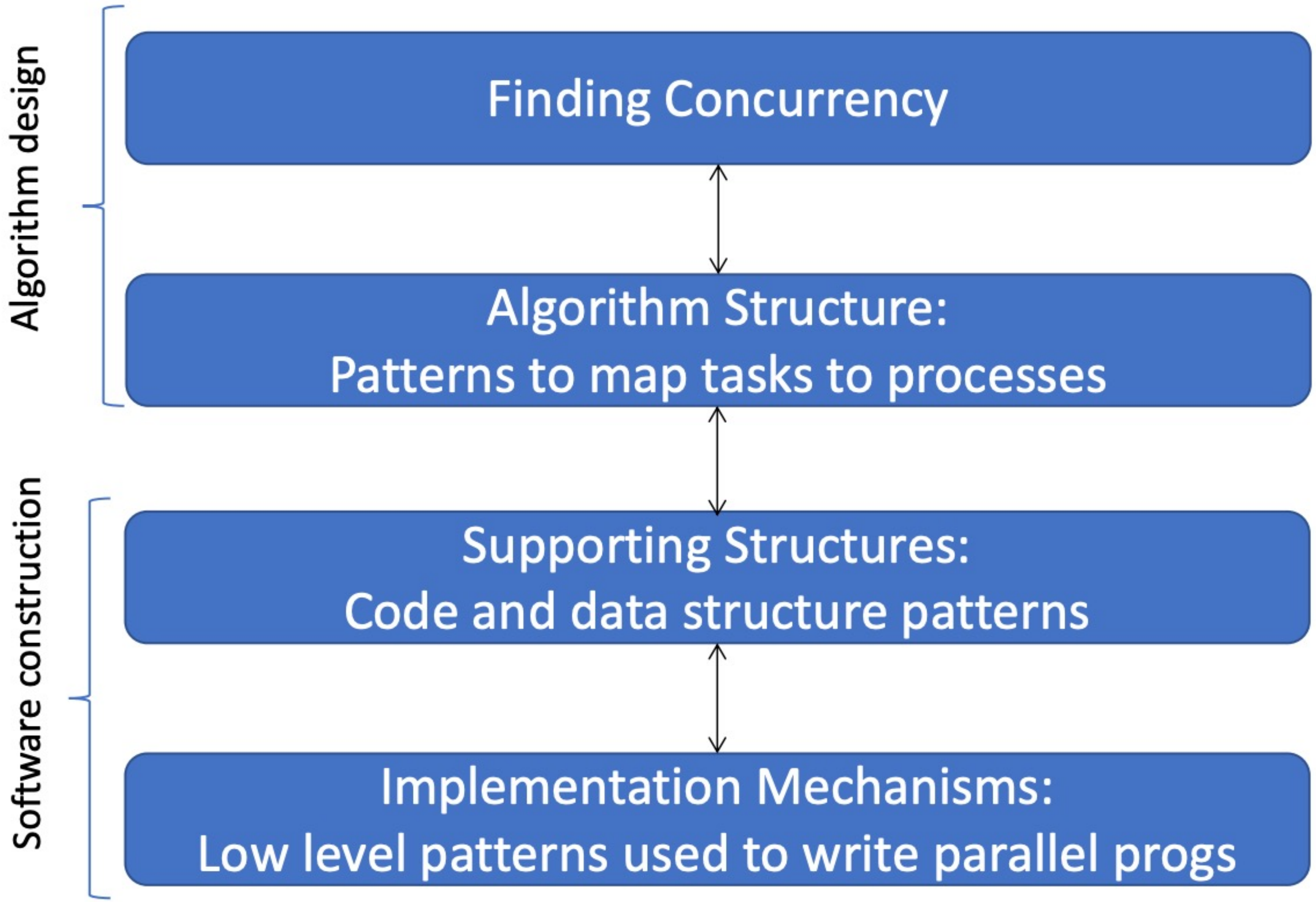
- Multiple threads can access the same resource safely only if:
  - all accesses have no effect on resource (read only)
  - Or all accesses are ***idempotent*** (i.e., produce the same result when called over and over)
  - Or only one access at a time (mutual exclusion, mutex) - allows write access

- Mutex prevents multiple threads accessing a critical block of code at the same time
  - Introduces synchronization between threads
    - But it creates potential for deadlocks
  - Wider blocks create safety but induce delays as they reduce parallelism

Challenges: Race Conditions

- Application behaviour depends on sequence or timing of processes/threads
  - E.g., accessing a buffer
- Cause:
  - Non-deterministic (timing dependent) results
  - Data corruption, crashes
- Caused by:
  - Programming errors
  - Failure to apply good lock discipline
  - Scheduler controlled interleaving of threads
  - Trying to make performance improvements to code
- They are difficult to detect, **reproduce** + eliminate
- Avoid by appropriate use of mutual exclusion

Four Design Spaces



Process for Parallelising Code

- Identify what variables need to be shared
  - These need explicit synchronization/locks
- Identify what variables need to be private

- For distributed memory (we will talk about this later), *identify which variables should be setup for reductions*

Problem Decomposition

- Identify concurrency and decide at what level to exploit it
  - Tasks, data, pipelines
- Break computation into tasks to be divided among processors
  - Number of tasks may vary with time
  - Tasks may become available dynamically
- Strive to 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 communication

- Programmers really worry about partitioning first
  - i.e., figuring out the parts of the application that they need to compose to make the application
    - 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)

Finding Concurrency

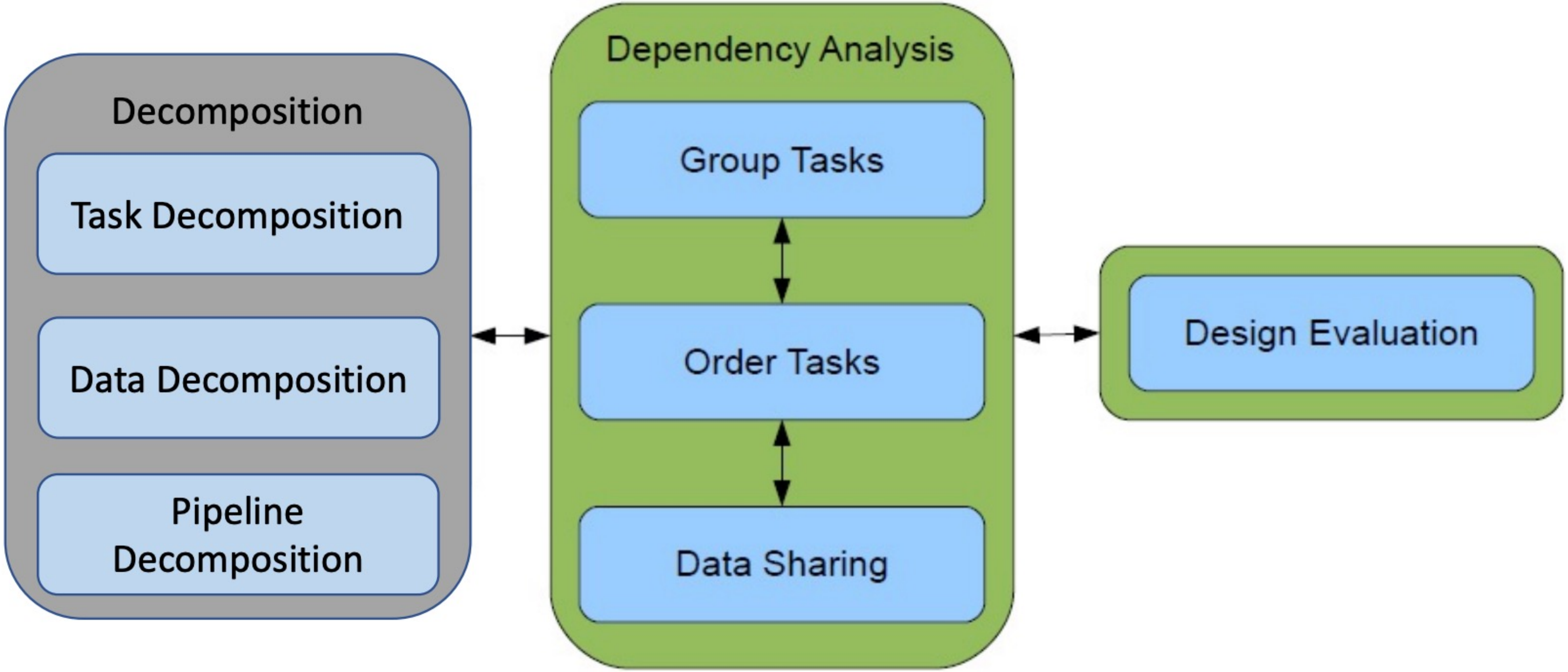


Figure based on ref [3]

Task Decomposition

- Task decomposition is often harder than identifying data parallelism
- Requires a good understanding of the problem
- Goal, find independent coarse-grained computations/activities 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., start by examining loops and function calls
- Most tasks follow from the way the programmer thinks of a problem
    - Often, it is easier to start with too many tasks and fuse them later rather than trying to split
- Task choices will impact software 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
    - E.g., if each task needs to talk to others => need to amortize communication 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)

- 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 e.g., 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[]`

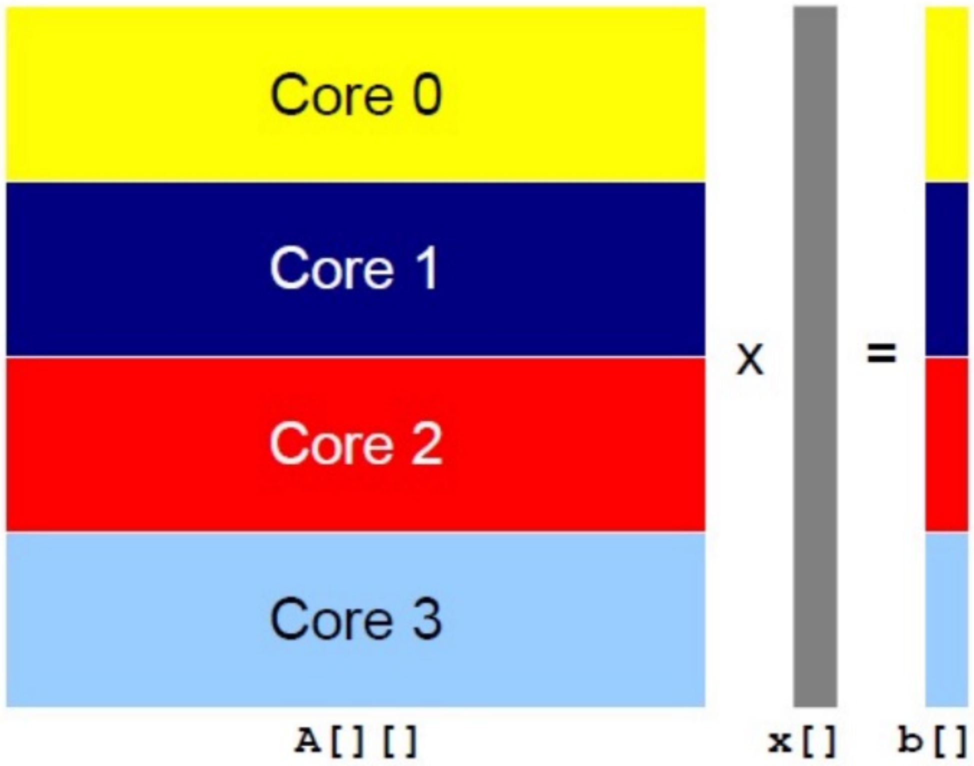
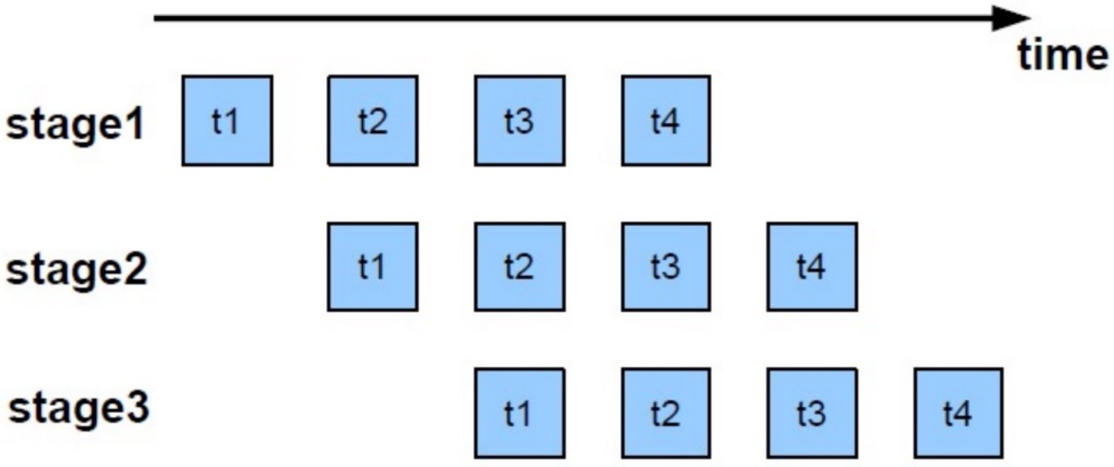


Figure based on ref [4]

Pipeline Decomposition

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



Speech recognition:
1. Discrete Fourier Transform (DFT)
2. manipulation e.g. log
3. Inverse DFT
4. Truncate 'Cepstrum' ..

Figure based on ref [3]

Summary

- Concurrent and parallel programming has many pitfalls, but…
- 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

Concurrent Java – Basic Constructs

Recap on Java Threads (/1)

- Java thread is a lightweight process with 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:

  Thread myProcess = new Thread();
- New thread is started by executing:

  MyProcess.start();
- The *start* method invokes a *run* method in the thread
- As *run* method is undefined as yet, the code above does nothing

Recap on Threads (/2)

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

```
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 you don't need a ref to new thread omit p and simply write:

  new myProcess().start();

Recap on Threads (/3)

As well as extending Thread class, you 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:

```
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 a thread has nothing immediate to do (e.g., it updates screen regularly) you should suspend it by putting it to sleep
- 2 flavours of join( ) method – wait forever or for a specific times (milli, nano)
- **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 thread will block until the currently running thread instance has finished executing (or time elapses if provided):

```
try {
        myThread.join (1000); // wait for 1 sec
} catch (InterruptedException e ) {}
```

In Java, Threads are Everywhere

- Every Java application uses threads:
  - When the JVM starts, it creates:
    - threads for JVM housekeeping tasks (garbage collection, finalization)
    - and a main thread for running the main method

- Timer creates threads for executing deferred tasks

Example: Non-thread-safe Java

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

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

Listing 1.1. Non-thread-safe sequence generator.

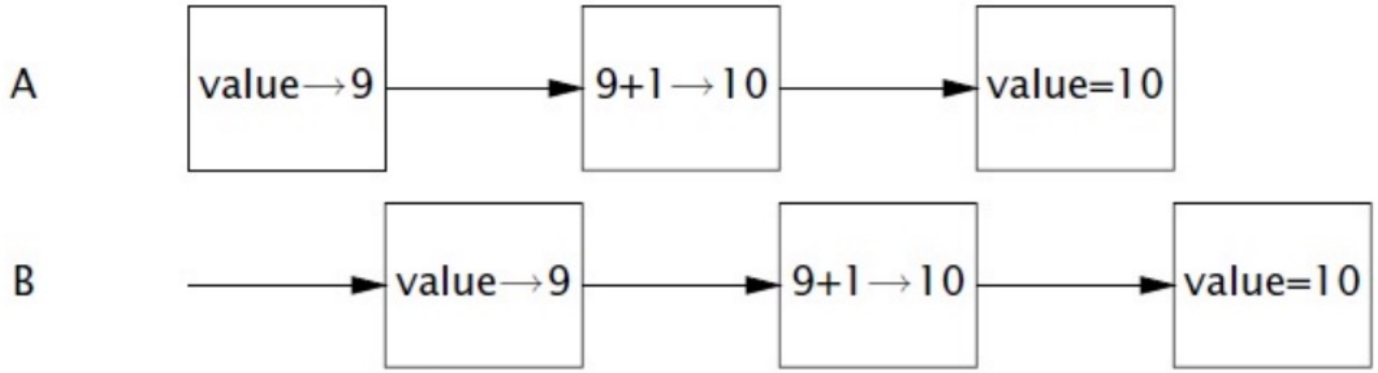


Figure 1.1. Unlucky execution of UnsafeSequence.getNext.

Figure based on ref [6]

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
  - Or make the state variable immutable (does not change)
  - 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
- **Rule of Thumb**:
  - Stateless objects are always thread-safe
  - Immutable objects are always thread-safe
  - Atomic state variable updates are safe

    If we've 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 (with Locks):**
  - Supported in JVM via object locks (a.k.a., 'mutex')
  - Enables multiple threads to independently work on shared data without interfering with each other

- **Cooperation (with Monitors):**
  - Supported in JVM via the **synchronized** keyword and ***wait***(), ***notify***() methods
  - Enables threads to work together towards a common goal

Mutual Exclusion in Java: Synchronization

- Conceptually threads in Java execute concurrently,
  - hence they could simultaneously access shared variables (i.e., A Race Condition)
- To prevent race condition 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 class (on the right) will potentially have problems if its update method is executed by several threads concurrently

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

Mutual Exclusion in Java: Synchronization (/2)

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

- To preserve state consistency, we update related state variables in a single atomic operation
- There is 1 default lock created 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)

```
class ExclusionByGroup {
        private int data = 0;
        public void update ( ) {
                synchronized (this) { // lock this object;
                        data ++

                        // for these 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?

```
@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

Figure based on ref [6]

Rationale

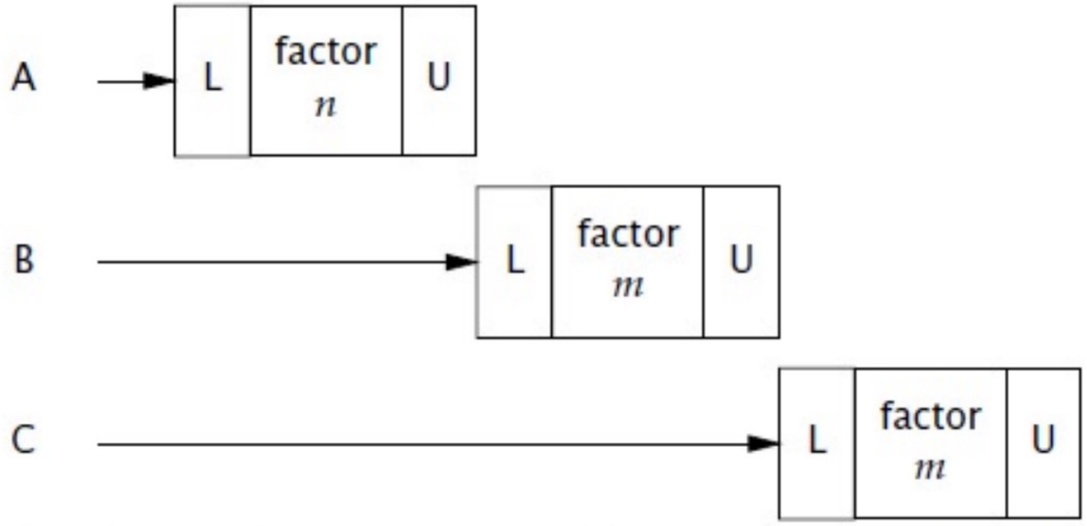


FIGURE 2.1. Poor concurrency of SynchronizedFactorizer.

- Caching required shared state
- Protected it with coarse-grained mutex => safe
- But if servlet is busy, new customers (threads) must wait
- Even with multiple CPUs, all threads must wait
  => we should try to exclude from synchronized blocks long- running operations (e.g. I/O) that do not affect shared state

Figure based on 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 based on ref [6]

```java
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

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/writes must use the same lock - read and write
  - For convenience Java creates one intrinsic lock per object so you don't have to explicitly create lock objects ...
- 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 –** e.g. be cognizant to use synchronized getter/setter methods with private variables
- It is **up to you** to construct locking protocols or synchronization policies that let you access a shared state safely, and to use them consistently throughout your program
  - See Design Patterns

References

1. Timothy Mattson , Beverly Sanders , Berna Massingill, Patterns for parallel programming, Addison-Wesley Professional, 2004. ISBN-13: 978-0321228116
2. MIT 6.189 Multicore Programming Primer, IAP 2007
3. Gethin Williams, Patterns for parallel programming lecture notes,2010 https://www.researchgate.net/publication/234826291_Patterns_for_Parallel_Programming
4. Moreno Marzolla, Parallel Programming Patterns, 2018, https://www.moreno.marzolla.name/teaching/HPC/L03- patterns.pdf
5. Bill Venners, Inside the Java Virtual Machine,Book, https://www.artima.com/insidejvm
6. Brian Goetz, Java Concurrency in Practice, ISBN: 0321349601

Concurrent and Distributed Programming (CA4006)

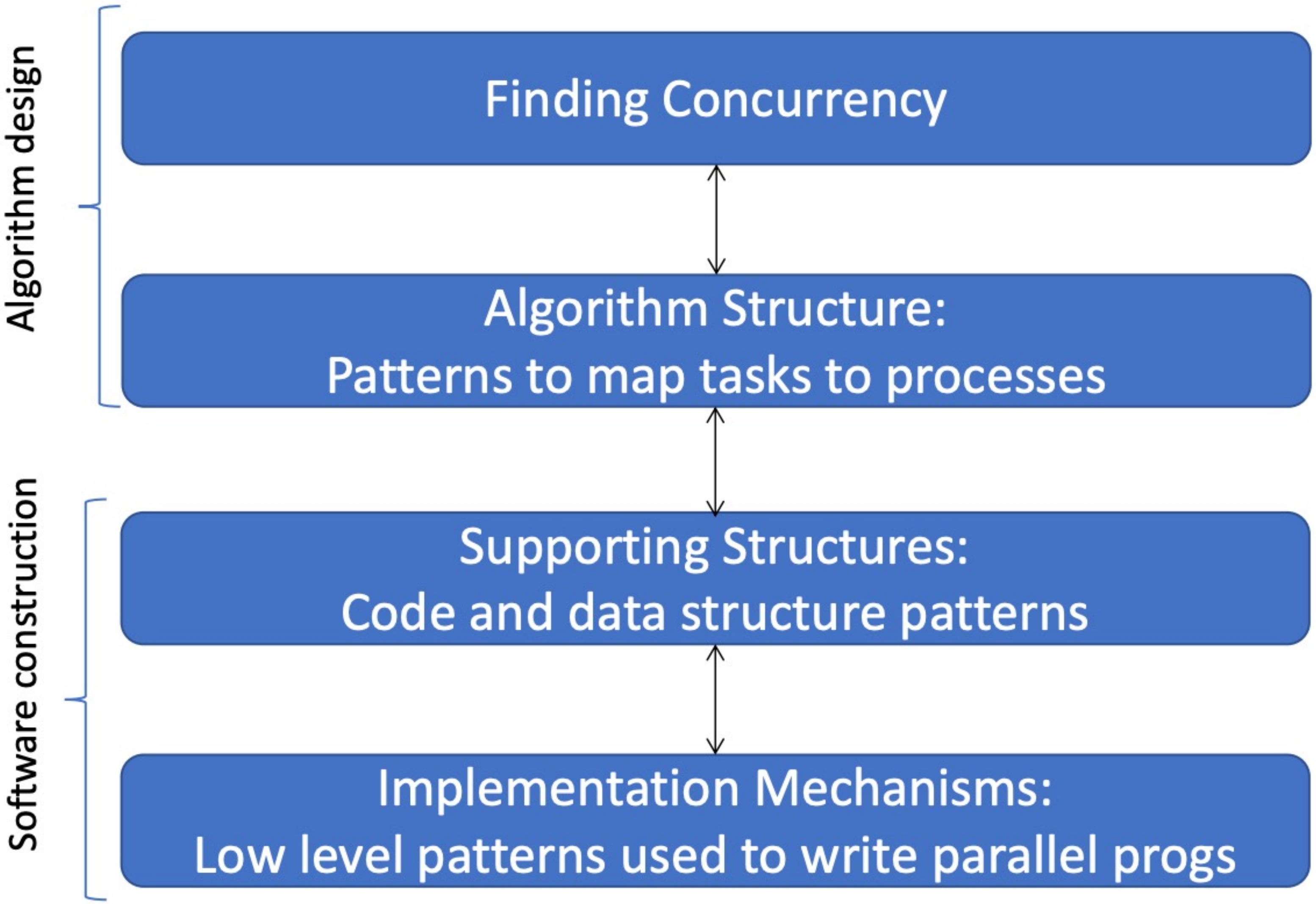
A Design Pattern Based Approach to Concurrency and Parallelization (Part 2)

2022/2023

Graham Healy

These course slides are partly adapted from the original course slides prepared by:
Dr Martin Crane, Dr Rob Brennan and Dr Takfarinas Saber

Four Design Spaces



Finding Concurrency

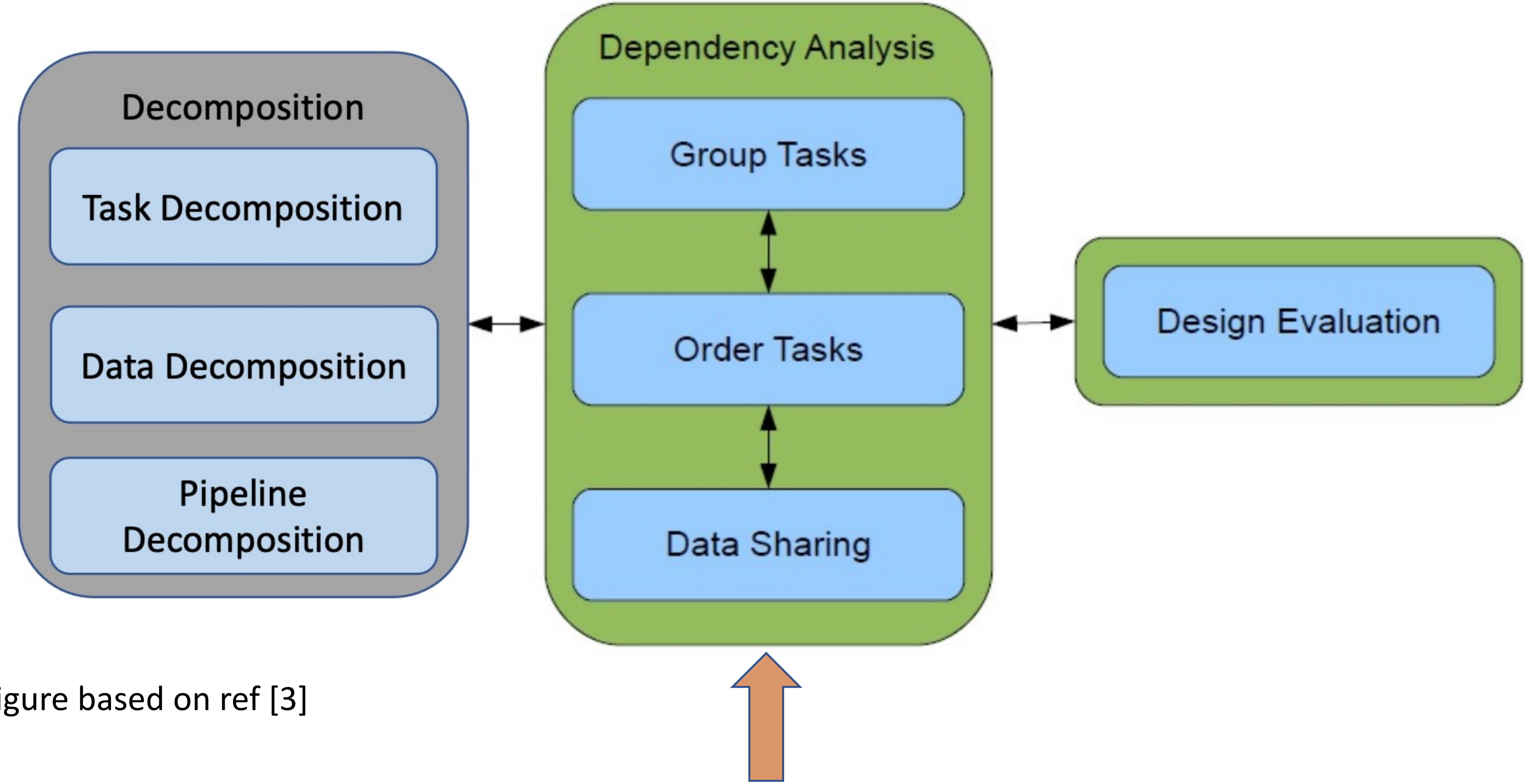


Figure based on ref [3]

Finding Relationships between Concurrent Tasks

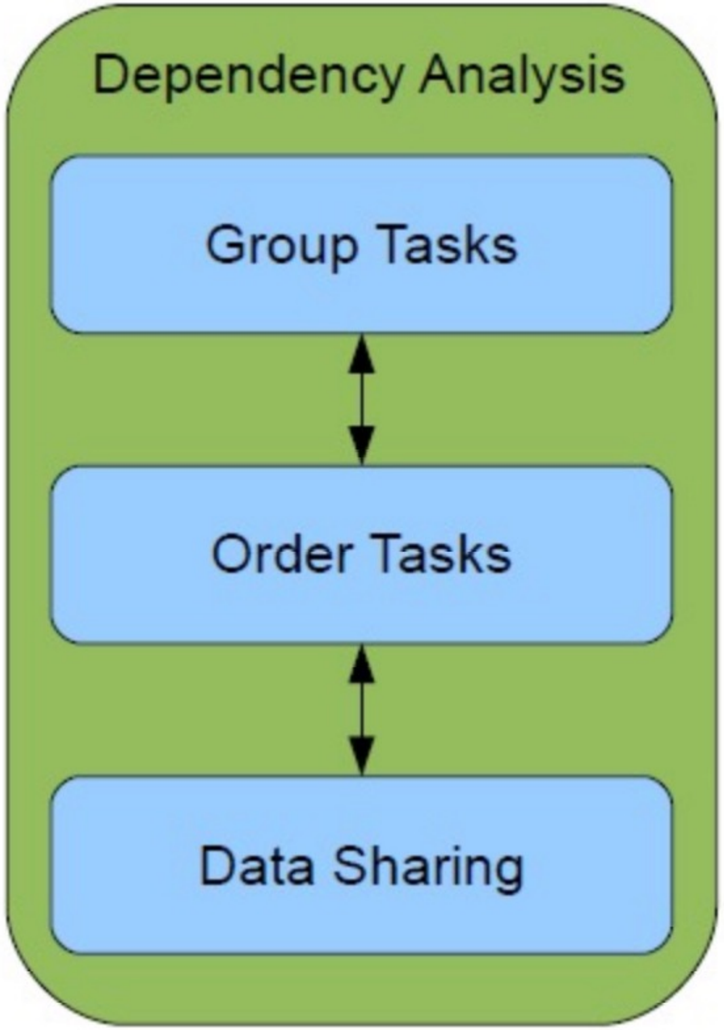


Figure based on ref [6]

Defining Dependencies

- To what extent must data be shared between tasks?
- To what extent do tasks depend on the results of other tasks?
  - E.g., cell boundaries
- What operations are used?
  - Read
  - Write
  - Accumulate
- Create a block diagram of tasks, data structures, and their connections

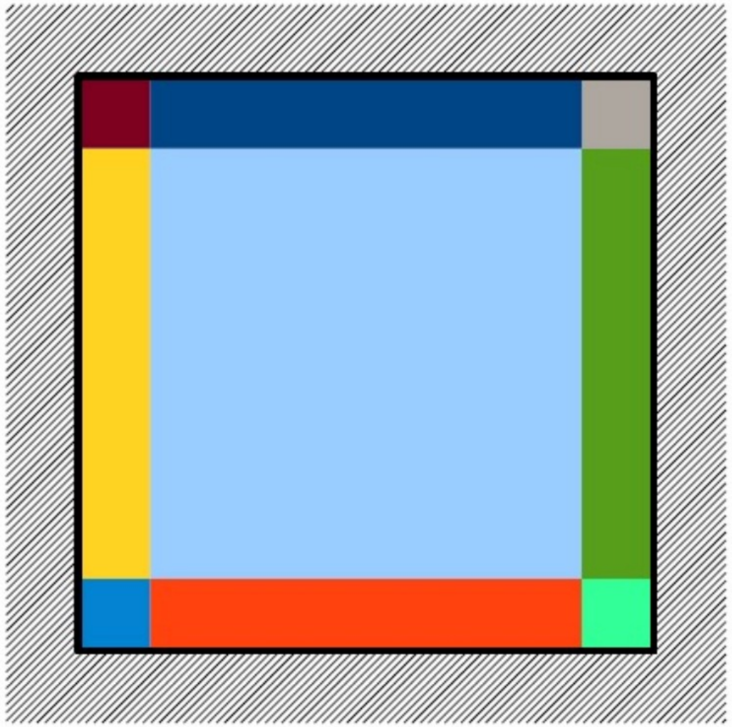


Figure based on ref [4]

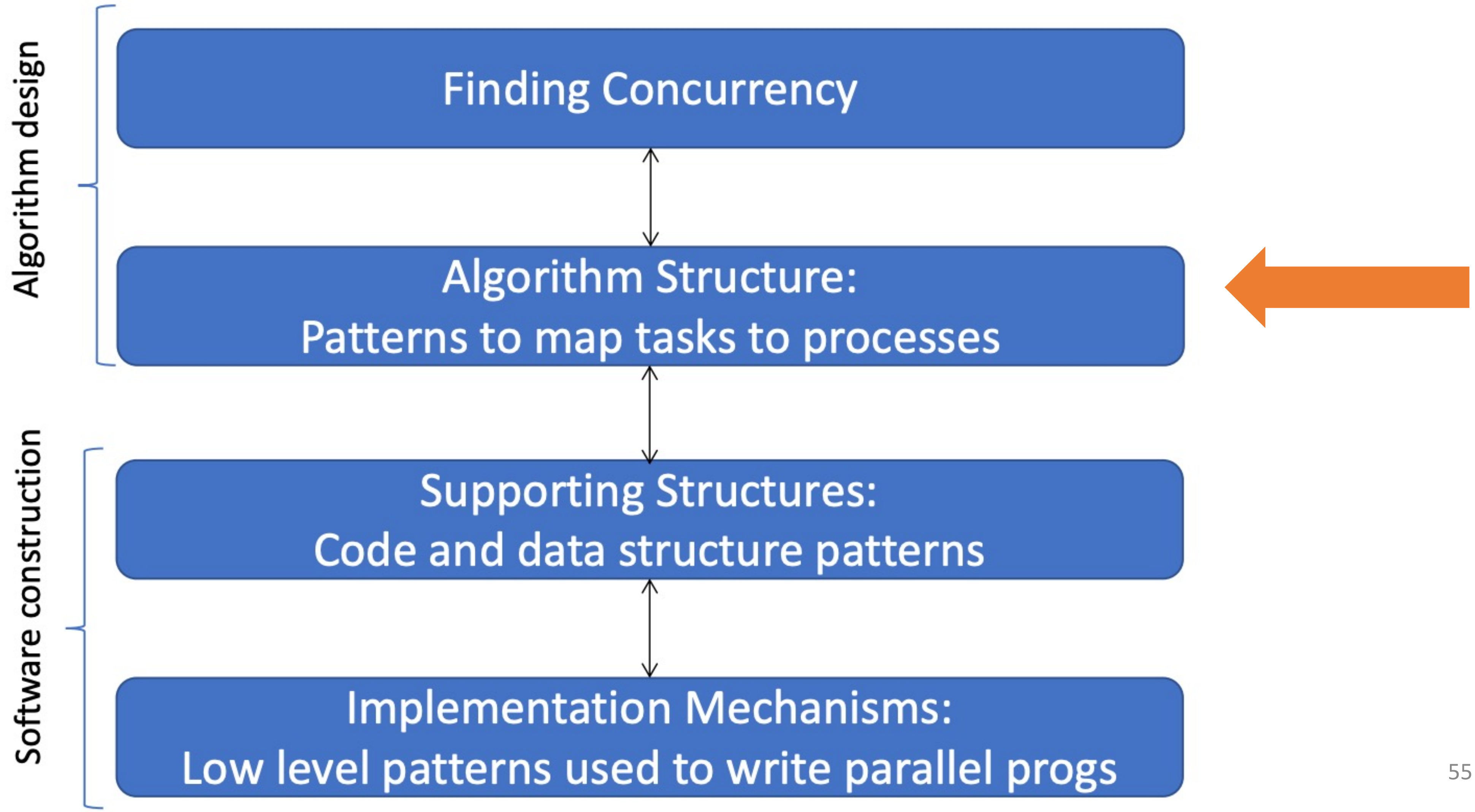
Dependency Analysis

- Given 2 tasks how can you determine if they can run in parallel?

- Tasks T1, T2 may be made parallel if and only if (iff)
  - All data consumed by T1 is not output by T2
  - All data consumed by T2 is not output by T1
  - Outputs from T1, T2 do not overlap

- Compilers can exploit this for automatic parallelization

Four Design Spaces



Mapping Tasks to Units of Execution (i.e. Threads/Processes)

- First Considerations
  - How many threads can your platform support? (10s -1000s?)
  - Cost of sharing info between execution units (**overhead**)
- Avoid over-constraining the implementation
  - Ideally should support multiple hw architectures
- Major organizing principles
  - By tasks
  - By data decomposition
  - By flow of data

Algorithm Structure Design Space

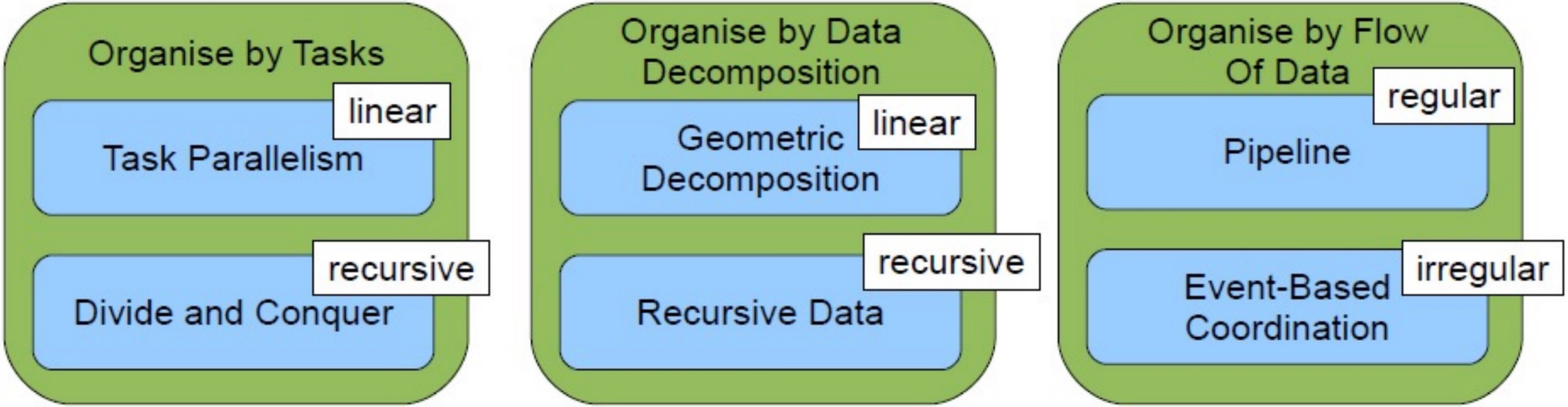


Figure based on ref [3]

Organising by Tasks

First decide on the type of problem

- If recursive => use Divide + Conquer patterns
- If not recursive => use Task Parallelism patterns

Task Parallelism Patterns

- E.g., Ray tracing, molecular dynamics
- Each task is associated with an iteration of a loop
- Often know all the tasks at the start of a computation
- All tasks may not need to complete to arrive at a solution
- Can be tricky to determine when can terminate

Divide + Conquer

- E.g., Merge/Sort
- Need to spit work and recombine the results
- Sub-problems may not be uniform
- May require dynamic load balancing, i.e., work re-distribution during execution

Organising by Data

- If you see that code has operations on a central data structure ...

- 2 types of decompositions:
  - **Arrays and linear data structure patterns (geometric decomp)**
    - E.g., gravitational body simulator, call forces between pairs of objects and update accelerations
  - **Recursive data structures patterns**
    - data in list, tree, graph
- Sometimes it looks like a sequential approach is the only one
  - It could be, we can split the task such that you end up doing more work but you still end up with a faster solution
    - Requires you to think about problem differently
  - Most strategies on this pattern trade off increase in total work for faster execution time.

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[]`**

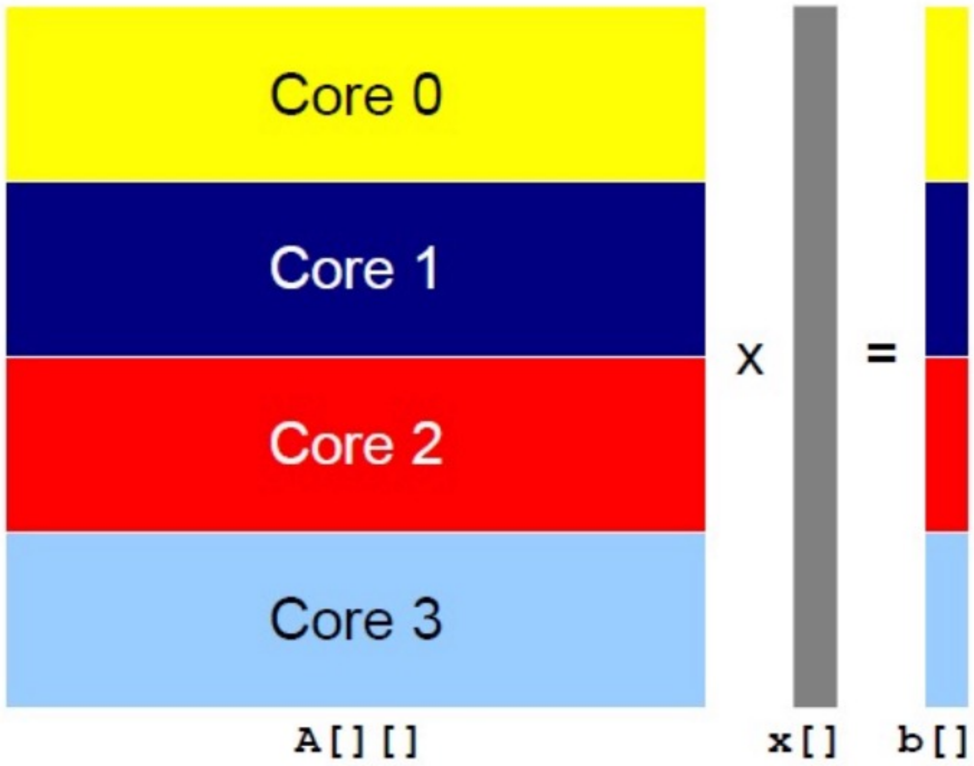
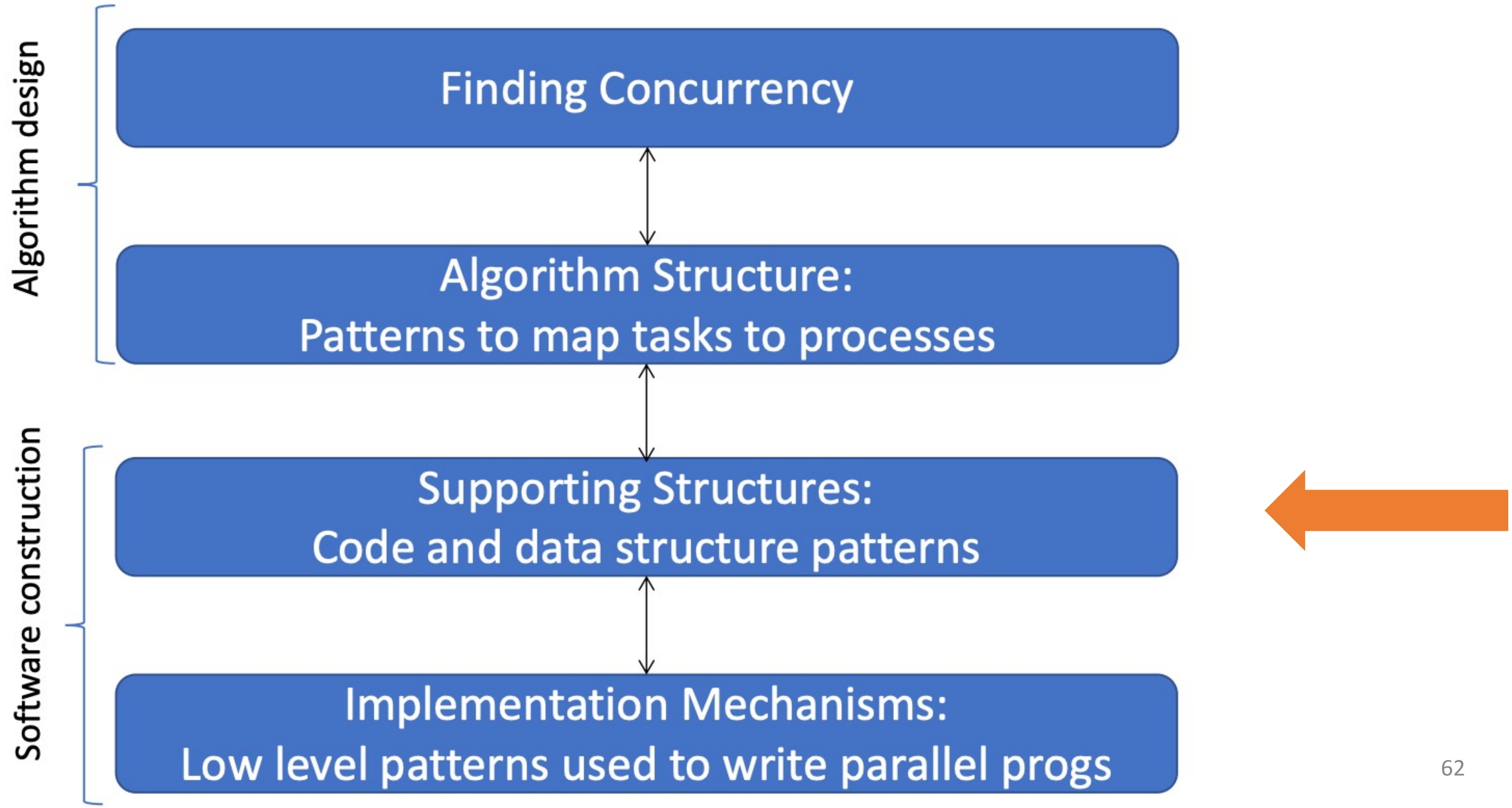


Figure based on ref [4]

Four Design Spaces



Supporting Structures

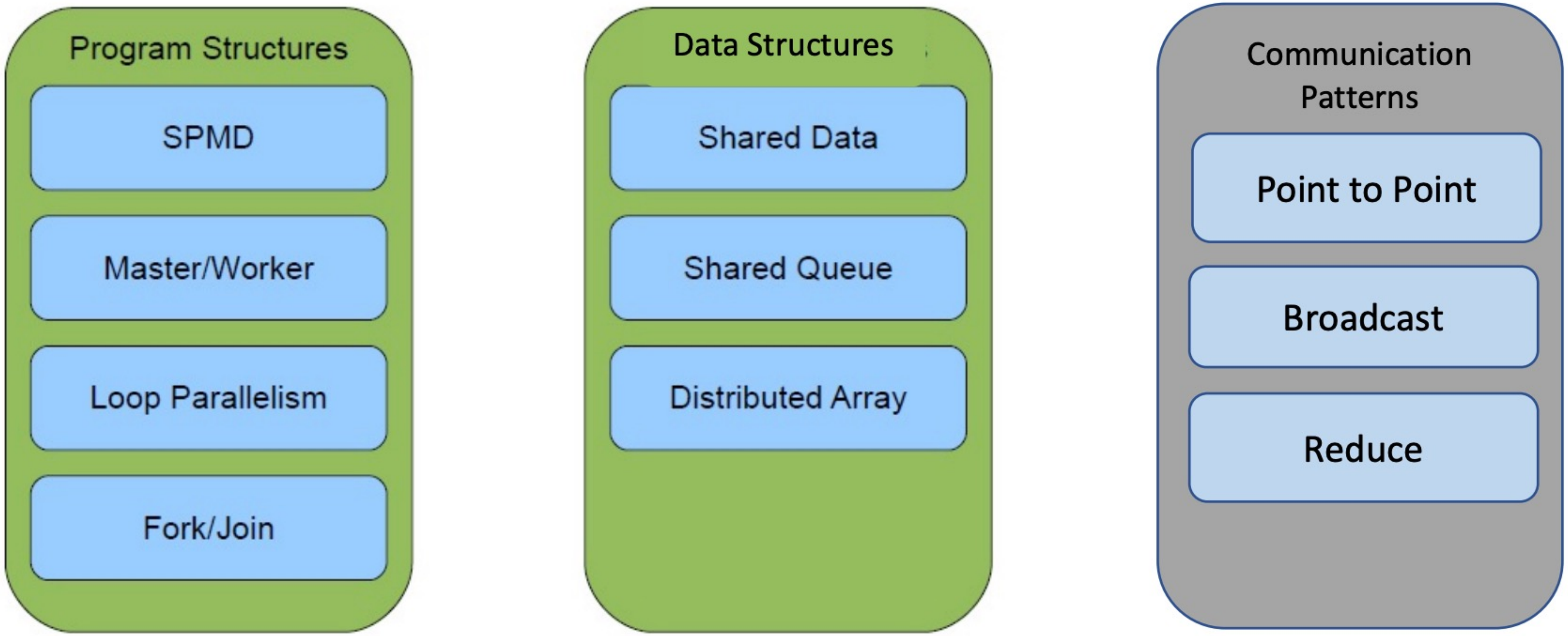


Figure based on ref [3]

Supporting Structures with Example Technologies

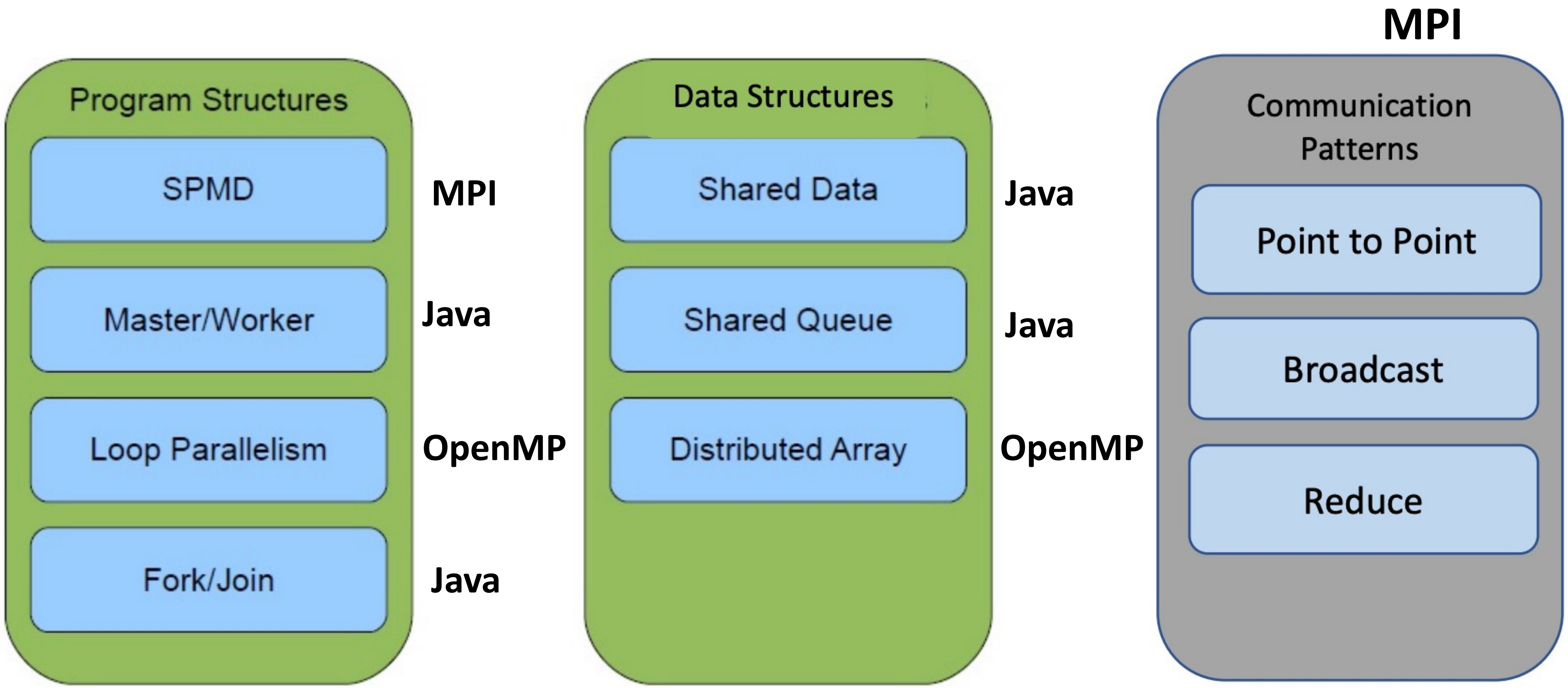


Figure based on ref [3]

Note: Patterns generally apply to multiple technologies, here we just show the ones we use for illustration purposes in this course

Shared Queue

- How can concurrently running units of execution efficiently share a queue data structure without race condition?
  - E.g., Task queue is commonly used as a scheduling and load-balancing framework, with the **Master/Worker pattern (a.k.a. Master-Slave or Map-Reduce pattern)**

Solution

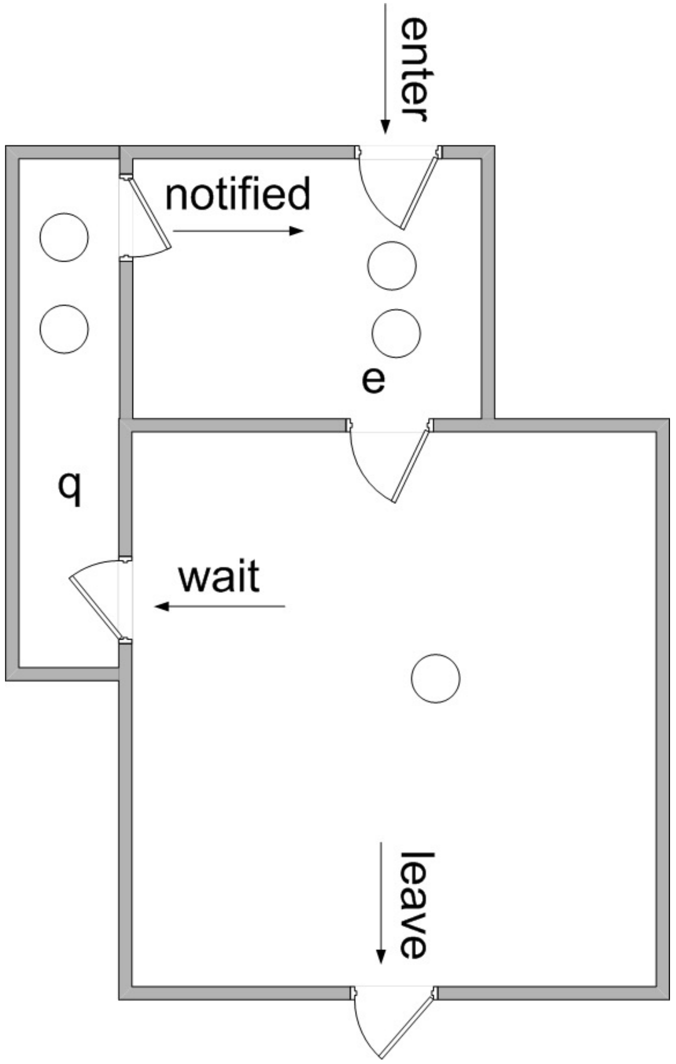
1. Define an **abstract data type (ADT)/interface**: ADT values are ordered lists of objects of some type (e.g., task IDs)

   The operations on the queue are put (or enqueue) and take (or dequeue)
2. Define a concurrency control protocol to ensure safe access to queue

   simplest solution is to make **all operations on the ADT exclude each other**
3. ***May*** also consider using distributed shared queues as centralized queues are often a bottleneck

Quick recap: Monitors in Java

- Java implements a slimmed down version of monitors
- Java's monitor supports two kinds of thread synchronization:
    - mutual exclusion and cooperation



Cooperation:

- Supported in JVM via the **wait()**, **notify()** methods of class **Object**,
- Enables threads to work together towards a common goal.

Image: https://en.wikipedia.org/wiki/Monitor_(synchronization)#/media/File:Monitor_(synchronization)-Java.png

Monitors in Java: Condition Variables

- **Condition variables**: there is one implicitly declared for each synchronised object
  - Java's wait() & notify() can only be executed in synchronized code parts (when object is locked):
- **wait**():
  - releases object lock, suspending the executing thread in a FIFO delay queue (one per object)
  - thus gets it to yield the monitor & sleep until some thread enters monitor and calls **notify**()
- **notify**():
  - wakes thread at the front of object's delay queue
  - **notify**() has signal-and-continue semantics, so thread calling **notify**() still holds the object lock
  - Awakened thread goes later when it reacquires the object lock
- Java has notifyAll(), waking all threads blocked on same object

Monitors in Java: Example 1: Queue Class

wait() & **notify**() in Java are used in **Queue** implementation:

```
/**
One thread calls push() to put an object on the queue. Another calls pop() to
* get an object off the queue. If there is none, pop() waits until there is
* using wait()/notify(). wait() and notify() must be used within a synchronized
* method or block. */

import java.util.*;

public class Queue {
        private LinkedList q = new LinkedList(); // Where objects are stored

        public synchronized void push(Object o) {
                q.add(o); // Append the object at end of the list
                this.notify(); // Tell waiting threads data is ready
        }

        public synchronized Object pop() {
                while(q.size() == 0) {
                        try { this.wait(); }
                        catch (InterruptedException e) { /* Ignore this exception */ }
                }

                return q.remove(0);
        }
}
```

Aside Example 2: Readers/Writers Class (/2)

```
class ReadersWriters {
        private int data = 0; // our database
        private int nr = 0;

        private synchronized void startRead(){
                nr++;
        }

        private synchronized void endRead(){
                nr--;
                if (nr == 0)
                        notify(); // wake a
        }                           //waiting writer

        public void read ( ) {
                startRead ( );

                System.out.println("read"+data);
                endRead ( );

        }
```

```
public synchronized void write ( ) {
        while (nr > 0)
                try {
                        wait ( ); //wait if any
                        //active readers }
                catch (InterruptedException ex){
                        return;
                }

                data++;
                System.out.println("write"+data);

                notify(); // wake a waiting writer

        }
}
```

Aside Example 2: Readers/Writers Class (/2)

```
class Reader implements Runnable {
        int rounds;
        ReadersWriters RW;

        Reader(int rounds, ReadersWriters RW) {
                this.rounds = rounds;
                this.RW = RW;
        }

        public void run ( ){
                for (int i = 0; i < rounds; i++)
                        RW.read ( );
        }

}
```

```
class Writer implements Runnable {
        int rounds;
        ReadersWriters RW;

        Writer(int rounds, ReadersWriters RW) {
                this.rounds = rounds;
                this.RW = RW;
        }

        public void run ( ){
                for (int i = 0; i < rounds; i++)
                RW.write ( );
        }
}
```

This is the Reader Preference Solution.

```
class RWProblem {
        static ReadersWriters RW = new ReadersWriters ( );

        public static void main(String[] args){
                int rounds = Integer.parseInt
                                        (args[0], 10);

                new Thread(new Reader(rounds, RW)).start ( );
                new Thread(new Writer(rounds, RW)).start ( );
        }
}
```

Master-Worker Pattern

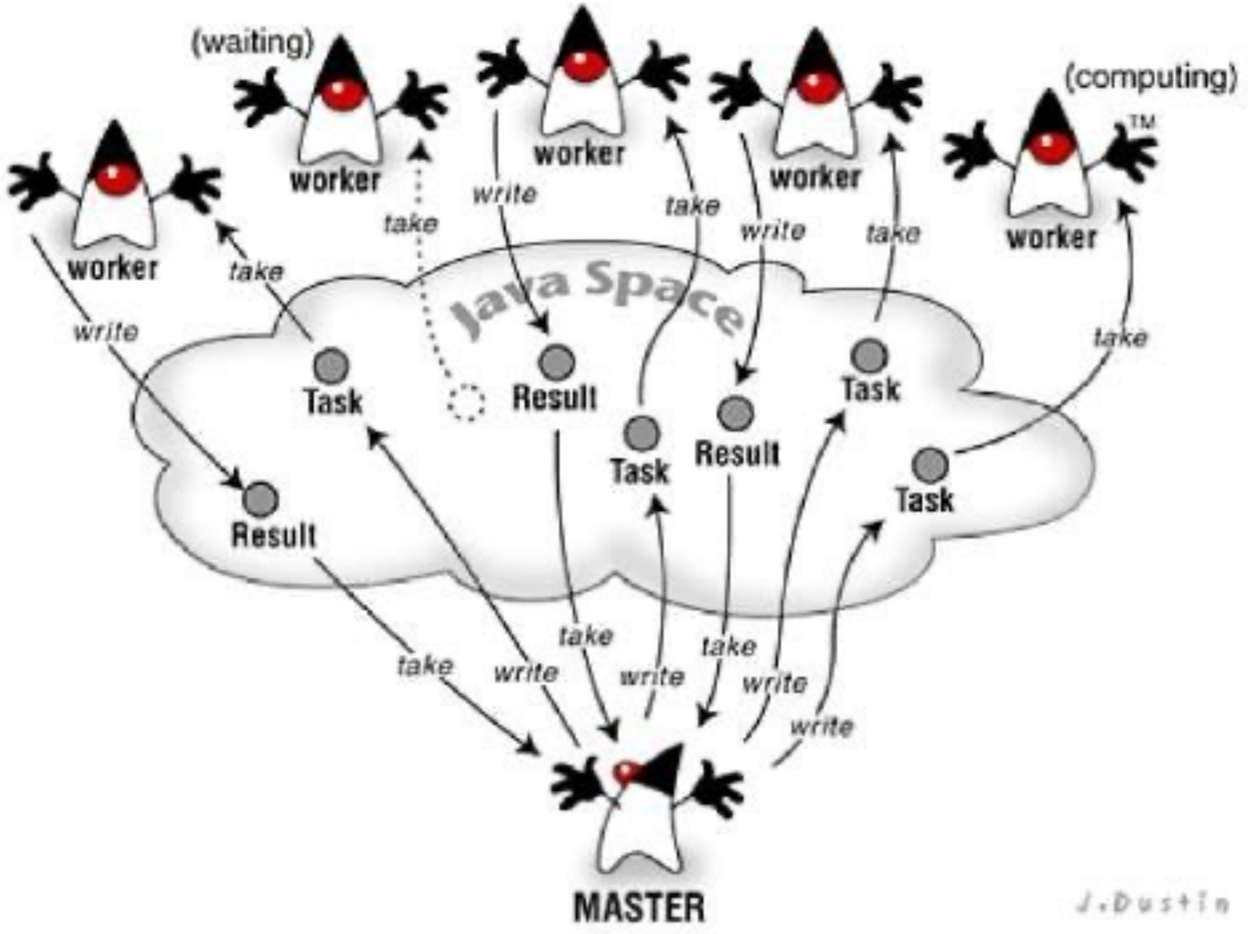


Image: https://docs.gigaspaces.com/sbp/master-worker-pattern.html

Master-Worker Pattern

- Suitable when you have a lot of independent tasks
- Master works to distribute the work
  - Sometimes workers can also send work (information about new tasks) back to master if they "discover" it during their execution
- Ideal when:
  - tasks vary in nature/load
  - problems are embarrassingly parallel (scientific computing)
- Challenge:
  - How do you determine when the whole problem is complete?
    - Often you can accept an answer within some range of error, when you are there do you stop current threads or let them complete?

Pre-History of Executors

- As seen, one method of creating a multithreaded application is to implement **Runnable**
- In **J2SE 5.0**, this became the *preferred* means (using package java.lang)
- Built-in methods and classes are used to create Threads that execute the **Runnables**
- As also seen, the **Runnable** interface declares a single method named **run**
- **Runnables** are executed by an object of a class that implements the **Executor** interface
- This can be found in package **java.util.concurrent**

Using Executors

- Let's stop thinking about concurrency in terms of protecting shared resources
  - instead think of tasks (logical units of work) to be run
  - e.g. Master-Worker
- Seen already how to create multiple threads and coordinate them
  - via synchronized methods and blocks, as well as via Lock objects
- Cannot simply assume 1 thread/task, practical drawbacks
  - Thread creation overhead
  - Resource consumption e.g., quickly run out of memory for 1000s of threads
  - Stability: platform will eventually run out of threads, dealing with that is risky
- There are 2 mechanisms in Java
  - **Executor** Interface and Thread Pools
  - **Fork/Join** Framework

Executors: Executor Interface & Thread Pools

- **java.util.concurrent** package provides 3 executor interfaces:
  - **Executor**: Simple interface that launches new tasks.
  - **ExecutorService**: Subinterface of Executor that adds features that help manage tasks’ lifecycle.
  - **ScheduledExecutorService**: Subinterface of **ExecutorService** supporting future and/or periodic execution of tasks.
- The **Executor** interface provides a single method, **execute**.
- For **Runnable** object **r** , Executor object e then **e.execute** (**r**); may:
  - execute a thread
  - or use an existing worker thread to run **r**
  - or with thread pools, queue **r** to wait for available worker thread.

Executors: Executor Interface & Thread Pools (/2)

- Thread pool threads execute **Runnable** objects passed to **execute**()
- The **Executor** assigns each **Runnable** to an available thread in the thread pool
- If none available, it creates one or waits for one to become available & assigns that thread the **Runnable** passed to method **execute**
- Depending on the **Executor** type, there may be a limit to the number of threads that can be created
- A subinterface of Executor (Interface **ExecutorService**) declares other methods to manage both **Executor** and task/ thread life cycle

Example: Executors

```
//From Deitel & Deitel PrintTask class sleeps a random time 0 - 5 seconds
import java.util.Random;

class PrintTask implements Runnable {
        private int sleepTime; // random sleep time for thread
        private String threadName; // name of thread
        private static Random generator = new Random(); // assign name to thread

        public PrintTask(String name) {
                threadName = name; // set name of thread
                sleepTime = generator.nextInt(5000); // random sleep 0-5 secs
        } // end PrintTask constructor

        // method run is the code to be executed by new thread
        public void run() {
                try { // put thread to sleep for sleepTime
                        System.out.printf("%s sleeps for %d ms.\n",threadName,sleepTime );
                        Thread.sleep( sleepTime ); // put thread to sleep
                } // end try
                // if thread interrupted while sleeping, print stack trace
                catch ( InterruptedException exception ) {
                        exception.printStackTrace();
                } // end catch

                // print thread name
                System.out.printf( "%s done sleeping\n", threadName );
        } // end method run
} // end class PrintTask
```

Example: Executors (/2)

- When a **PrintTask** is assigned to a processor for the first time, its **run** method begins execution
- Static method **sleep** of class **Thread** is called to place the thread into the timed waiting state
- At this point, thread loses the processor & system lets another execute
- When the thread awakens, it re-enters the runnable state
- When the **PrintTask** is assigned to a processor again, thread's name is output saying thread is done sleeping; run terminates

Example: Executors Main Code

```
//RunnableTester: Multiple threads printing at different intervals
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
        public static void main( String[] args ) {
                // create and name each runnable
                PrintTask task1 = new PrintTask( "thread1" );
                PrintTask task2 = new PrintTask( "thread2" );
                PrintTask task3 = new PrintTask( "thread3" );

                System.out.println( "Starting threads" );

                // create ExecutorService to manage threads
                ExecutorService threadExecutor = Executors.newFixedThreadPool( 3 );

                // start threads and place in runnable state
                threadExecutor.execute( task1 ); // start task1
                threadExecutor.execute( task2 ); // start task2
                threadExecutor.execute( task3 ); // start task3 thread

                Executor.shutdown(); // shutdown worker threads


                System.out.println( "Threads started, main ends\n" );

        } // end main

} // end RunnableTester
```

Example: Executors Main Code (/2)

- The code above creates three threads of execution using the **PrintTask** class
- main
  - creates & names three **PrintTask** objects
  - creates a new **ExecutorService** using method **newFixedThreadPool**() of class **Executors**, which creates a pool consisting of a fixed number (3) of threads
  - These threads are used by **threadExecutor** to run the execute method of the **Runnables**
  - If **execute()** is called and all threads in **ExecutorService** are in use, the **Runnable** will be placed in a queue
  - It is then assigned to the first thread completing its previous task

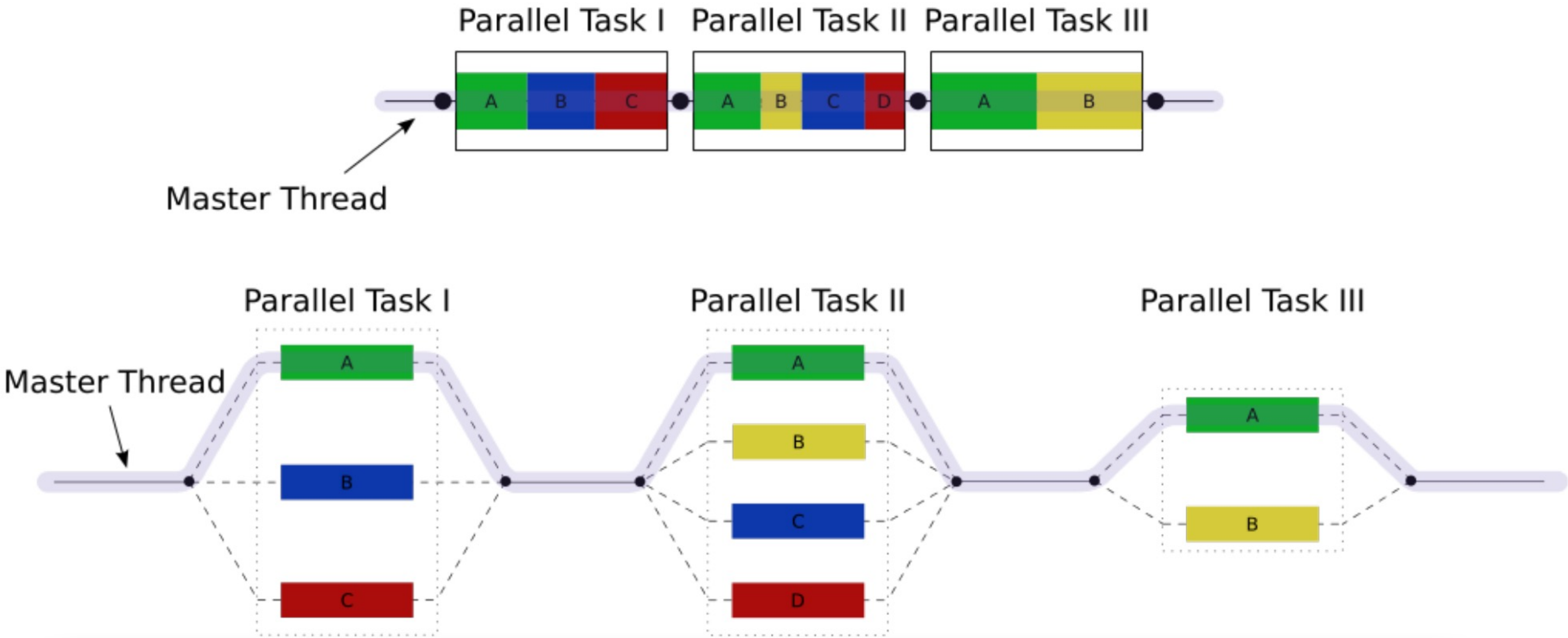
Example: Executors Main Sample Output

```
Starting threads
Threads started, main ends
thread1 sleeps for 1217 ms.
thread2 sleeps for 3989 ms.
thread3 sleeps for 662 ms.
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
```

Fork/Join Pattern

- Similar to master-worker
  - But more dynamic
- Parent = creates sub-tasks
  - i.e., children
- Tasks are created dynamically + later terminated
- Manages tasks according to their relationship
  - Parent creates tasks (fork)
  - then waits until they complete (join)
  - before continuing with the computation

The Fork-Join Pattern



Java ForkJoin Framework

- Since Java 7, the Fork/Join framework can be used to distribute threads among multiple cores
  - It’s an implementation of **ExecutorService** interface designed for work that can be broken into smaller pieces recursively.
  - Goal: use all available processors to enhance application performance
- This framework thus adopts a divide-and-conquer approach:
  - **If task can be easily solved**
    -> current thread returns its result.
  - **Otherwise -> thread divides the task into simpler tasks and forks a thread for each sub-task.**

  When all sub-tasks are done, the current thread returns its result obtained from combining the results of its sub-tasks.
- One difference between **Fork/Join** framework and **Executor** Interface is Fork/Join implements a work stealing algorithm
  - This allows idle threads to steal work from busy threads (i.e. pre-empting)

ForkJoin Framework (/2)

- A key class is the **ForkJoinPool**
  - an implementation of ExecutorService implementing work-stealing
- A **ForkJoinPool** is instantiated like so:

  numberOfCores = Runtime.getRunTime().availableProcessors( );

  ForkJoinPool pool = new ForkJoinPool(numberOfCores);
- There are 3 ways to submit tasks to a ForkJoinPool

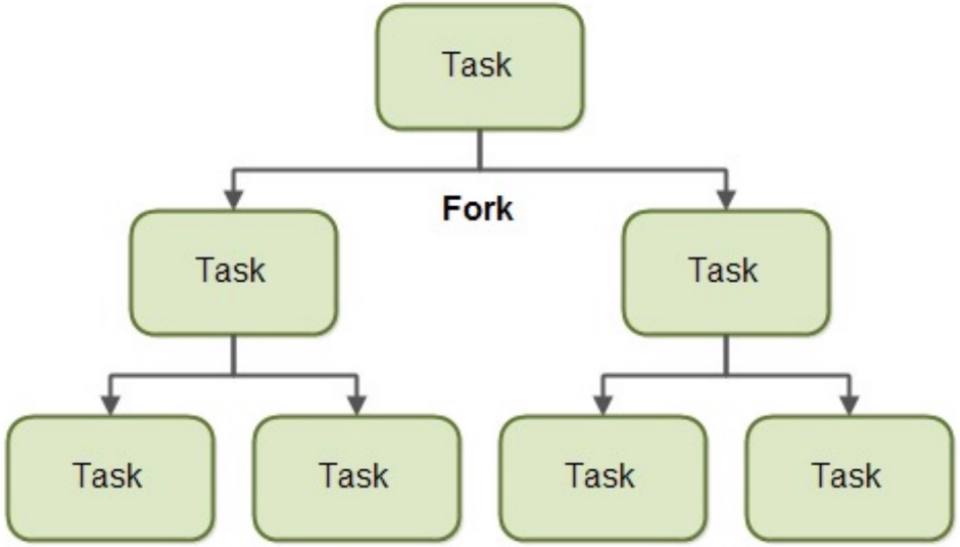
  execute() : asynchronous execution

  invoke() : synchronous execution - wait for the result

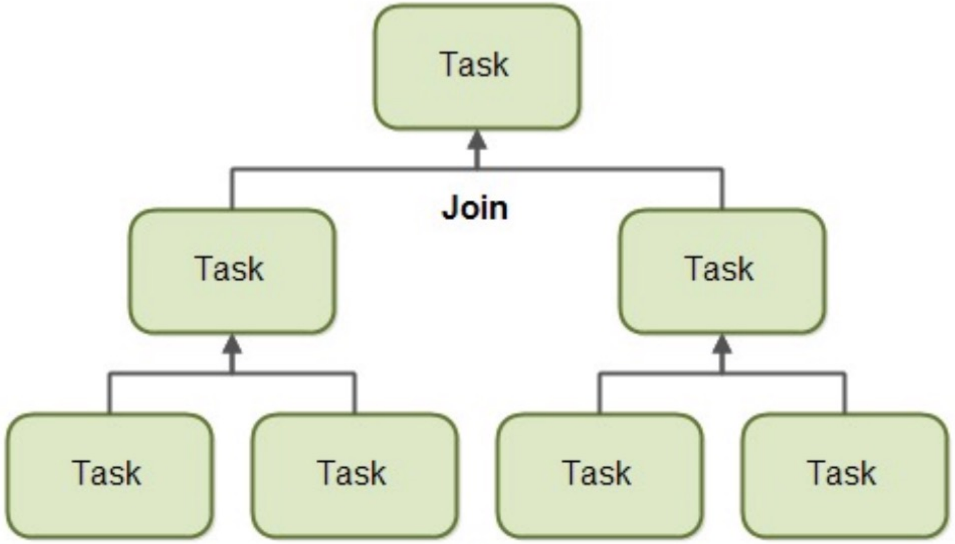
  invoke() : asynchronous execution - returns a Future object that can be used to check the status of the execution and obtain the results

ForkJoin Framework (/3)

- Thus, **ForkJoinPool** facilitates tasks to split work up into smaller tasks
  - These smaller tasks are then submitted to the **ForkJoinPool** too
  - This aspect differentiates **ForkJoinPool** from **ExecutorService**
- Task only splits itself up into subtasks if work it was given is large enough for this to make sense
  - Reason for this is the overhead to splitting up a task into subtasks
  - For small tasks this may be greater than speedup from executing subtasks concurrently



Fork



Join

ForkJoin Framework (/4)

- Submitting tasks to a **ForkJoinPool** is like submitting tasks to an **ExecutorService**
- Can submit two types of tasks
  - A task that does not return any result (aka an "action"), and
  - One which does return a result (a "task")
- These two types of tasks are represented by **RecursiveAction** and **RecursiveTask** classes, respectively.

- To use a **ForkJoinPool** to return a result:
  1. first create a subclass of **RecursiveTask<V>** for some type V
  2. In the subclass, override the **compute()** method
  3. Then you call the invoke() method on the **ForkJoinPool** passing an object of type **RecursiveTask <V>**

The use of tasks and how to submit them is summarised in the following example

Example 9: Returning a Result from a ForkJoinPool

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class Globals {
        static ForkJoinPool fjPool = new
ForkJoinPool();
}

//This is how you return a result from fjpool
class Sum extends RecursiveTask<Long> {
        static final int SEQ_LIMIT = 5000;

        int low;
        int high;
        int[] array;

        Sum(int[] arr, int lo, int hi) {
                array = arr;
                low = lo;
                high = hi;
        }
```

```
    protected Long compute() {
        // override the compute() method
        if(high - low <= SEQ_LIMIT) {
                long sum = 0;
                for(int i=low; i < high; ++i)
                        sum += array[i];
                return sum;
        }
        else {
                int mid = low + (high - low) / 2;
                Sum left = new Sum(array, low, mid);
                Sum right = new Sum(array,mid, high);
                left.fork();
                long rightAns = right.compute();
                long leftAns = left.join();
                return leftAns + rightAns;
        }
    }

    static long sumArray(int[] array) {
        return Globals.fjPool.invoke(new
                Sum(array,0,array.length));
    }

}
```

This example sums all the elements of an array, using parallelism to potentially process different 5000-element segments in parallel.

Example 9: Returning a Result from a ForkJoinPool(/2)

- Sum object gets an array & its range; **compute** sums elements in range
  - If range has **< SEQ_LIMIT** elements, use a simple for-loop
  - Else, create two **Sum** objects for problems of half the size
- Uses fork to compute left half in parallel to computing the right half, which this object does itself by calling **right.compute()**
- To get the answer for the left, it calls **left.join()**
- Create more **Sum** objects than available processors as it's framework's job to do a number of parallel tasks efficiently
- But also to schedule them well - having lots of fairly small parallel tasks can do a better job.
- Especially true if number of processors available varies during execution (e.g., due to OS is also running other programs)
- Or maybe, despite load balancing, tasks end up taking different time.

Exploiting java.util.concurrent

java.util.concurrent Features in Brief

- **Semaphore** objects resemble those seen already
  - except **acquire()** & **release()** instead of **P , V** (resp)
- **Lock** objects support locking idioms that simplify many concurrent applications
  - don’t mix up with implicit locks!
- **Executors** give high-level API for launching, managing threads
  - **Executor** implementations provide thread pool management suitable for large-scale applications.
- Concurrent **Collections** support concurrent management of large data collections in HashTables, different kinds of Queues, etc.
- **Future** objects are enhanced to have their status queried and return values when used in connection with asynchronous threads (in java.util.concurrent)
- Atomic variables (e.g., **AtomicInteger**) support atomic operations on single variables
  - features that minimize synchronization & help avoid memory consistency errors
  - i.e. useful in applications that call for atomically incremented counters
- ...

Semaphore Objects

- Used to control the number of activities that can access a certain resource or perform a given action at the same time
- **Counting semaphores** can be used to implement resource pools or to impose a bound on a collection
- **Semaphore** object maintains a set of permits (allowed usages of resource):
  - e.g., Semaphore exampleSemaphore = new Semaphore(int permits);
- To use a resource protected by a semaphore:
  - Must invoke **exampleSemaphore**.**acquire**() method
  - If all permits for that semaphore are not used => your thread may continue
    Else your thread blocks until permit is available;
  - When you are finished with the resource, use the semaphore.release() method
  - Each release adds a permit
- **Semaphore** constructor also accepts a fairness parameter: **Semaphore(int permits, boolean fair);**

  permits: initial value
  fair:
    if true semaphore uses **FIFO** to manage blocked threads
    if set false, class doesn't guarantee order threads acquire permits.
- Otherwise, barging
  - i.e., thread doing acquire()can get a permit ahead of one waiting longer

Throttling with Semaphore class

- Often must throttle number of open requests for a resource.
  - to improve throughput of a system ...
    by reducing contention for that particular resource.
- Alternatively, it might be a question of starvation prevention
  - This was shown in the room case of Dining Philosophers (above)
    - Only want to let 4 philosophers in the room at any one time
- Can write the throttling code ourselves, but it's often easier to use Semaphore class - does it for you!

Example 3: Semaphore Example

```
//SemApp: code to demonstrate throttling with semaphore class © Ted Neward
import java.util.*;
import java.util.concurrent.*;

public class SemApp {
        public static void main( String[] args ) {
                Runnable limitedCall = new Runnable () {
                        final Random rand = new Random();
                        final Semaphore available = new Semaphore(3); //semaphore obj with 3 permits
                        int count = 0;

                        public void run() {
                            int time = rand.nextInt(5);
                            int num = count++;

                            try {
                                available.acquire();
                                System.out.println("Executing “ + “longrun action for “ +
                                                 time + “ secs.. #" + num);
                                Thread.sleep(time * 1000);
                                System.out.println("Done with # “ + num);
                                available.release();
                            } catch (InterruptedException intEx) {
                                intEx.printStackTrace();
                            }
                        }

                };

                for (int i=0; i<10; i++)
                        new Thread(limitedCall).start(); // kick off worker threads

        } // end main
} // end SemApp
```

Monitoring Threads in the JVM

- Even though the 10 threads in Example 3 code are running, only three are active (= permits)
- You can verify by executing jstack (**Java concurrency debug support tool**) against the Java process running SemApp*
  - The other seven are held at bay pending release of one of the semaphore counts
- Actually, the Semaphore class supports acquiring and releasing more than one permit at a time
  - However, that wouldn't make sense in this scenario.

*For more see for example https://experienceleague.adobe.com/docs/experience-cloud-kcs/kbarticles/KA-17452.html?lang=en

Yet More Locking – Coordinated Access to Shared Data in Java 5+

Many situations are not easy to handle with Intrinsic Locks/mutex/synchronized keyword:

- We want to interrupt a thread waiting to acquire a lock
- We want to acquire a lock but cannot afford to wait forever
- We want to release a lock in a different block of code from the one that acquired it
  - to support a more complex locking protocol

Introducing the Lock Interface

```
public interface Lock {
      void lock();
      void lockInterruptibly() throws InterruptedException;
      boolean tryLock();
      boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException;
      void unlock();
      Condition newCondition();
}
```

Interface Lock

- Lock implementations operate like the implicit locks used by synchronized code
  - only 1 thread can own a Lock object at a time[1]
- Unlike intrinsic locking all lock and unlock operations are explicit and have bound to them explicit Condition objects
- Biggest advantage over intrinsic locks is: can back out of an attempt to acquire a Lock:
  - i.e., mitigates issues regarding livelock, starvation & deadlock
- For example, these Lock methods:
  - **tryLock**() returns if lock is not available immediately or before a timeout (optional parameter) expires
  - **lockInterruptibly**() returns if another thread sends an interrupt before the lock is acquired

A thread can't get a lock owned by another thread, but it can get a lock that it already owns. Letting a thread acquire the same lock more than once enables Reentrant Synchronization (i.e. thread with the lock on a synchronized code snippet can invoke another bit of synchronized code e.g. in a monitor.)

Canonical code form for using a Lock

```
Lock egLock = new ReentrantLock();
...
egLock.lock();
try {
      // update object state
      // catch exceptions and restore
      // invariants if necessary
} finally {
      egLock.unlock();
}
```

Interface **Lock**

- **Lock** interface also supports a **wait/notify** mechanism, through the associated **Condition** objects
- Thus not restricted with basic monitor methods (**wait**(), **notify**() & **notifyAll**()) with specific objects:
    - Lock in place of **synchronized** methods and statements.
    - An associated **Condition** in place of Object's monitor methods.
    - A **Condition** instance is intrinsically bound to a Lock.
- To obtain a **Condition** instance for a particular Lock instance use its **newCondition**() method.

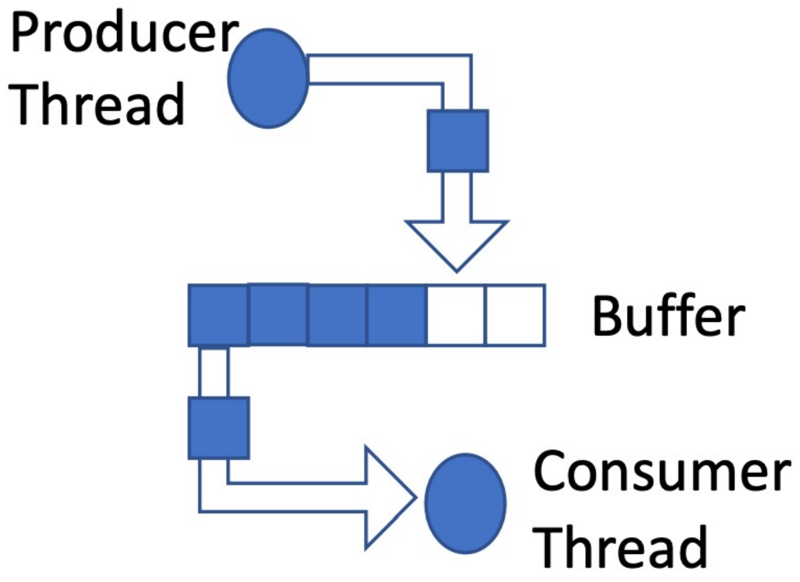
Reentrantlocks & synchronized Methods

- **Reentrantlock** implements **Lock interface** with the same mutual exclusion guarantees as **synchronized**
- Acquiring/releasing a **Reentrantlock** has the same memory semantics as entering/exiting a **synchronized** block
- So why use a Reentrantlock in the first place?
  - Using **synchronized** gives access to the implicit lock an object has, but forces all lock acquisition/release to occur in a block-structured way:
    - if multiple locks are acquired they must be released in the opposite order
  - **Reentrantlock** allows a more flexible locking/releasing mechanism
  - **Reentrantlock** supports scalability and is nice where there is high contention among threads
- So why not get rid of **synchronized**?
  - Firstly, a lot of legacy Java code uses it
  - Secondly, there are performance implications to using **Reentrantlock**

Bounded Buffer Problem

- Need a Lock protocol such that:
  - Producer cannot add data to buffer when it is full
  - Consumer cannot take data from an empty buffer

- Define two Lock Conditions for each of these buffer states (notFull, notEmpty)



Example 4: Bounded Buffer Using Lock & Condition Objects

```
class BoundedBuffer {
        final Lock lock = new ReentrantLock();
        final Condition notFull = lock.newCondition();
        final Condition notEmpty= lock.newCondition();
        final Object[] items = new Object[100];
        int putptr, takeptr, count;
```

```
public void put(Object x) throws
                InterruptedException {
        lock.lock(); // Acquire lock on object

        try {
                while (count == items.length)
                        notFull.await(); //condition

                items[putptr] = x;
                if (++putptr == items.length)
                        putptr = 0;
                ++count;
                notEmpty.signal();
        }
        finally {
                lock.unlock(); // release the lock
        }
    }
```

```
public Object take() throws InterruptedException
{
        lock.lock();// Acquire lock on object

        try {
                while (count == 0)
                        notEmpty.await(); // condition
                Object x = items[takeptr];
                if (++takeptr == items.length)
                        takeptr = 0;
                --count;
                notFull.signal();

                return x;
        } finally {
                lock.unlock(); // release the lock
        }
    }
}
```

```java
import java.util.concurrent.locks.*;
/**
* Bank.java shows use of the locking mechanism with ReentrantLock object for money transfer fn. @author www.codejava.net
*/
public class Bank {
   public static final int MAX_ACCOUNT = 10;
   public static final int MAX_AMOUNT = 10;
   public static final int INITIAL_BALANCE = 100;
   private Account[] accounts = new Account[MAX_ACCOUNT];
   private Lock bankLock;
   public Bank() {
       for (int i = 0; i < accounts.length; i++) {
           accounts[i] = new Account(INITIAL_BALANCE);
       }
       bankLock = new ReentrantLock();
   }
   public void transfer(int from, int to, int amount) {
       bankLock.lock();
       try {
           if (amount <= accounts[from].getBalance()) {
               accounts[from].withdraw(amount);
               accounts[to].deposit(amount);
               String message = "%s transfered %d from %s to %s. Total balance: %d\n";
               String threadName = Thread.currentThread().getName();
               System.out.printf(message, threadName, amount, from, to, getTotalBalance());
           }
       } finally {
           bankLock.unlock();
       }
   }
   public int getTotalBalance() {
       bankLock.lock();
       try {
           int total = 0;
           for (int i = 0; i < accounts.length; i++) {
               total += accounts[i].getBalance();
           }
           return total;
       } finally {
           bankLock.unlock();
       }
   }
}
```

```java
/**
* Account.java is a bank account @author www.codejava.net
*/
public class Account {
   private int balance = 0;
   public Account(int balance) {
       this.balance = balance;
   }
   public void withdraw(int amount) {
       this.balance -= amount;
   }
   public void deposit(int amount) {
       this.balance += amount;
   }
   public int getBalance() {
       return this.balance;
   }
}
```

Example 5: Bank Account Example using `Lock` Object

Example 6: Dining Philosophers Using `Lock` Objects

```
public class Fork {
  private final int id;
  public Fork(int id) {
    this.id = id;  }
// equals, hashcode, and toString() omitted
}
public interface ForkOrder {
  Fork[] getOrder(Fork left, Fork right);
} // We will need to establish an order of pickup


// Vanilla option w. set pickup order implemented
class Philo implements Runnable {
  public final int id;
  private final Fork[] Forks;
  protected final ForkOrder order;

  public Philo(int id, Fork[] Forks, ForkOrder
order) {
    this.id = id;
    this.Forks = Forks;
    this.order = order;
  }
```

```
  public void run() {
    while(true) { eat();
    }
  }

  protected void eat() {
// Left and then Right Forks picked up
    Fork[] ForkOrder = order.getOrder(getLeft(),
getRight());
    synchronized(ForkOrder[0]) {
      synchronized(ForkOrder[1]) {
        Util.sleep(1000);
      }
    }
  }

  Fork getLeft() {  return Forks[id]; }
  Fork getRight() { return Forks[(id+1) %
Forks.length]; }
  }
}
```

- This can, in principle, be run & philosophers just eat forever: choosing which fork to pick first; picking it up; then picking the other one up then eating etc.
- If you look at the code above in the eat() method, ‘grab the fork’ by synchronizing on it, locking the fork's monitor.

Example 6: Dining Philosophers Using **Lock** Objects (/2)

```
class Philo implements Runnable {
  public final int id;
  private final Fork[] Forks;
  protected final ForkOrder order;

  public Philo(int id, Fork[] Forks, ForkOrder
order) {
    this.id = id;
    this.Forks = Forks;
    this.order = order;
  }

public class GraciousPhilo extends Philo {
  private static Map ForkLocks = new
ConcurrentHashMap();

  public GraciousPhilo(int id, Fork[] Forks,
ForkOrder order) {
    super(id, Forks, order);
// Every Philo creates a lock for their left Fork
    ForkLocks.put(getLeft(), new ReentrantLock());
  }

  protected void eat() {
    Fork[] ForkOrder = order.getOrder(getLeft(),
getRight());
    Lock firstLock = ForkLocks.get(ForkOrder[0]);
    Lock secondLock = ForkLocks.get(ForkOrder[1]);
    firstLock.lock();

    try {
      secondLock.lock();

      try {
        Util.sleep(1000);
      } finally {
        secondLock.unlock();
      }
    } finally {
      firstLock.unlock();
    }
  }
}
```

- Just replace `synchronized` with `lock()` & end of `synchronized` block with a `try { } finally { unlock() }`.
- This allows for timed wait (until finally successful) or employ a strategy using:
  - `lockInterruptibly()` - block if lock already held, wait until lock is acquired; if another thread interrupts waiting thread `lockInterruptibly()` - will throw `InterruptedException` or `tryLock()` / `tryLock(timeout)` ...

Concurrent Annotations

- Annotations were added as part of Java 5.
- Java comes with some predefined annotations
  - e.g., @Override,
  - but custom annotations are also possible, e.g., @GuardedBy
- Annotations are processed at compile time or at runtime (or both).
- Good programming practice to use annotations to document code

```
public class BankAccount {
        private Object credential = new Object();
        @GuardedBy("credential") // amount guarded by credential because
        private int amount; // access only if synch lock on credential held
}
```

References

1. Timothy Mattson , Beverly Sanders , Berna Massingill, Patterns for parallel programming, Addison-Wesley Professional, 2004. ISBN-13: 978-0321228116
2. MIT 6.189 Multicore Programming Primer, IAP 2007
3. Gethin Williams, Patterns for parallel programming lecture notes,2010 https://www.researchgate.net/publication/234826291_P atterns_for_Parallel_Programming
4. Moreno Marzolla, Parallel Programming Patterns, 2018, https://www.moreno.marzolla.name/teaching/HPC/L03- patterns.pdf
5. Bill Venners, Inside the Java Virtual Machine,Book, https://www.artima.com/insidejvm
6. Brian Goetz, Java Concurrency in Practice, ISBN: 0321349601
7. Our Pattern Language, Berkley 2019, https://patterns.eecs.berkeley.edu/