Concurrent and Distributed Programming (CSC1101)

Concurrency in Practice
(Java, OpenMP, OpenMPI...)

2024/2025

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

Java Concurrency
(extending Concurrent Correctness)

Last day

- Spoke about about **synchronized** keyword, and **notify()** and **wait()**

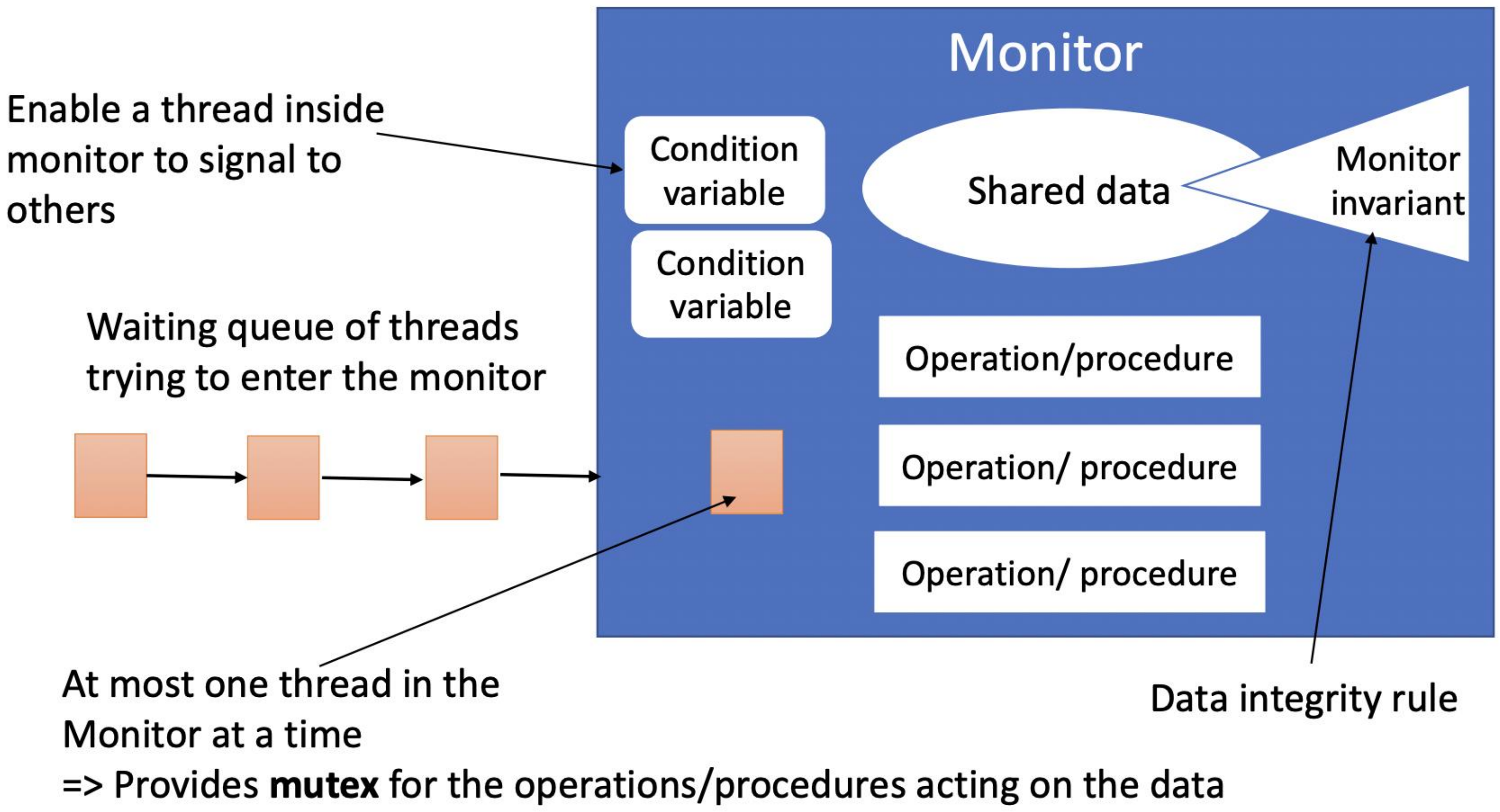
- Implicit monitors in Java

- But didn't show how to produce threads that might use such code

```
Class Counter {
        //Prints value every time count>0
        private int count = 0;

        public void synchronized increment() {
                int n = this.count;
                this.count = n + 1;
                If (count > 0)
                   notify(); //let print know
        }

        public void synchronized decrement(){
                int n = this.count;
                count = n - 1;
        }

        public void synchronized printVal() {
                while (this.count <= 0)
                   wait();

                System.out.println("Count=" + count);
        }
}
```

Last day

- Monitor concept



Java Threads (/1)

- A 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

Threads (/2)

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

```
class myThread extends Thread
{
        public void run ( )
        {
                System.out.println ("Hello from the thread");
        }
}
myThread t = new myThread ( );
t.start ( );
```

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

    new myThread().start();

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 myThread implements Runnable
{
        public void run ( ) {
                System.out.println ("Hello from the thread");
        }
}

Runnable myThreadInstance = new myThread ( );
Thread t = new Thread(myThreadInstance);
t.start( );
```

Threads (/4)

- We can wait for the thread to finish
- 2 flavours of join( ) method – wait forever or for a specific times (milli, nano)
- **join**( ) waits for specified thread to finish, 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 calling thread will block until the running thread instance has finished executing (or time elapses if provided):

```
try {
        t.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

- We could even use a Timer() & TimerTask() to create threads for executing deferred tasks

Example: Non-thread-safe 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.

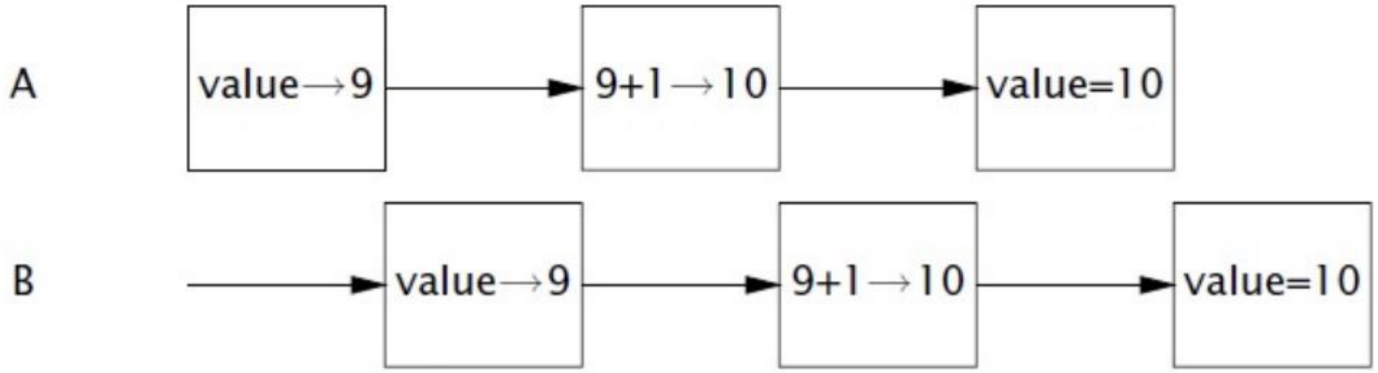


FIGURE 1.1. Unlucky execution of UnsafeSequence.getNext.

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***(), etc methods
  - Enables threads to work together towards a common goal

Mutual Exclusion in Java: Synchronization

- Threads in Java (can) execute concurrently,
  - Hence, they could simultaneously access shared variables ... leading to 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 – if other threads that respect the lock!
- 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
  - ... 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

- Also, just synchronising every method is sometimes not optimal
  - When locks are used, they make code serial
  - Can lead to very poor performance
  - Let’s see

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]

Why is this Example Bad?

```java
@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);
        }
    }
}
```

Hint: What is it we are trying to protect by using **synchronized** in the first place?

Cache of last result
To improve performance

Figure based on ref [6]

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);
        }
    }
}
```

Hint: What is it we are trying to protect by using **synchronized** in the first place?

Cache of last result
To improve performance

Figure based on ref [6]

Rationale

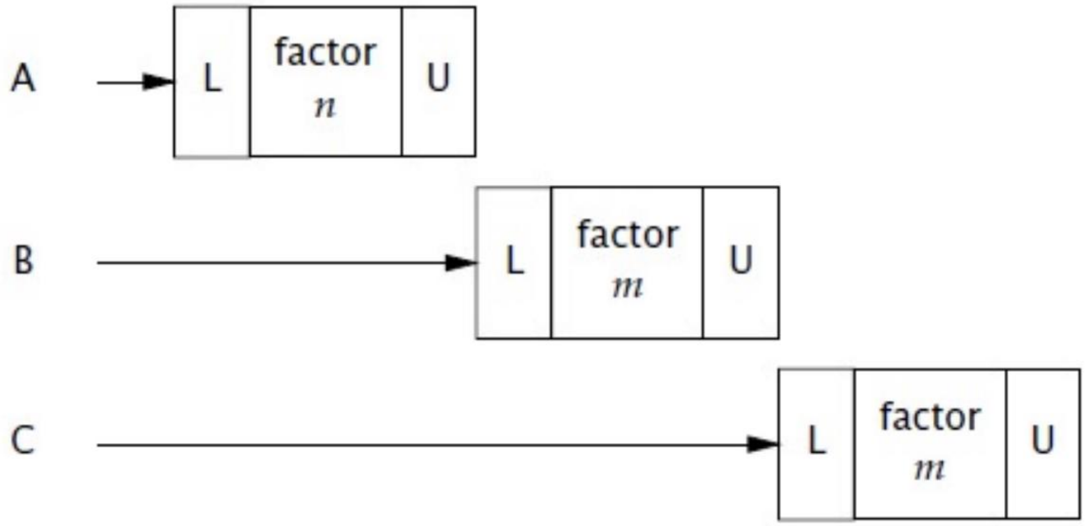


FIGURE 2.1. Poor concurrency of SynchronizedFactorizer.

- Caching required shared state
- Protected it with coarse-grained lock
  => yes, it's 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

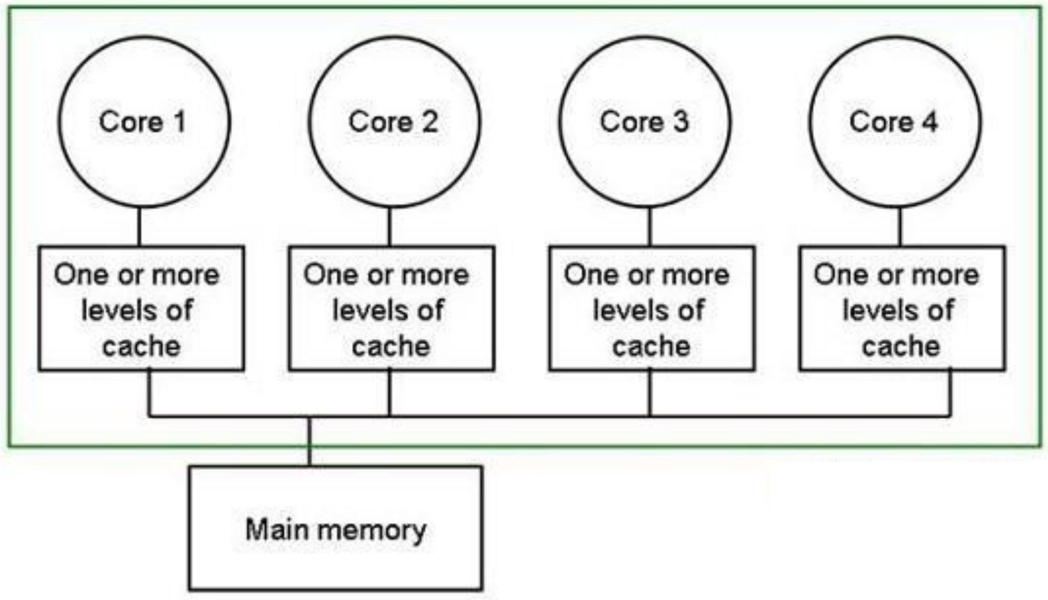
```java
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    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(s) is accessed
  - E.g., not just when initialized
  - All accesses/writes must use the same lock - read and write
  - For convenience Java has one intrinsic lock per object so you don't have to explicitly create lock objects ...
- Remember: 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 **<u>up to us</u>** to construct locking protocols or synchronization policies that let you access a shared state safely, and to use them consistently throughout your program

Java Memory Visibility

- Synchronized is not only about atomicity or demarcating “critical sections” of code
  - It also determines **memory visibility**
  - Guarantees all threads will see the same data (effects of all modifications), if guarded by the same lock!
- 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



Aside Example 1: Readers/Writers - Monitors

```
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 1: Readers/Writers - Monitors

```
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 ( );
        }
}
```

Monitors in Java: implementing Queue Class

wait() & **notify**() in Java for **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);
        }
}
```

Master-Worker Pattern

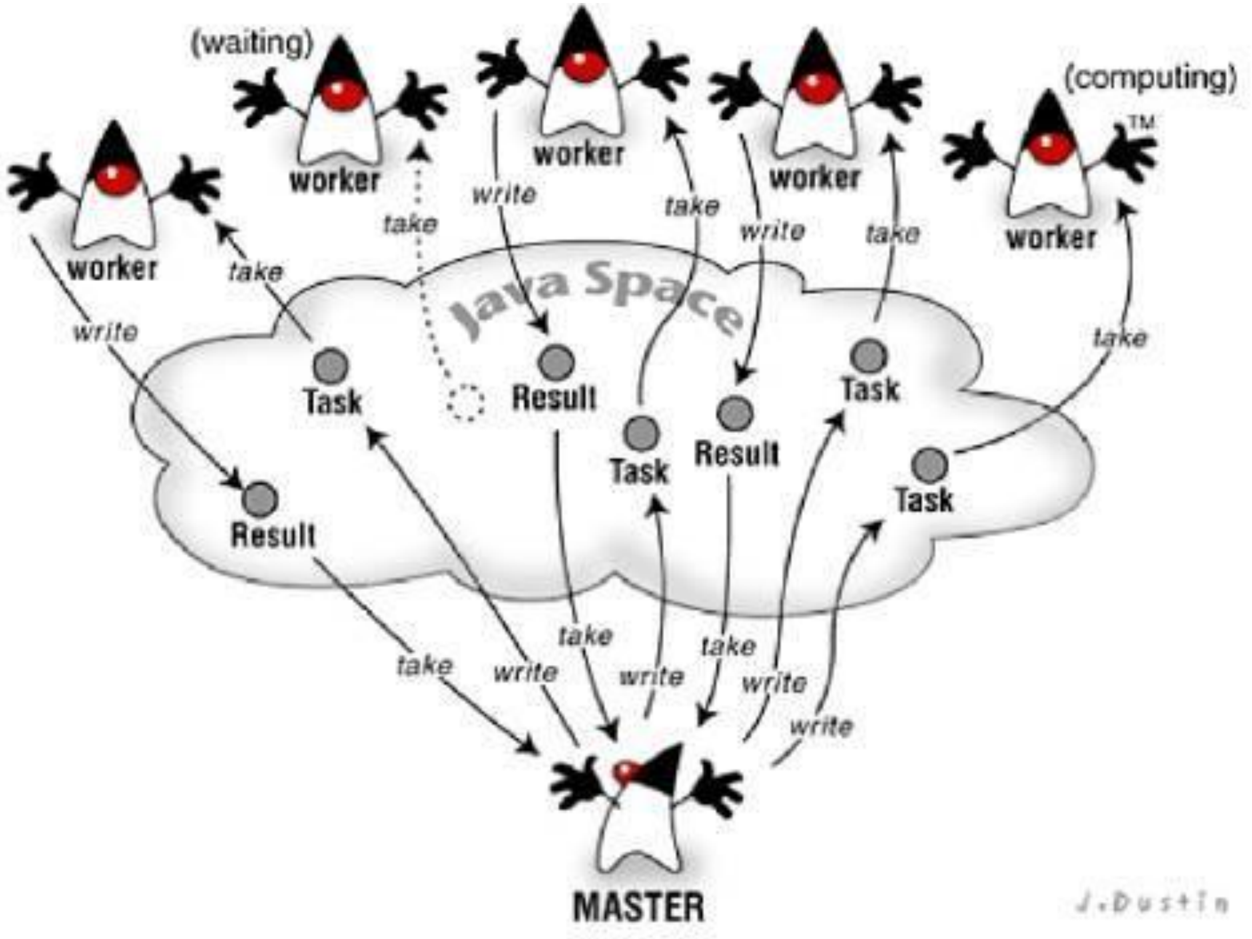


Image: https://docs.gigaspaces.com/sbp/master-worker-pattern.html

Our use of threads is becoming more complex

- We need to begin thinking of these problems in terms of patterns

- Two important ones are:
  - Master/Worker
  - Fork/Join

- Let’s see look at java Executors first though
  - Because we (might) need better ways to handle threads

Executors

- As seen, one way 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 just protecting shared resources
- Seen already how to create multiple threads and coordinate them
  - via synchronized methods and blocks, as well as via Lock objects

- But 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

have a look at: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html
we'll look fixedthreadpools

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
        }

        // 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
                }
                // if thread interrupted while sleeping, print stack trace
                catch ( InterruptedException exception ) {
                        exception.printStackTrace();
                }

                // 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

Executors: Futures/Callables

- **`Executor`** interface uses **`Runnable`**s -> i.e. **`Runnable`** can't return a result.
- A **`Callable`** object allows return values after completion.
- **`Callable`** uses generics to define type of object returned.
- If you submit a **`Callable`** object to an **`Executor`**, framework returns **`java.util.concurrent.Future`** object.
- This **`Future`** object can be used to check the status of a **`Callable`** and to retrieve the result from the **`Callable`**.

Executors: Futures/Callables

- So, writing asynchronous concurrent programs that return results using executor framework requires:
  - Define class/task implementing either `Callable` interface
  - Configure & implement `ExecutorService`
  - (This because need `ExecutorService` to run the `Callable` object.)
  - The service accepts `Callable`s to run using `submit()` method
  - Submit task using `Future` class to retrieve result if task is `Callable`
- Difference between a `Runnable` and `Callable`:
  - `Runnable` interfaces do not return a result V `Callable` permits returning values after completion.
  - When a `Callable` is submitted to the executor framework, it returns an object of type `java.util.concurrent.Future`.
  - The `Future` can be used to retrieve results

Example: Futures/Callables

```
package de.vogella.concurrency.callables;
import java.util.concurrent.Callable;
public class MyCallable implements Callable<Long> {
  @Override
  public Long call() throws Exception {
    long sum = 0;
    for (long i = 0; i <= 100; i++) {
      sum += i;
    }
    return sum;
  }
}
```

[1] This code and associated piece on the next page were written and are Copyright © Lars Vogel. Source Code can be found at *de.vogella.concurrency.callables*.

Example: Futures/Callables

```java
package de.vogella.concurrency.callables;
import java.util.ArrayList;
import java.util.List;import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;import java.util.concurrent.Future;
public class CallableFutures {
  private static final int NTHREDS = 10;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
    long sum = 0;
    System.out.println(list.size());
    // now retrieve the result
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum);executor.shutdown();
  }
}
```

Lars Vogel

Example: Futures/Callables

```
package de.vogella.concurrency.callables;
import java.util.ArrayList;
import java.util.List;import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;import java.util.concurrent.Future;
public class CallableFutures {
  private static final int NTHREDS = 10;
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
    List<Future<Long>> list = new ArrayList<Future<Long>>();
    for (int i = 0; i < 20000; i++) {
      Callable<Long> worker = new MyCallable();
      Future<Long> submit = executor.submit(worker);
      list.add(submit);
    }
    long sum = 0;
    System.out.println(list.size());
    // now retrieve the result
    for (Future<Long> future : list) {
      try {
        sum += future.get();
      } catch (InterruptedException e) {
        e.printStackTrace();
      } catch (ExecutionException e) {
        e.printStackTrace();
      }
    }
    System.out.println(sum);executor.shutdown();
  }
}
```

These can run asynchrously

And we get() the result later (this blocks)

Lars Vogel

Executors: Futures/Callables

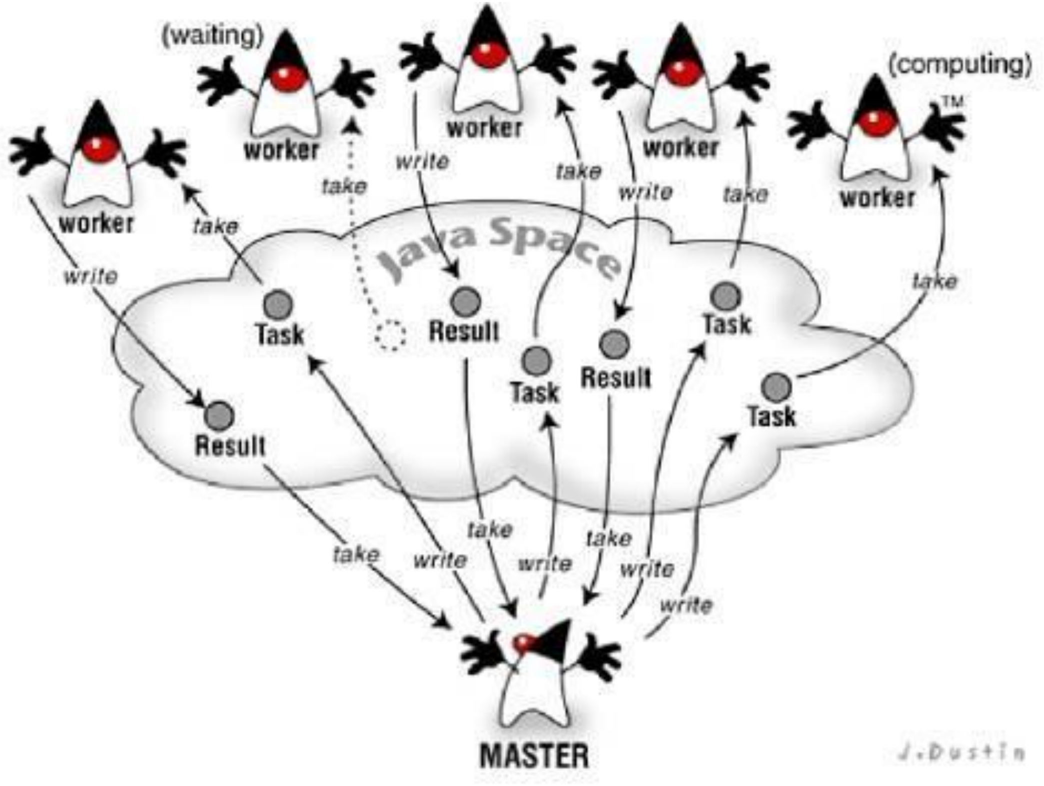
- **`java.util.concurrent.Future`**.
  - Besides **.get()** – there is also **isDone()** method – meaning that we could check periodically if the result is computed or not yet. or could call **.cancel()** ...

There's also ....

- **`java.util.concurrent.CompletableFuture`**.
  - It's like java.util.concurrent.Future but has async callbacks

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



Fork/Join Pattern

- Similar to master-worker
  - But 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

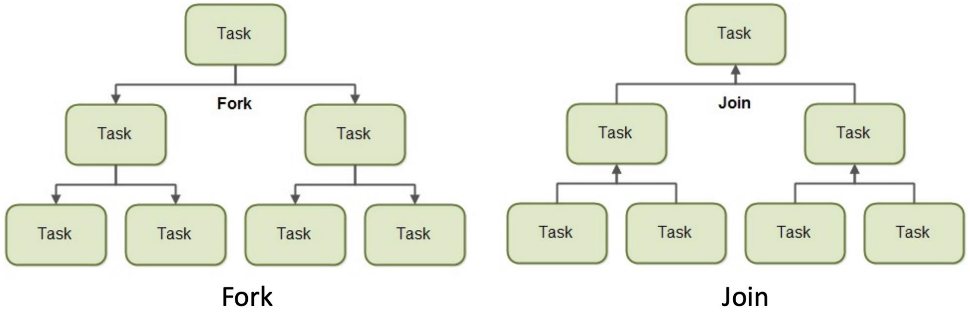
Example: Returning a Result from a ForkJoinPool

```
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.

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 allows us to adopt 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.

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 (/2)

- A key class is the **ForkJoinPool**
  - an implementation of ExecutorService implementing work-stealing

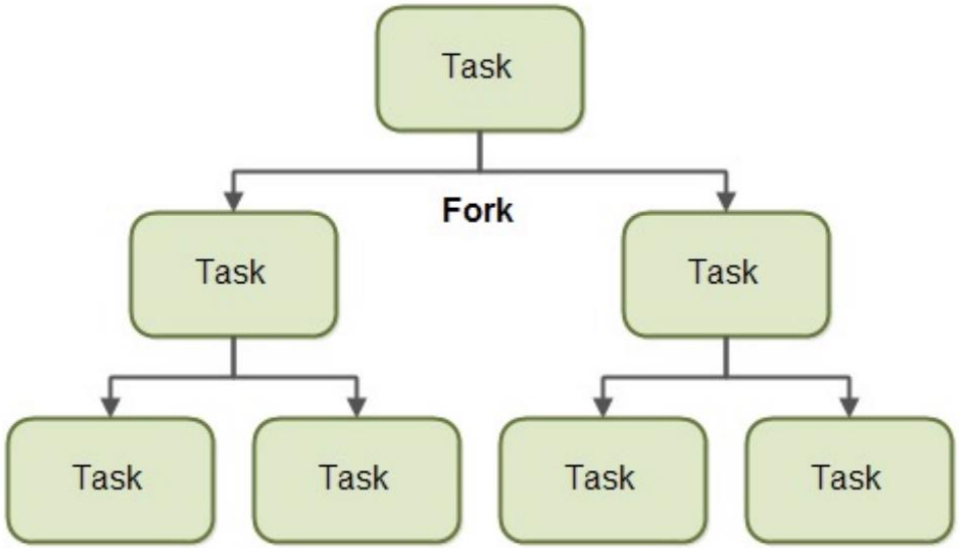
- A **ForkJoinPool** is instantiated like so:

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

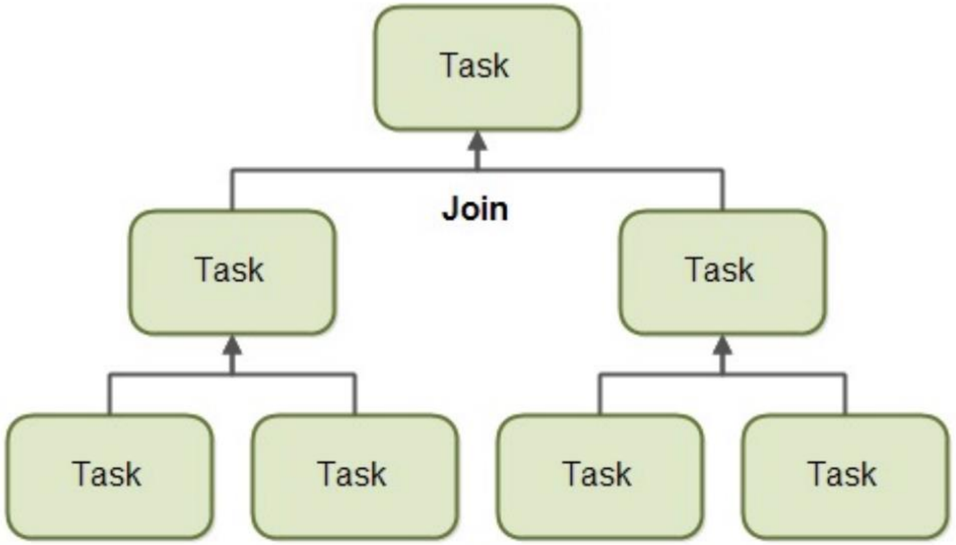
  ForkJoinPool pool = new ForkJoinPool(numberOfCores);

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: 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 cores available varies during execution (e.g., due to OS is also running other programs)

Exploiting java.util.concurrent

java.util.concurrent

- **Semaphore** objects resemble those seen already
  - except **acquire()** & **release()** instead of **P() , V()**
- **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

Example: *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 ) {

                final Random rand = new Random();
                final Semaphore available = new Semaphore(3); //semaphore obj with 3 permits


                Runnable limitedCall = new Runnable () {
                        public void run() {
                          int time = rand.nextInt(5);

                          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
- To note... 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 monitor locks/mutex/synchronized keyword, such as :

- 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();
        boolean tryLock();
        boolean tryLock(long timeout, TimeUnit unit)
                throws InterruptedException;
        void unlock();
        void lockInterruptibly() throws InterruptedException;
        Condition newCondition();
}
```

```
Class XYZ {
        Lock lock1 = new ReentrantLock();
        …
        Condition l1cond1 = lock.newCondition();
        Condition l1cond2 = lock.newCondition();
}
```

Interface Lock

- Lock implementations operate like the implicit locks used by synchronized code
  - only 1 thread can own a particular Lock object at a time[1]
- Unlike intrinsic locking all lock and unlock operations are explicit, and can have bound to them explicit Condition objects
- Big 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**

- As we seen **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.

```
Class XYZ {
        Lock lock1 = new ReentrantLock();
        ...
        Condition l1cond1 = lock.newCondition();
}
```

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 -- 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**

Example use of implicit monitor in Java

```
Class Counter {
        //Prints value every time count>0
        private int count = 0;

        public void synchronized incrementTwiceAndDecrement()
{
                increment();
                increment();
                decrement();
        }

        public void synchronized increment() {
                int n = this.count;
                this.count = n + 1;
        }

        public void synchronized decrement(){
                int n = this.count;
                count = n - 1;
        }

}
```

If thread-1 calls incrementTwiceAndDecrement()

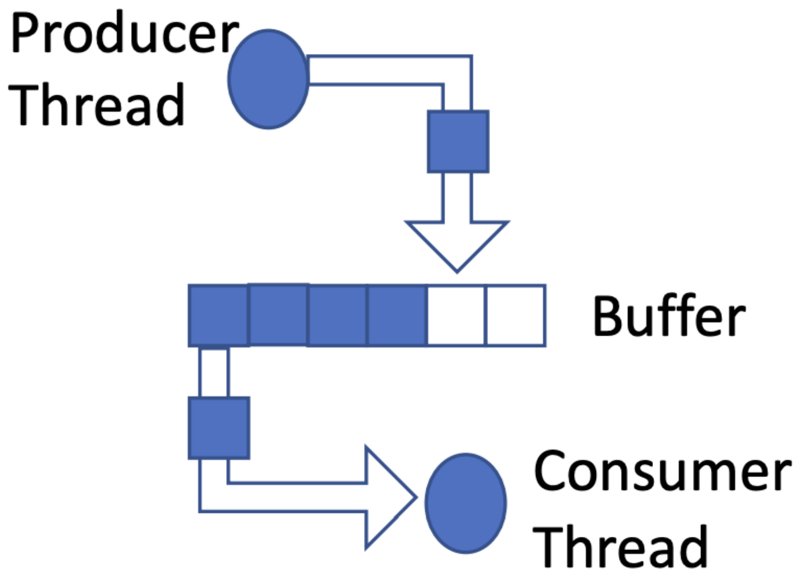
this ends up calling increment() and decrement()

It's fine, as it's a ReentrantLock !
It already has the locked when it calls increment() and decrement()

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)



Canonical code form for using a Lock with a Condition

```
void stateDependentMethod() throws InterruptedException
{
        // condition predicate must be guarded
        // by lock
        synchronized(lock) {
                while (!conditionPredicate())
                        lock.wait();//or Condition.await()
        // object is now in desired state
        }
}
```

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 = java.util.concurrent.ReentrantLock();
   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: 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) ...`

Dining Philosophers Using **`ReentrantLocks`** (/3)

- Can leverage additional power of **`ReentrantLock`** to do some niceties:
  - First, don't have to block forever on the **`lock`** call.
  - Instead we can do a timed wait using **`tryLock()`**.
  - One form of this method returns immediately if the lock is already held
  - Other can wait for some time for the lock to become available before giving up.
  - In both, could effectively loop and retry the **`tryLock()`** until it succeeds.

- Another nice option is to **`lockInterruptibly()`**
  - Calling this allows for waiting indefinitely but reply to thread being interrupted.
  - Possible to write an external monitor that either watches for deadlock or allows a user to forcibly interrupt one of the working threads.
  - Could be provided via JMX to allow a user to recover from a deadlock.

Concurrent Annotations

- Annotations were added as part of Java 5.
- Java comes with some predefined annotations
  - e.g., @Override,
  - but other 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

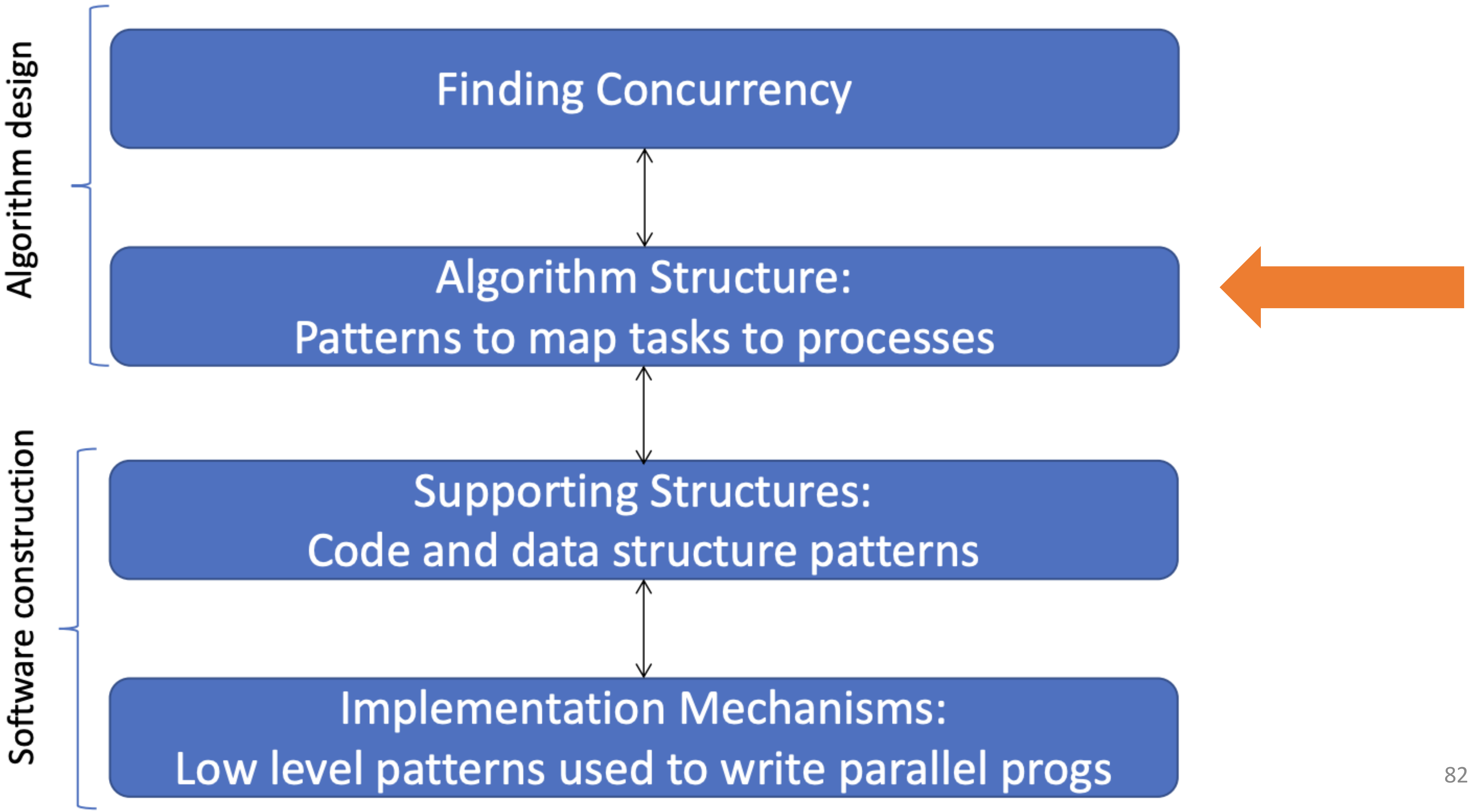
```
public class BankAccount {
        private Object credential = new Object();
        @GuardedBy("credential") // amount guarded by credential because
        private int amount; // access only if synch lock on credential held
}
```

Concurrent Java Summary

- Executables and Fork-Join vs basic Threads/Runnable
- Build on standard concurrent classes like ConcurrentHashMap rather than building your own custom protected data types whenever possible
- Use Intrinsic Locks where possible, but be aware of their limitations.
    - While ReentrantLocks offer ‘more’, there is associated overhead vs inbuilt implicit lock

Interlude

Four Design Spaces



Problem Decomposition / Finding Concurrency

- Identify concurrency and decide at what level to exploit it
  - Tasks, data, pipelines e.g., task decomposition vs data decomposition

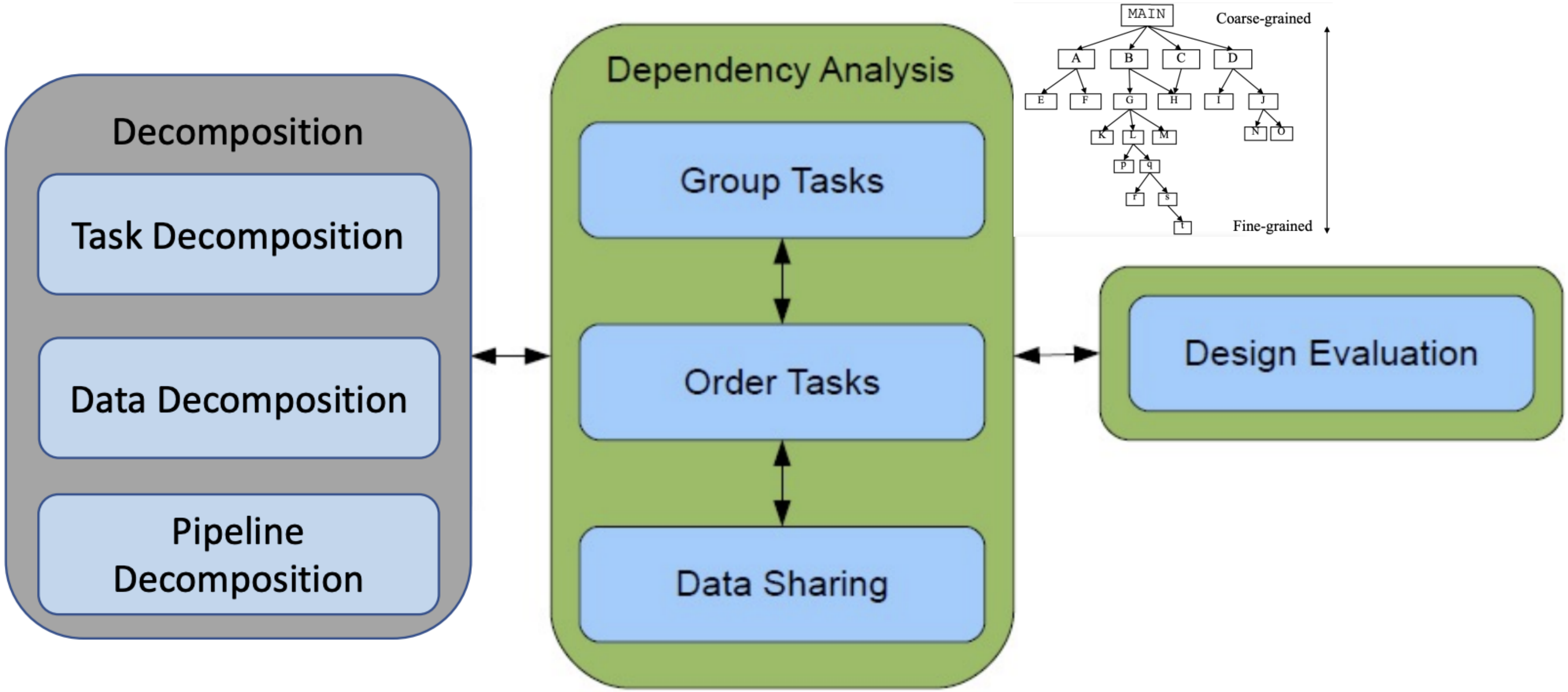


Figure based on ref [3]

Algorithm Structure Design Space

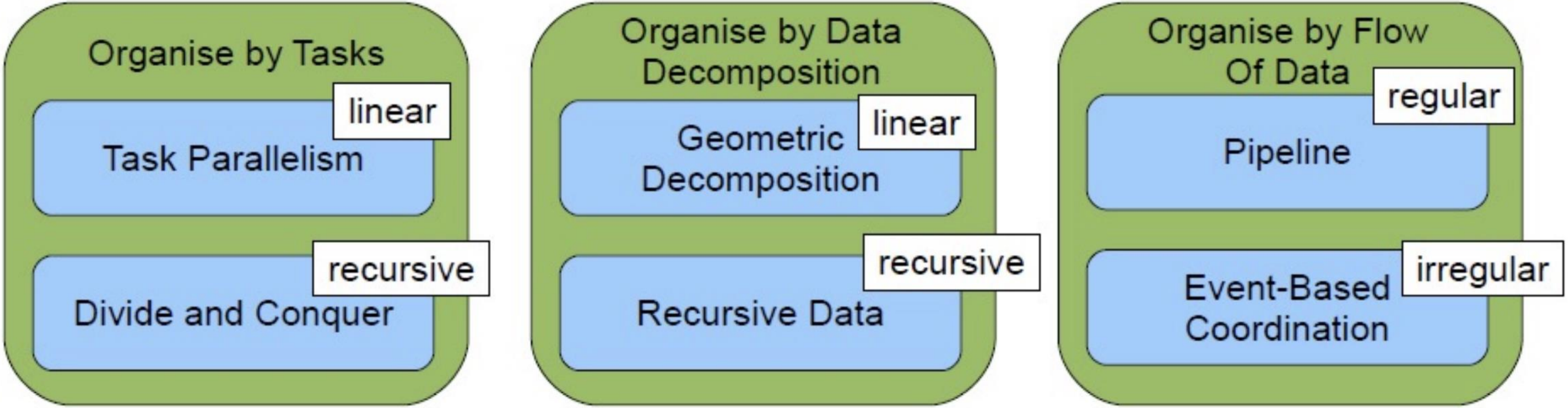


Figure based on ref [3]

Supporting Structures with Example Technologies

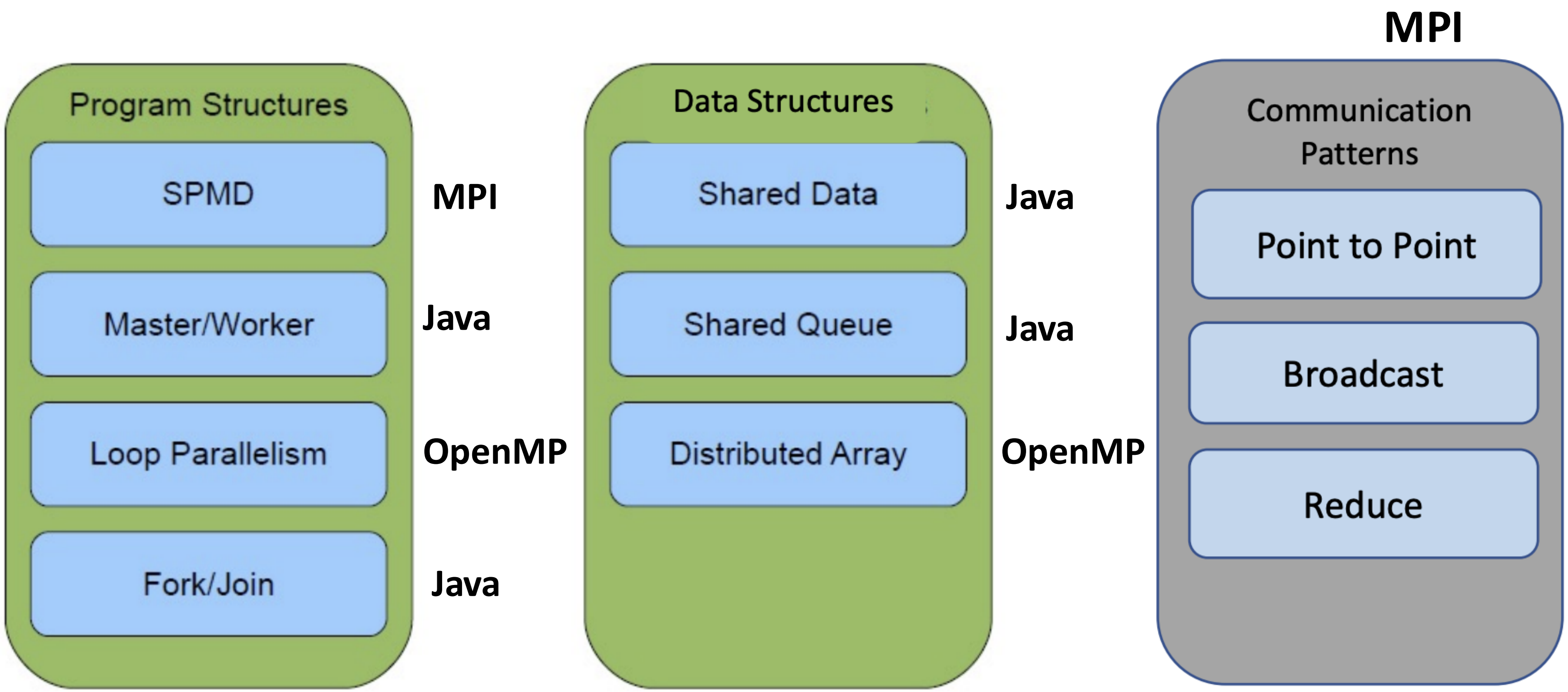


Figure based on ref [3]

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

OpenMP

Introduction to OpenMP

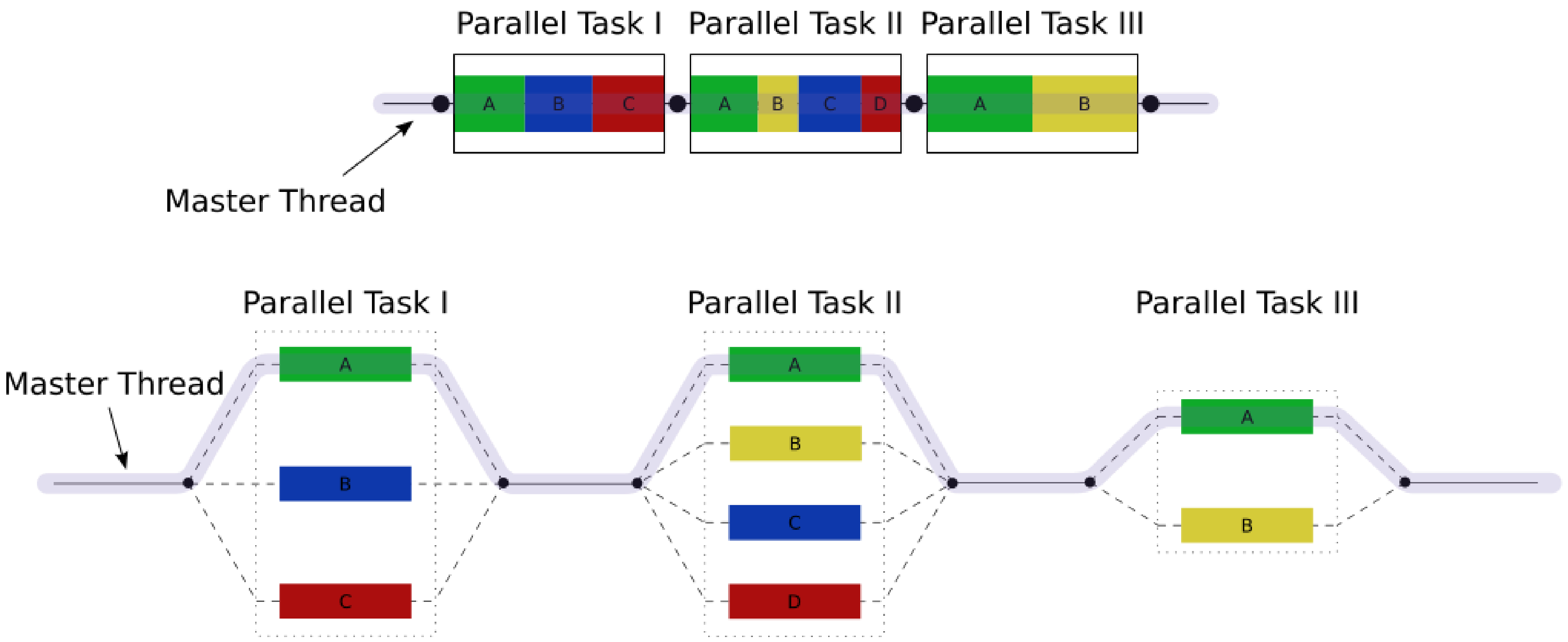
- Stands for *Open Multi-Processing*, or *Open specifications for Multi-Processing*
- Represents collaboration between interested parties from h/w and s/w - industry, government and academia.
- An API to facilitate **explicit** programmer-directed multi-threading, shared memory parallelism.
- Supported in C, C++, and Fortran, and on most processor architectures and OS.
- Comprises a set of compiler directives, library routines, and environment variables affecting run-time behaviour.
- Introduce it here as complementary to and usable in conjunction with MPI (more later on this) to achieve speedup

Motivations to use OpenMP

- Provides a standard among a variety of shared memory architectures/platforms.
  - Currently at OpenMP Version 6 stable (as of Nov 2024)
  - More details at `openmp.org/resources/`
- Establishes a simple and limited set of directives for programming shared memory machines.
  - (like MPI) we can get quite good parallelism using 3 or 4 directives ...
- Unlike MPI:
  - Facilitates incremental parallelization of a serial program,
  - Does not require 'all or nothing' approach to parallelization,
  - MPI scales well but is non-trivial to implement for code originally written for serial machines (& those not good for shared memory)

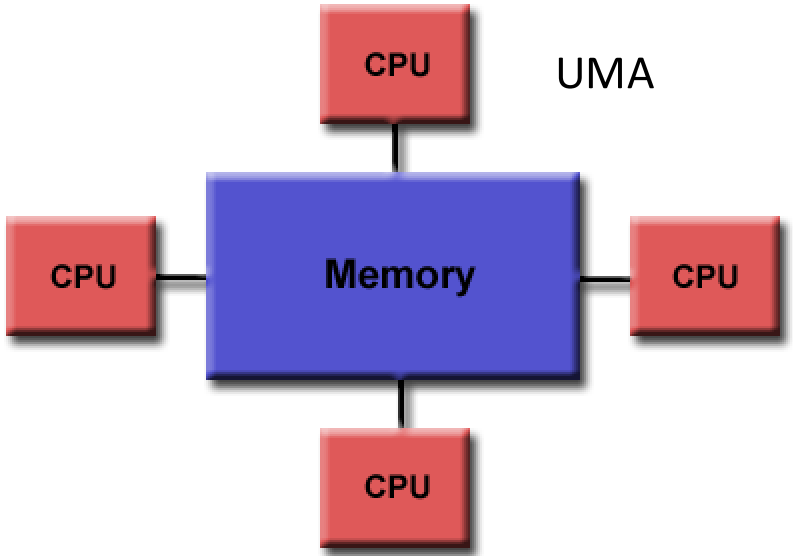
OpenMP Programming Model

The *Fork-Join* Model of parallel execution

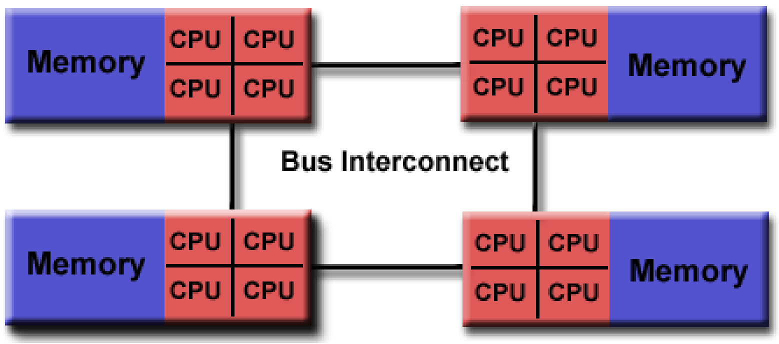


OpenMP Programming Model

- Shared Memory Model:
  - OpenMP is designed for multi-processor/core, shared memory machines.
  - The underlying architecture can be shared memory UMA or NUMA.



UMA



NUMA

What OpenMP is not

- OpenMP is :
  - not meant for distributed memory parallel systems (by itself)
  - not guaranteed to make the most efficient use of shared memory
  - requires explicit user-directed parallelization

- OpenMP *will not:*
  - Check for data dependencies, data conflicts, race conditions, or deadlocks
  - Check for code lines that cause program to be classified as non-conforming

Parallelism in OpenMP

- *Thread-Based Parallelism*
  - OpenMP programs accomplish parallelism solely using ***threads***.
  - A *thread of execution* is smallest processing unit schedulable by OS.
    - Analogous conceptually to a subroutine that can be scheduled to run autonomously.
  - These threads exist within resources of a single process, without which they cannot exist.
  - Usually, the number of threads match the number of machine processors/cores.
    - However, the actual use of threads is up to the application.

Parallelism in OpenMP (/2)

- *Explicit Parallelism*
  - OpenMP is an **<u>explicit</u>** (not automatic) programming model, offering the programmer control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives....
  - The general form of these are:

    ```
    #pragma omp construct [clause [clause]…]
    ```

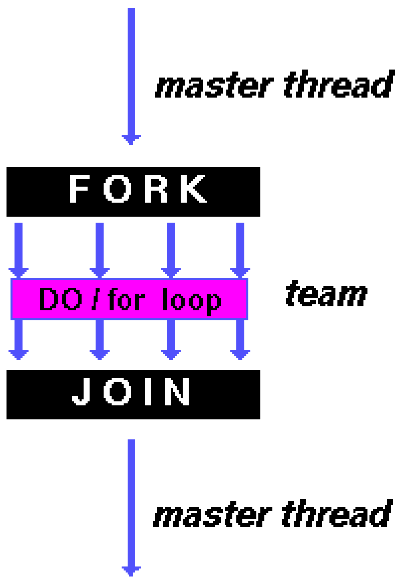
    - Example of this:

    ```
    #pragma omp parallel num_threads(4)
    ```

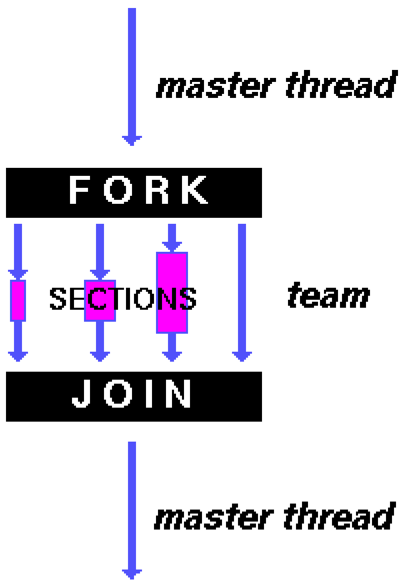
  - Note about `#pragma`
    - These are special preprocessor instructions.
    - Typically added to system to allow behaviours that aren't part of the basic language specification.
    - Compilers that don't support the pragmas ignore them.

OpenMP Work Constructs (Summary)

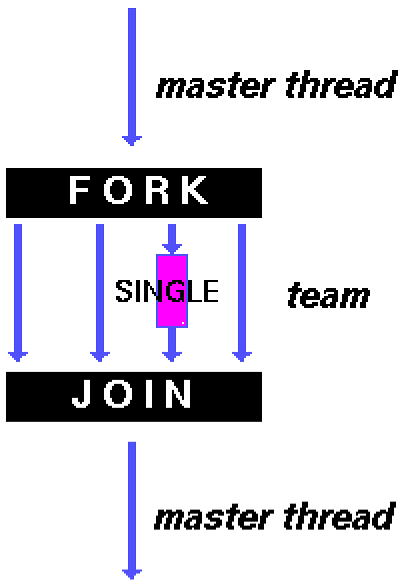
- `do` / `for` - shares loop iterations across team.
- Akin to "data parallelism"



- `sections` - breaks work into separate, discrete sections.
- Each executed by a thread.
- Can be used to implement a type of "functional parallelism".

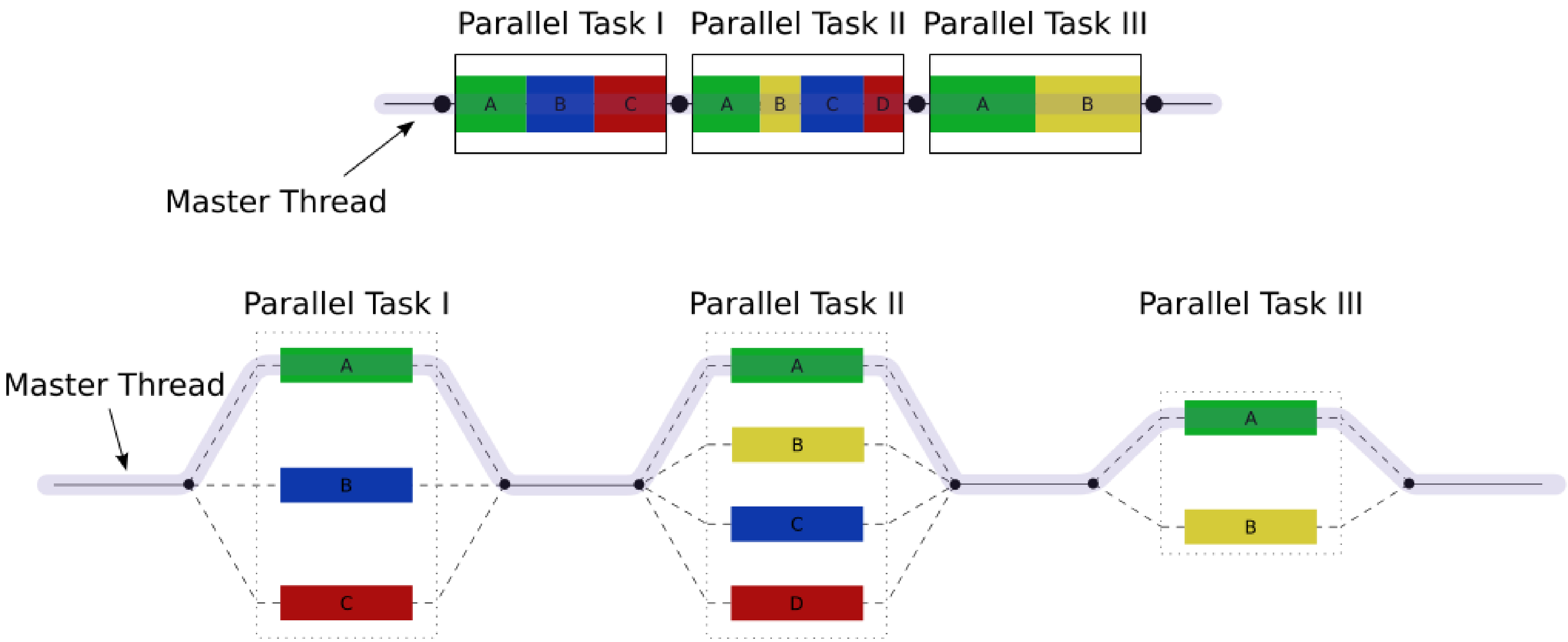


- `single` - serializes a chunk of code



OpenMP Programming Model (/2)

The *Fork-Join* Model



Example 1: My first OpenMP Code.

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel num_threads(2)
  printf("Hello, world.\n");
  return 0;
}
```

- *Thread Creation*
  - `pragma omp parallel` used to fork additional threads (here 2) to carry out the work enclosed in the construct in parallel.
  - The original thread is denoted as `master` thread with thread ID 0.
  - `num_threads(2)` is one of a number of clauses that can be specified .... e.g. `private` variables, `shared` variables, `reduction` operation
  - Simple Example: Display "Hello, world." using multiple threads.
  - Complex: insert subroutines to set multiple levels of parallelism, locks and even nested locks.

Example 1: My first OpenMP Code (/2).

- *Thread Creation (/2)*
  - When a thread reaches a `parallel` directive, creates a team of threads & becomes master of the team.
  - From the start of this parallel region, code is duplicated, and all threads execute that code.
  - Implicit barrier at the end of a parallel section.
    - Only the master thread continues execution past this point.
  - If any thread terminates in a parallel region, all threads in team stop
  - If this happens, the work done up until that point is *undefined*.

Running this Example in OpenMP

- Use flag `-fopenmp` to compile using GCC:

```
$ gcc -fopenmp hello.c -o hello
```

- Outputs on a computer with 2 cores, and thus 2 threads:

```
Hello, world.
Hello, world.
```

- However, output may also be garbled due to race condition caused from the two threads sharing the standard output:

```
Hello, wHello, woorld.
rld.
```

- A helpful step by step example on how to run can be found at `https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html`

Example 2: More Complex OpenMP Code.

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a[100];
  #pragma omp parallel for
  for (int i = 0; i < 100; i++)
    a[i] = 2 * i;
  return 0;
}
```

- Work-sharing constructs
  - `omp for`/ `omp do` for forking extra threads to do work enclosed in `parallel` (aka *loop* constructs).
  - This is equivalent to:

```
{   // stuff here
#pragma omp parallel

#pragma omp for
for (int i = 0; i < 100; i++)
  a[i] = 2 * i;
return 0;
}
```

Example 3: Data Dependencies

- Data on one thread can be dependent on data on another one
- This can result in wrong answers
  - Thread 0 may require a variable that is calculated on thread 1
  - Answer depends on timing – When thread 0 does the calculation, has thread 1 calculated its value yet?
- Example – Fibonacci Sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ more bunnies!

- Parallelize on 2 threads
  - Thread 0 gets $i = 2$ to 25, Thread 1 gets $i = 25$ to 49
    - Look carefully at calculation for $i = 49$ on thread 1
    - What will be values of for $i - 1$ and $i - 2$?

```
unsigned long A[50];
A[0] = 0;
A[1] = 1;
for(int i = 2; i <= 49; i++){
    A [i] =  A[i-1] + A[i-2];
}

printf("49th val is %ld\n", A[49]);
```

Data Dependencies (/2)

- *A Test for Dependency:*
  - If serial loop is executed in reverse order, will it give same result?
  - If so, it's (probably) okay
  - You can test this on your serial code

- What about subprogram calls?

```
for(i = 0; i < 100; i++){
    mycalc(i,&x,&y);
}
```

- Does the subprogram *write* **`x`** or **`y`** to memory?
  - If so, they need to be private
  - Variables local to subprogram are local to each thread
- Be careful with global variables and common blocks

Other Work Constructs in OpenMP

- **`sections`**

  Used to assign consecutive but independent code blocks to different threads

- **`single`**

  Specifying a code block that is executed by only one thread, a barrier is implied in the end

  Uses first thread that encounters the construct.

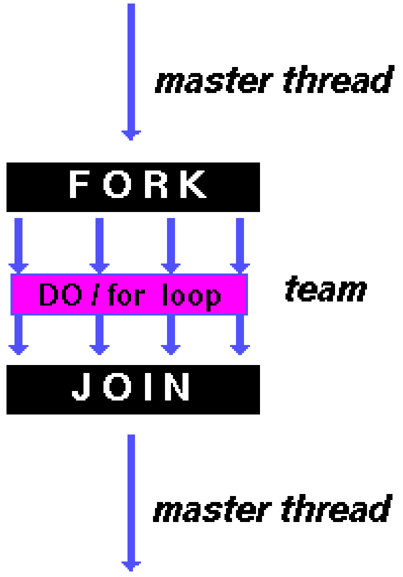
- **`master`**

  Similar to single, but code block is executed by **`master`** thread only – all others skip it

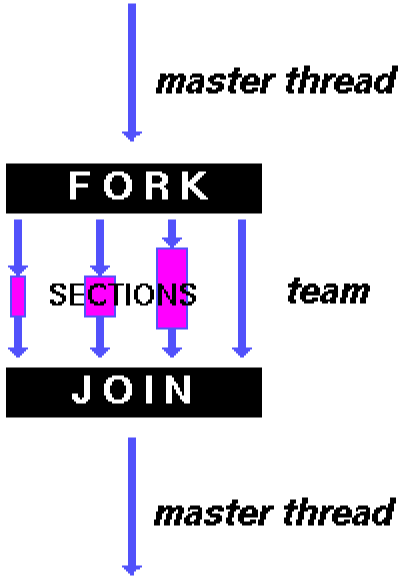
  No barrier implied in the end.

OpenMP Work Constructs (Summary)

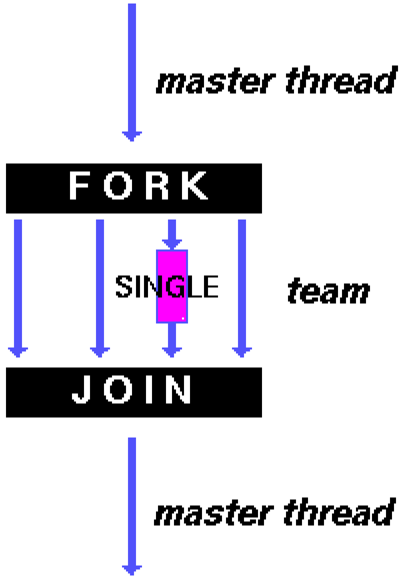
- `do` / `for` - shares loop iterations across team.
- Akin to "data parallelism"



- `sections` - breaks work into separate, discrete sections.
- Each executed by a thread.
- Can be used to implement a type of "functional parallelism".



- `single` - serializes a chunk of code



Example 4: **Sections** Construct.

```
void XAXIS(); void YAXIS(); void ZAXIS();
void sect_example()
{
#pragma omp parallel sections
{
  #pragma omp section
        XAXIS();
  #pragma omp section
        YAXIS();
  #pragma omp section
        ZAXIS();
}
```

- Purpose
  - This `sections` directive is used to execute routines **`XAXIS`**, **`YAXIS`**, and **`ZAXIS`** concurrently

Example 5: `single` Construct.

```
#include <stdio.h>
void work1() {}
void work2() {}
void single_example() {
#pragma omp parallel
         {
         #pragma omp single
                  printf("Beginning work1.\n");
                  work1();
         #pragma omp single
                  printf("Finishing work1.\n");
         #pragma omp single nowait
                  printf("Finished work1, starting work2.\n");
                  work2();
         }
}
```

- Purpose
  - `single` directive specifies that the enclosed code is to be executed by only one thread in the team.
  - Useful dealing with sections of code that are not thread safe (such as I/O)
  - There is **an implicit barrier** at end of each except where a `nowait` clause is specified

Synchronisation Constructs in OpenMP

- **`atomic`**

  Commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

- **`critical`**

  Specifies a critical section i.e. a region of code that must be executed by only one thread at a time.

- **`barrier`**

  Synchronizes all threads in the team.

  When reached, a thread waits there until all other threads have reached that barrier.

  All then resume executing in parallel the code that follows the barrier.

- **`master`**

  Strictly speaking, **`master`** is a synchronisation directive - master thread only and no barrier implied in the end. Identifies a section of code that must be run only by the master thread

Example 6: Data Scope Attributes

```
#include <stdio.h>
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic //
    b += a; // one thread can't interrupt another here
}
```

- *Purpose*
  - These attribute clauses specify data scoping/ sharing.
  - As OpenMP based on shared memory programming model, most variables shared by default.
  - Used with directives e.g. `Parallel`, `Do`/ `for`, `Sections` to control the scope of enclosed variables.
  - `a` is explicitly specified `private` (each thread has own copy) and `b` is `shared` (each thread accesses same variable).

Example 7: A more Complex HelloWorld

```
#include <iostream>
using namespace std;
#include <omp.h>
int main(int argc, char *argv[])
{
  int th_id, nthreads;
  #pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num(); // returns thread id
    #pragma omp critical // only one thread can access this at a time!
    {
      cout << "Hello World from thread " << th_id << "\n";
    }

    #pragma omp barrier // one thread waits for all others
    #pragma omp master  // master thread access only!
    {
      nthreads = omp_get_num_threads(); // returns number of thread
      cout << "There are " << nthreads << " threads" << " \n";
    }
  }
  return 0;
}
```

- *Purpose*
  - `Private`, `shared` declares that threads have their own copy of the variable or share a copy, respectively.

Reduction Clauses

- *Reduction*
  - (Like MPI – you'll see later), OpenMP supports the Reduction operation.

```
int t;
#pragma omp parallel reduction(+:t)
{
    t = omp_get_thread_num() + 1;
    printf("local %d\n", t);
}
printf("reduction %d\n", t);
```

  - Reduction Operators: `+ * -` logical operators and `Min()`, `Max()`
  - The operation makes the specified variable private to each thread.
  - At the end of the computation it combines private results
  - Very useful when combined with `for` as shown below see below:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i=0; i < 100; i++) {
    sum += array[i];
}
```

Common Mistakes in OpenMP:
#1 Missing **`Parallel`** keyword

```
#pragma omp for //this is incorrect as parallel keyword omitted
... // your code
```

- The code fragment will be successfully compiled, and the `#pragma omp for` directive will be simply ignored by the compiler.
- So only one thread executes the loop, and it could be tricky for a developer to uncover.
- The correct form should be:

```
#pragma omp parallel //this is correct
{
 #pragma omp for
 ... //your code
}
```

Common Mistakes in OpenMP: #2 Missing **for** keyword

- **#pragma omp parallel**
- This directive may be applied to a single code line as well as to a code fragment. This may cause unexpected behaviour of the `for` loop:

```
#pragma omp parallel num_threads(2) // incorrect as for keyword omitted
for (int i = 0; i < 10; i++)
  myFunc();
```

- If the developer wanted to share the loop between two threads, they should use the `#pragma omp parallel for` directive.
- Here the loop would have been executed 10 iterations x 2 threads.
- However, the code above will be executed once in every thread. As the result, the `myFunc();` function will be called 20 times.
- The correct version of the code is provided below:

```
#pragma omp parallel for num_threads(2) // now correct
for (int i = 0; i < 10; i++)
  myFunc();
```

Common Mistakes in OpenMP: #3 Redundant Parallization

- Applying the `#pragma omp parallel` directive to a large code fragment can lead to unexpected behaviour in cases like below:

```
#pragma omp parallel num_threads(2)
{
        ... // some lines of code
        #pragma omp parallel for
        for (int i = 0; i < 10; i++)
        {
                myFunc();
        }
}
```

- A naïve programmer wanting to share the loop execution between two threads placed the `parallel` keyword inside a parallel section.
- The result of execution is similar to previous example: the `myFunc` function will be called 20 times, not 10.
- The correct version of the code is the same as the above except for

```
...
#pragma omp parallel for
for (int i = 0; i < 10; i++) ...
```

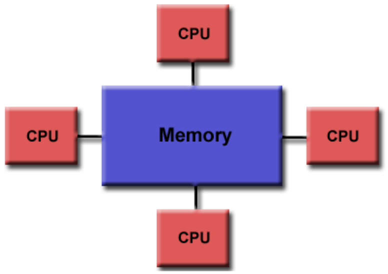
References

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
7. Our Pattern Language, Berkley 2019, https://patterns.eecs.berkeley.edu/

- We’ll talk about OpenMPI, even though it’s distributed, as it fits with OpenMP – it makes sense to explain them together!
- I’ll introduce it from a practical low/level, and then formalise all of this later.

Basics of Message Passing

Introduction to Message Passing



- To now concurrency constructs[1] based on shared memory systems
  - But for n/w architectures & distributed systems where processors are only linked by a comms medium, message passing is more common

- In message passing the processes which comprise a concurrent program are linked by *channels*.

- If the 2 interacting processes are on the same processor
  - *Channel* could simply be the processor's local memory.
- If the 2 processes are on separate processors
  - Channel between them is a physical comms medium (network) between corresponding 2 processors

[1]e.g. critical sections, semaphores, monitors, ...

Communication

- Expensive
- In general, we have a hierarchy:
  - Computation is faster than
  - Communication which is faster than
  - I/O

- 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)
  - ...

Message Passing Constructs

- There are 2 basic message passing primitives, `send` & `receive`
  - **send** primitive: sends a message (data) on a specified channel from one process to another,
  - **receive** primitive: receives a message on a specified channel from other processes.

- NB Send has different semantics depending on whether the message passing is *synchronous* or *asynchronous*.

Asynchronous v Synchronous Communication

- Asynchronous: (non-blocking*)
  - Sender resumes execution as soon as the message is passed to the communication/middleware software

- Synchronous: sender is blocked* until
  - The OS or middleware notifies acceptance of the message, *or*
  - The message has been delivered to the receiver, *or*
  - The receiver processes it & returns a response

* Can have both blocking and non-blocking forms. A blocking send waits until the message is fully transmitted.

Synchronous Message Passing

- In synchronous message passing each channel forms a direct link between two processes.
- Suppose process A is sending data to process B:

  When process A executes **`send`** primitive

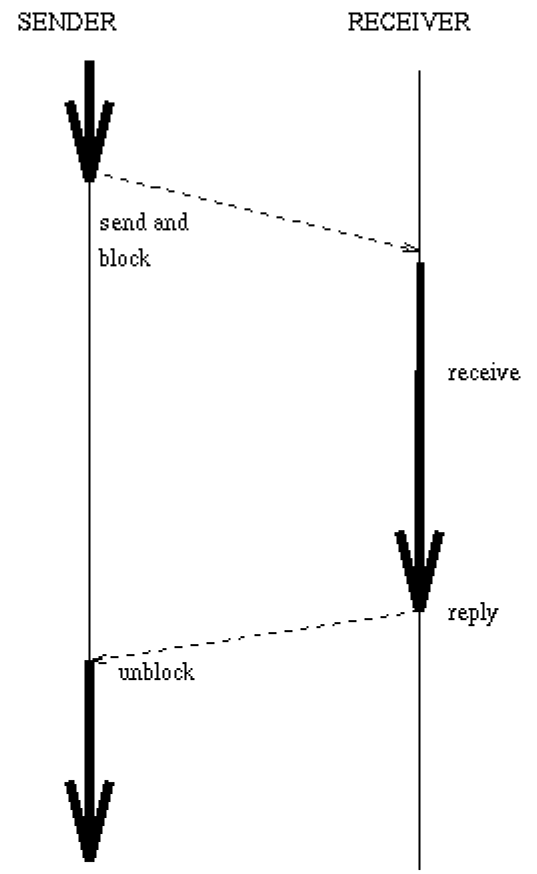
  it waits/blocks until process B executes

  its **`receive`** primitive.
- Before data can be sent both A & B must ready

  to participate in the exchange.
- Similarly the **`receive`** primitive in one process

  blocks until **`send`** primitive in the other

  process has been executed.



Asynchronous Message Passing

- In asynchronous message passing **`receive`** has the same meaning/behaviour as in synchronous
- **`but`** **`send`** primitive has different semantics.
- Now the channel between processes A & B isn't

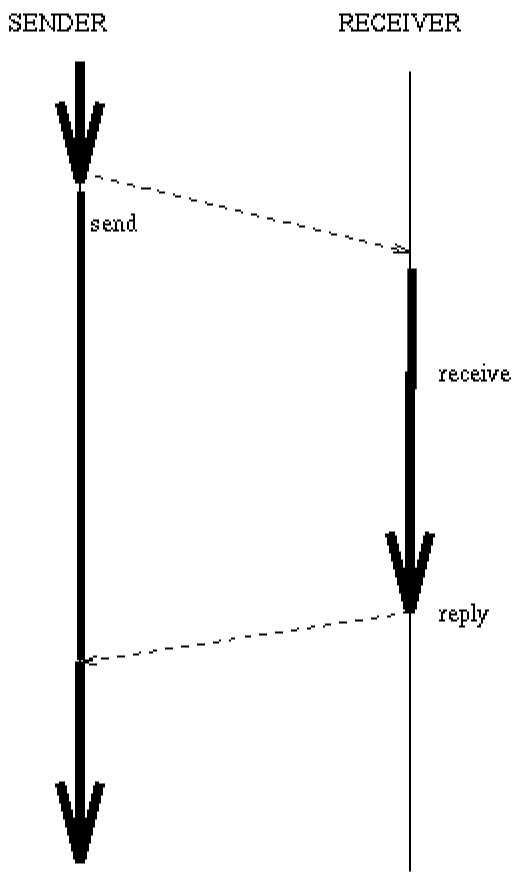
  a direct link but a message queue.
- Therefore when A sends a message to B, it is

  appended to the asynchronous channel's

  <u>message queue</u> and A continues.
- To receive a message from the channel, B executes

  a **`receive`**, taking the message at the head

  of the channel's queue and continuing.
- If there is no message in the channel **`receive`** blocks until some process adds a message to the channel.



Message Passing Interface (MPI)

Some background on MPI

- *MPI*
  - Developed by MPI forum (Industry, Academia & Govt.)
  - 1994: Set up a standardised Message-Passing Interface (MPI-1)
  - It was intended as an interface to both **C** and **FORTRAN**.
  - Aim was to provide a specification implementable on any parallel computer or cluster => **portability of code was a big aim**
- MPI provides support for:
  - Point-to-point & collective (i.e. group) communications
  - Inquiry routines to query the environment (how many nodes, what node number am I, etc.)
  - Constants and data-types
  - ...
- Start with basics: initialising MPI & using point-to-point comms

Some background on MPI

- MPI
  - Not a language (e.g., Java) or compiler specification (e.g., OpenMP)
  - Not a specific implementation or product
  - Supports SPMD (/MPMD) model (remember SIMD and MIMD?)
  - For parallel computers, clusters, heterogeneous networks and multicores
  - Supports many communication patterns
  - ...

Example 2: Exchanging 2 Values

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
        int myid, otherid, myvalue, othervalue, size, length = 1, tag = 1;
        MPI_Status status;
        /* initialize MPI and get own id (rank) */
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size!=2) {
                printf("use exactly two processes\n");
                exit(1);
        }
        if (myid == 0) {
                otherid = 1; myvalue = 14;
        }
        else {
                otherid = 0; myvalue = 25;
        }
        printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
        /* Send one integer to the other node (i.e. "otherid") */
        MPI_Send(&myvalue,1,MPI_INT,otherid,tag,MPI_COMM_WORLD);
        /* Receive one integer from any other node */
        MPI_Recv(&othervalue,1,MPI_INT,MPI_ANY_SOURCE,
        MPI_ANY_TAG,MPI_COMM_WORLD, &status);
        printf("process %d received a %d\n", myid, othervalue);
        MPI_Finalize();  /* Terminate MPI */
        return 0;
}
```

MPI Preliminaries…

- *Naming convention*
  - All MPI identifiers are prefixed by '**`MPI_`**'.
  - C routines contain lower case (i.e. '**`MPI_Init`**'),
  - Constants are upper case (e.g. '**`MPI_FLOAT`**' is MPI C data-type).
  - C routines are integer functions which return a status code (you should check these for errors!).
- *Compiling MPI*
  - Using openmpi implementation:

    *e.g., **mpicc hello.c –o hello**
- *Running MPI*
  - Number of processes is specified in the command line, when running MPI loader that loads MPI program onto the processes,
  - This is to avoid hard-coding it into the program

  *e.g. **`mpirun -np N exec`**

*May be different between MPI implementations...
OpenMPI vs **MPICH vs MVAPICH …**

Compiling and Running MPI Programs

- To compile programs using MPI, you need an “MPI-enabled” compiler (not your standard **gcc**)
- On cluster, use `mpicc` to compile C code with MPI or `mpic++` for C++.
- Before running an executable using MPI, ensure "multiprocessing daemon" (MPD) is running (*for MPICH*).... But different MPI implementations with different setups...
- It makes the workstations into (sort of) *virtual machines* to run MPI programs.

- Running MPI code, requests are sent to MPD daemons to start up copies of the program.
- Each copy then uses MPI to communicate with other copies of the same program running in the. To run the executable, type “`mpirun -np N./executable_file`”, where N is the number to be used to run the program.
- This value is then used in your program by `MPI_Init` to allocate the nodes and create the default communicator.

MPI Preliminaries… (/2)

- Writing a program using MPI: what is parallel, what is not
  - Only one program is written.
  - By default, each code line runs by each node running a program
  - And note.. if code contains `int result=0`, each node will locally create a variable and assign the value (**they don't share memory!)**

- We need to do things like:
  - When a section of the code needs to be executed by only a subset of nodes, should explicitly specify.
  - E.g., if using 8 nodes, and that **`MyID`** stores node rank (from 0 to 7), this bit of code assigns to **`result`** zero for the first half of them...

```
int result;
        if(MyID < 4) result = 0;
else result = 1;
...
```

Common MPI Routines

- MPI has a ‘kitchen sink’ approach of 129 different routines
- Most basic programs can get away with using less than 10.

- use `#include “mpi.h”` in C.

`MPI_Init`	Initialise MPI computation
`MPI_Finalize`	Terminate MPI computation
`MPI_Comm_size`	Determine number of processes
`MPI_Comm_rank`	Determine my process number
`MPI_Send, MPI_Isend`	Blocking, non-blocking send
`MPI_Recv, MPI_Irecv`	Blocking, non-blocking

Common MPI Routines (/2): MPI Initialisation, Finalization

- In all MPI programs, must initialise MPI before use & finalise at end
- Must handle all MPI-related commands, types in this section of code:

⋮

`MPI_Init`	Initialise MPI computation
`MPI_Finalize`	Terminate MPI computation

- `MPI_Init` takes two parameters as input (`argc` and `argv`),
  - It is used to start the MPI environment, create the default communicator (more later) and assign a rank to each node.
- `MPI_Finalize` cleans up all MPI state. Once this routine is called, no MPI routine (even `MPI_INIT`) may be called.
- The user must ensure that all pending communications involving a process completes before the process calls `MPI_Finalize`.

Common MPI Routines (/3): Basic Inquiry Routines

- At various stages in a parallel-implemented function, often **useful** to know **how many nodes** program is using, or what current node's **rank** is.
- `MPI_Comm_size` returns number of processes/ nodes as an integer, taking only one parameter, a communicator.
- Mostly only use the default Communicator: `MPI_COMM_WORLD`.
- The `MPI_Comm_rank` function is used to determine what the rank of the current process/node on a particular communicator.
- E.g. if there are two communicators, it is possible, and quite usual, that the ranks of the same node would differ.
- Again, in most cases, this function will only be used with the default communicator as an input (`MPI_COMM_WORLD`),
- It returns (as an integer) the rank of the node on that communicator.

A first MPI example: Hello World.

```
#include <mpi.h>
      int main(int argc, char *argv[]) {
            int myid, size;
            MPI_Init(&argc, &argv);
                  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
                  MPI_Comm_size(MPI_COMM_WORLD, &size);
                  printf("process %d out of %d says
                  Hello\n", myid, size);
            MPI_Finalize();
      return 0;
}
```

A first MPI example: Hello World.

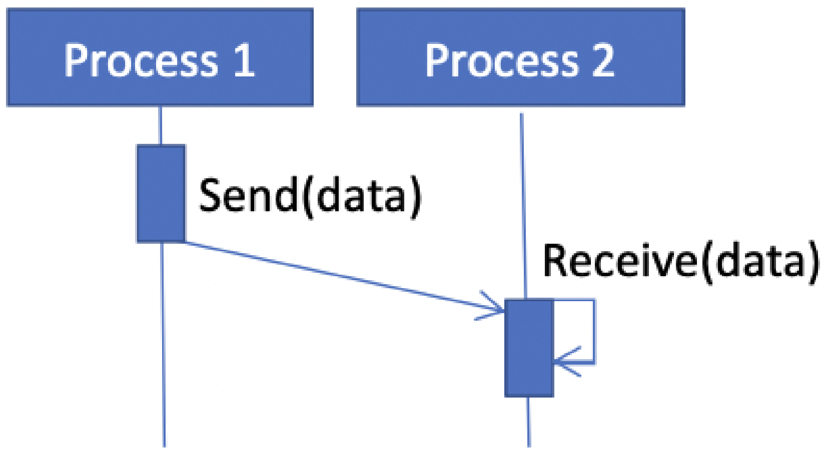
```
#include <mpi.h>
      int main(int argc, char *argv[]) {
            int myid, size;
            MPI_Init(&argc, &argv);
                  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
                  MPI_Comm_size(MPI_COMM_WORLD, &size);
                  printf("process %d out of %d says
                  Hello\n", myid, size);
            MPI_Finalize();
      return 0;
}
```

```
process 0 out of 6 says Hello
process 5 out of 6 says Hello
process 2 out of 6 says Hello
process 3 out of 6 says Hello
process 1 out of 6 says Hello
```

Point-to-Point communications in MPI

- Involves communication between two proceses, **one sending**, and **the other receiving**.

- Certain information is required for messaging:
  - Identification of sender process
  - Identification of destination/receiving process
  - Type of data (`MPI_INT`, `MPI_FLOAT` etc)
  - Number of data elements to send (i.e. array/vector info)
  - Where the data to be sent is in memory (pointer)
  - Where the received data should be stored in (pointer)



Common MPI Routines (/5): Sending data **`MPI_Send`**, **`MPI_Isend`**

- **`MPI_Send`** used for blocking send, (i.e. process waits for the communication to finish before going to the next command).

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- This function takes six parameters:
  - the location of the data to be sent i.e. a pointer (*input parameter*)
  - the number of data elements to be sent (*input parameter*)
  - the type of data e.g. **`MPI_INT`**, **`MPI_FLOAT`**, etc. (*input parameter*)
  - the rank of the receiving/destination node (*input parameter*)
  - a tag for identification of the communication (*input parameter*)
  - the communicator to be used for transmission (*input parameter*)
- **`MPI_Isend`** is non-blocking, so an additional parameter, to allow for verification of communication success is needed.
  - It is a pointer to an element of type **`MPI_Request`**.

Common MPI Routines (/6): Receiving data `MPI_Recv`, `MPI_Irecv`

- `MPI_Recv` is used to perform a blocking receive, (i.e. process waits for the communication to finish before going to the next command).

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
comm, MPI_Status *status);
```

- This function takes seven parameters:
  - the location of the receive buffer i.e. a pointer (*output parameter*)
  - the max number of data elements to be received (*input parameter*)
  - the type of data e.g. `MPI_INT`, `MPI_FLOAT`, etc. (*input parameter*)
  - the rank of the source/sending node (*input parameter*)
  - a tag for identification of the communication (*input parameter*)
  - the communicator to be used for transmission (*input parameter*)
  - a pointer to a structure of type `MPI_Status`, contains source processor's rank, communication tag, and error status (*output parameter*)
- For the non-blocking `MPI_Irecv`, `MPI_Request` replaces `MPI_Status`.

Example 2: Exchanging 2 Values

```
#include <mpi.h>
int main(int argc, char *argv[]) {
        int myid, otherid, myvalue, othervalue, size, length = 1, tag = 1;
        MPI_Status status;
        /* initialize MPI and get own id (rank) */
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size!=2) {
                printf("use exactly two processes\n");
                exit(1);
        }
        if (myid == 0) {
                otherid = 1; myvalue = 14;
        }
        else {
                otherid = 0; myvalue = 25;
        }
        printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
        /* Send one integer to the other node (i.e. "otherid") */
        MPI_Send(&myvalue,1,MPI_INT,otherid,tag,MPI_COMM_WORLD);
        /* Receive one integer from any other node */
        MPI_Recv(&othervalue,1,MPI_INT,MPI_ANY_SOURCE,
        MPI_ANY_TAG,MPI_COMM_WORLD, &status);
        printf("process %d received a %d\n", myid, othervalue);
        MPI_Finalize();  /* Terminate MPI */
        return 0;
}
```

Example 2: Exchanging 2 Values

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
        int myid, otherid, myvalue, othervalue, size, length = 1, tag = 1;
        MPI_Status status;
        /* initialize MPI and get own id (rank) */
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size!=2) {
                printf("use exactly two processes\n");
                exit(1);
        }
        if (myid == 0) {
                otherid = 1; myvalue = 14;
        }
        else {
                otherid = 0; myvalue = 25;
        }
        printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
        /* Send one integer to the other node (i.e. "otherid") */
        MPI_Send(&myvalue,1,MPI_INT,otherid,tag,MPI_COMM_WORLD);
        /* Receive one integer from any other node */
        MPI_Recv(&othervalue,1,MPI_INT,MPI_ANY_SOURCE,
        MPI_ANY_TAG,MPI_COMM_WORLD, &status);
        printf("process %d received a %d\n", myid, othervalue);
        MPI_Finalize();  /* Terminate MPI */
        return 0;
}
```

```
mpirun –n 2 ./myprog
process 0 sending 14 to process 1
process 1 sending 25 to process 0
process 0 received a 25
process 1 received a 14
```

Example 3:“Ring” Communication

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
        int rank, value, value2, size;
        MPI_Status status;
        /* initialize MPI and get own id (rank) */
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size);
        do {
                if (rank == 0) {
                        scanf("%d", &value );
                        /* Master Node sends out the value */
                        MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
                        /* received value at end */

                        MPI_Recv( &value2, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD,&status);
                        printf("master process %d got %d\n", rank, value2);
                }
                else {
                /* Slave Nodes block on receive then send on the value */
                MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
                if (rank < size - 1)
                        MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
                else if (rank == size-1)
                        MPI_Send( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                }
                printf("process %d got %d\n", rank, value);
        } while (value >= 0);
        /* Terminate MPI */
        MPI_Finalize();
        return 0;
}
```

Example 4: Matrix-Vector Product Implementation

```
#include <mpi.h>
int main(int argc, char *argv[]) {
int A[4][4], b[4], c[4], line[4], temp[4], local_value, myid;
MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myid);
if (myid == 0) {
    for (int i=0; i<4; i++) {
           b[i] = 4 - i;
           for (int j=0; j<4; j++)
                      A[i][j] = i + j; /* set some notional values for A, b */
    }
    line[0]=A[0][0]; line[1]=A[0][1];
    line[2]=A[0][2]; line[3]=A[0][3];
}
if (myid == 0) {
 for (int i=1; i<4; i++) {/* slaves do most of the multiplication */
           temp[0]=A[i][0];temp[1] = A[i][1];temp[2] = A[i][2];temp[3] = A[i][3];
           MPI_Send( &temp, 4, MPI_INT, i, i, MPI_COMM_WORLD);
           MPI_Send( &b, 4, MPI_INT, i, i, MPI_COMM_WORLD);
           }
}
else {
           MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
           MPI_Recv( b, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}  /* master node does its share of multiplication too*/
c[myid] = line[0] * b[0] + line[1] * b[1] + line [2] * b[2] + line[3] * b[3];
if (myid != 0) {
           MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
}
else {
 for (int i=1; i<4; i++) {
           MPI_Recv( &c[i], 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
MPI_Finalize();
return 0;
}
```

Why 1?

Example : matrix-vector product

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}^{\text{temp}} \text{X} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}^{\text{b}} = \begin{pmatrix} \boxed{c_1} \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

with

$$\begin{cases} c_1 = \boxed{a_{11} \times b_1} + a_{12} \times b_2 + a_{13} \times b_3 + a_{14} \times b_4 \\ c_2 = a_{21} \times b_1 + a_{22} \times b_2 + a_{23} \times b_3 + a_{24} \times b_4 \\ c_3 = a_{31} \times b_1 + a_{32} \times b_2 + a_{33} \times b_3 + a_{34} \times b_4 \\ c_2 = a_{41} \times b_1 + a_{42} \times b_2 + a_{43} \times b_3 + a_{44} \times b_4 \end{cases}$$

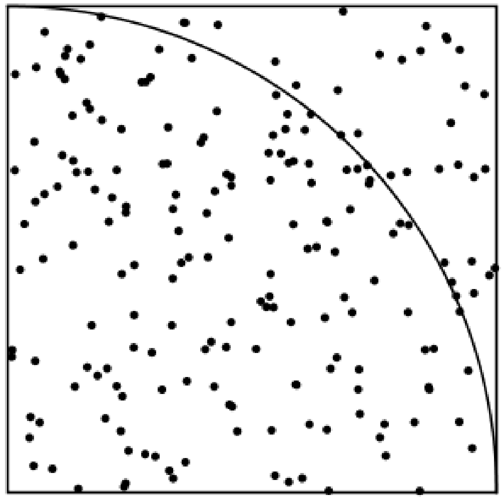
- A parallel approach:
  - Each element of vector *c* depends on vector *b* and only one row of *A*
  - So each *c* can be calculated independently from the others
  - Communication only needed to split problem & combine final results => a linear speed-up can be expected for large matrices

Example 5: Pi Calculation Implementation

```
int main(int argc, char *argv[]) {
        MPI_Init(&argc, &argv);
        int myid, size, inside=0, outside=0, points=10000;
        double x,y, Pi_comp, Pi_real=3.141592653589793238462643;
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (myid == 0) {
                for (int i=1; i<size; i++)   /* send out the value of points to all slaves */
                        MPI_Send(&points, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
        else
                MPI_Recv(&points, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        rands=new double[2*points];
        for (int i=0; i<2*points; i++ ){
                rands[i]=random(-1,1); // generate random between -1 and 1
        }
        for (int i=0; i<points;i++ ){
                x=rands[2*i];
                y=rands[2*i+1];
                if((x*x+y*y)<1) inside++    /* point is inside unit circle so incr var inside */
        }
        delete[] rands;
        if (myid == 0) {
                for (int i=1; i<size; i++) {
                        int temp;  /* master receives all inside values from slaves */
                        MPI_Recv(&temp, 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        inside+=temp;  }  /* master sums all insides sent to it by slaves */
        }
        else
                MPI_Send(&inside, 1, MPI_INT, 0, i, MPI_COMM_WORLD); /* send inside to master*/
        if (myid == 0) {
                Pi_comp = 4 * (double) inside / (double)(size*points);
                cout << "Value obtained: " << Pi_comp << endl << "Pi:" << Pi_real << endl;}
        MPI_Finalize(); return 0;
}
```

All nodes do this part

Example 5 : Monte-Carlo calculation of Pi



- *Monte Carlo Integration*
- π = 3.14159….= area of a circle of radius 1
- $\pi/4 \approx$ fraction of points in circle quadrant
- More points =>more accurate value for π
- A parallel approach:
  - Each point is randomly placed within the square & so each point's position is independent of the position of the others
  - Can split problem by letting each node randomly place a given number of points
  - Only need communicate number of points & take in final results
- => Can expect linear speed-up allowing for a larger number of points and hence greater accuracy in the estimation of π.

Some Sophisticated MPI Routines

- Advantage of global comms routines below is that MPI system implements them more efficiently than the coder – for us fewer function calls.
- Maybe some parallelism might be available to be exploited in the communications network too.

`MPI_Bcast`	Broadcast same data to all procs
`MPI_Reduce`	Combine data from all onto one proc
`MPI_Allreduce`	Combine data from all procs onto all procs
`MPI_Gather`	Get data from all procs
`MPI_Scatter`	Send different data to all procs
`MPI_Barrier`	Synchronise

Sophisticated MPI Routines: `MPI_Bcast`

- `MPI_Bcast` used to send data from one node to all the others in one single command.

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
```

- This function takes five parameters: -
  - location of data to be sent i.e. a pointer (*input/output* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data (*input* parameter)
  - rank of the broadcast node (*input* parameter)
  - communicator to be used (*input* parameter)

Example 6: A New Matrix-Vector Product

```
.
.
.
MPI_Bcast(b,4,MPI_INT,0,MPI_COMM_WORLD);
if (myid == 0) {
        for (int i=1; i<4; i++) {/* slaves do most multiplying */
                temp[0]=A[i][0];temp[1] = A[i][1];temp[2] = A[i][2];temp[3] =
                A[i][3];
                MPI_Send( temp, 4, MPI_INT, i, i, MPI_COMM_WORLD);
                /* No need to send vector b here */
        }
}
else {
        MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
        /* No need to receive vector b here */
}  {/* master node does its share of multiplication too*/
c[myid] = line[0] * b[0] + line[1] * b[1] + line [2] * b[2] + line[3] * b[3];
if (myid != 0) {MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);}
else {
 {
        for (int i=1; i<4; i++) {
                MPI_Recv( &c[i], 1, MPI_INT, i, i, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
}
MPI_Finalize(); return 0;
```

Example 4: Matrix-Vector Product Implementation

```
#include <mpi.h>
int main(int argc, char *argv[]) {
int A[4][4], b[4], c[4], line[4],temp[4], local_value, myid;
MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myid);
if (myid == 0) {
    for (int i=0; i<4; i++) {
          b[i] = 4 - i;
          for (int j=0; j<4; j++)
                      A[i][j] = i + j; /* set some notional values for A, b */
    }
    line[0]=A[0][0]; line[1]=A[0][1];
    line[2]=A[0][2]; line[3]=A[0][3];
}
if (myid == 0) {
 for (int i=1; i<4; i++) {/* slaves do most of the multiplication */
          temp[0]=A[i][0];temp[1] = A[i][1];temp[2] = A[i][2];temp[3] = A[i][3];
          MPI_Send( &temp, 4, MPI_INT, i, i, MPI_COMM_WORLD);
          MPI_Send( &b, 4, MPI_INT, i, i, MPI_COMM_WORLD);
          }
}
else {
          MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          MPI_Recv( b, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}  /* master node does its share of multiplication too*/
c[myid] = line[0] * b[0] + line[1] * b[1] + line [2] * b[2] + line[3] * b[3];
if (myid != 0) {
          MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
}
else {
 for (int i=1; i<4; i++) {
          MPI_Recv( &c[i], 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
MPI_Finalize();
return 0;
}
```

Sophisticated MPI Routines: **`MPI_Reduce`**

- **`MPI_Reduce`** used to reduce values on all nodes of a group to a single value on one node using some reduction operation (sum etc).

```
int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm )
```

- This functions takes seven parameters: -
  - location of the data to be sent i.e. a pointer (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output parameter*)
  - number of elements to be sent (*input* parameter)
  - type of data e.g. **`MPI_INT`**,**`MPI_FLOAT`**, etc. (*input* parameter)
  - operation to combine the results e.g. **`MPI_SUM`** (*input* parameter)
  - identity of root (i.e. receiving) node (*input* parameter)
  - communicator used for transmission (*input* parameter)


```
MPI_Reduce(&inside,&total,1,MPI_INT,MPI_SUM,0, MPI_COMM_WORLD);
```

Example 7: A New Pi Calculation Implementation

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int myid, size, inside=0, outside=0, points=10000;
    double x,y, Pi_comp, Pi_real=3.141592653589793238462643;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
          /* Again send/receive replaced by MPI_Bcast */
    MPI_Bcast(&points,1,MPI_INT, 0, MPI_COMM_WORLD);
    rands=new double[2*points];
    for (int i=0; i<2*points; i++ ){
          rands[i]=random(-1, 1);
          }
    for (int i=0; i<points;i++ ){
          x=rands[2*i];
          y=rands[2*i+1];
          if((x*x+y*y)<1) inside++      /* point is inside unit circle so incr var inside */
    }
    delete[] rands;

    if (myid == 0) {
          for (int i=1; i<size; i++) {
             int temp;                  /* master gets all inside values from slaves */
             MPI_Recv(&temp, 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
             inside+=temp;    }         /* master sums all insides sent to it by slaves */
    }
    else
          MPI_Send(&inside, 1, MPI_INT, 0, i, MPI_COMM_WORLD); /* send inside to master */

    MPI_Reduce(&inside,&total,1,MPI_INT,MPI_SUM,0, MPI_COMM_WORLD);
    if (myid == 0) {
          Pi_comp = 4 * (double) inside / (double)(size*points);
          cout << "Value obtained: " << Pi_comp << endl << "Pi:" << Pi_real << endl;}
    MPI_Finalize(); return 0;
}
```

Reduction Pattern

- **MPI_REDUCE -** Reduction combines data from all processors and returns to a single process
  - Can apply associative operation on the gathered data eg ADD, OR, Min, Max
  - No process can finish reduction until all have contributed a value

- For many numerical algorithms, SEND/RECEIVE can be replaced by Broadcast/Reduce to simplify code

Reduce operations (MPI_Op's)

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

From: https://www.open-mpi.org/doc/v4.1/man3/MPI_Reduce.3.php

Sophisticated MPI Routines: **`MPI_Allreduce`**

- **`MPI_Allreduce`** is used to reduce values on all group nodes to a one value, and send it back to all (i.e. equals **`MPI_Reduce+MPI_Bcast`**)

```
int MPI_Allreduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm )
```

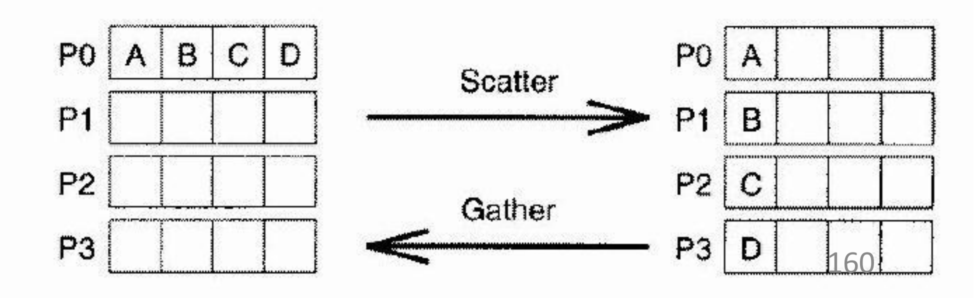
- This functions takes six parameters: -
  - location of the data to be sent i.e. a pointer (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output parameter*)
  - number of elements to be sent (*input* parameter)
  - type of data e.g. **`MPI_INT`**,**`MPI_FLOAT`**, etc. (*input* parameter)
  - operation to combine the results e.g. **`MPI_SUM`** (*input* parameter)
  - communicator used for transmission (*input* parameter)

Sophisticated MPI Routines: **`MPI_Scatter`**

- **`MPI_Scatter`** used to scatter data from single node to a group

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- This function takes eight parameters: -
  - location of data to be sent i.e. a pointer (*input* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data to be sent (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output parameter*)
  - number of elements to be received (*input* parameter)
  - type of data to be received (*input* parameter)
  - rank of the sending node (*input* parameter)
  - communicator to be used for transmission. (*input* parameter)

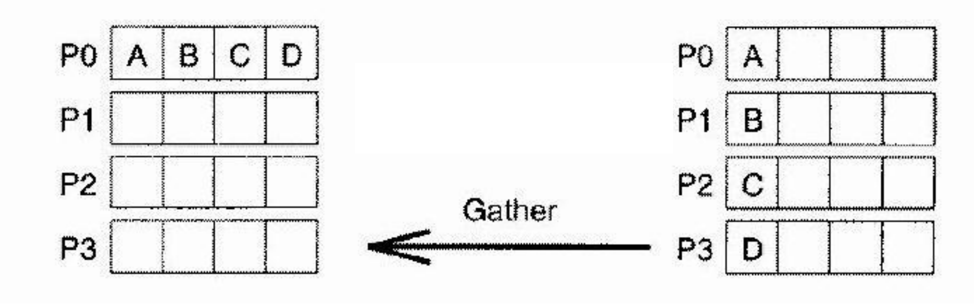


Sophisticated MPI Routines: **`MPI_Gather`**

- **`MPI_Gather`** used to gather on 1 node data scattered over a group .

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- This functions takes eight parameters: -
  - location of data to be sent i.e. a pointer (*input* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data to be sent (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output parameter*)
  - number of elements to be received (*input* parameter)
  - type of data to be received (*input* parameter)
  - rank of the sending node (*input* parameter)
  - communicator to be used for
    transmission. (*input* parameter)



Example: Scatter / Gather Averaging

```
....
MPI_Init()

int myid = 0; int size;
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &size);
num_elements_per_proc = 100; // number of nums each node should average

...
if (myid == 0) {
        float *rand_nums = NULL;
        rand_nums = (float *) random_numbers(num_elements_per_proc * size); // generate rand numbers
}

float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc); // memory per node to store these



MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums, num_elements_per_proc, MPI_FLOAT, 0,
MPI_COMM_WORLD);

float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);

if (myid == 0) {
   sub_avgs = (float *)malloc(sizeof(float) * size);
   assert(sub_avgs != NULL);
}

MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

if (myid == 0) {
   float avg = compute_avg(sub_avgs, world_size);
   printf("Avg of all elements is %f\n", avg);
}
MPI_Finalize();
```

Adapted from: https://raw.githubusercontent.com/mpitutorial/mpitutorial/gh-pages/tutorials/mpi-scatter-gather-and-allgather/code/avg.c

Sophisticated MPI Routines: **`MPI_Barrier`**

- **`MPI_Barrier`** is used to synchronise a set of nodes.

```
int MPI_Barrier( MPI_Comm comm )
```

MPI_Barrier (MPI_COMM_WORLD);

- It blocks the caller until all group members have called it.
- i.e., call returns at any process only after all group members have processed the call.
- This functions takes only parameter, the communicator (i.e. group of nodes) to be synchronised.
- As we previously saw with other functions, it will most of the times be used with the default communicator, **`MPI_COMM_WORLD`**.

Collective communications in MPI

- Groups are sets of processors interacting with each other in certain way.
- Such communications permit a more flexible mapping of the language to the problem (allocation of nodes to subparts of the problem etc).
- MPI implements Groups using data objects called Communicators.
- A special Communicator is defined (called ‘`MPI_COMM_WORLD`’) for the group of all processes.
- Each Group member is identified by a number (its Rank 0..n-1).
- There are three steps to create new communication structures:
  - accessing the group corresponding to MPI_COMM_WORLD,
  - using this group to create sub-groups,
  - allocating new communicators for this group.

Using Communicators

- Creating a new group (and communicator) by excluding the first node:

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    .
    .
    .
    .
    MPI_Comm comm_world, comm_worker;
    MPI_Group group_world, group_worker;
    comm_world = MPI_COMM_WORLD;
    MPI_Comm_group(comm_world, &group_world);
    MPI_Group_excl(group_world, 1, &[0], &group_worker);
               /* process 0 not member */
    MPI_Comm_create(comm_world, group_worker, &comm_worker);
    .
    .
    .
    .
}
```

- **Warning:**

`MPI_Comm_create()` is a collective operation, so all processes in the old communicator must call it.

Example 8: Using Communicators

```
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8
int main(int argc, char *argv[]) {

    int rank, newrank, sendbuf, recvbuf;
    ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm new_comm;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;
                /* Extract the original group handle */
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
    if (rank < NPROCS/2) {/* Split tasks into 2 distinct groups based on rank */
      MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
    else
      MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
    /* Create new communicator and then perform collective communications */
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
    MPI_Group_rank (new_group, &new_rank);

    printf("rank= %d newrank= %d recvbuf= %d\n", rank, newrank, recvbuf);

    MPI_Finalize();
}
```

The best of Both Worlds…
Hybridization of MPI & OpenMP

MPI & OpenMP: Advantages & Disadvantages

- Pure MPI Pros:
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
- Pure MPI Cons:
  - Difficult to develop and debug
  - Explicit communication
  - High latency, low bandwidth
  - Coarse granularity
  - Difficult load balancing

- Pure OpenMP Pros:
  - Easy to implement parallelism
  - Coarse[1] and [2]fine granularity
  - Implicit Communication
  - Low latency, high bandwidth
  - Dynamic load balancing
- Pure OpenMP Cons:
  - Only on shared memory machines
  - Scale within one node
  - ...

[1]e.g. Parallel Regions [2]e.g. Loop-level

Hybridization: What it is & How it Helps

- *What it is:*
  - Using inherently different models of programming in a complimentary manner, to achieve a benefit not possible otherwise.
  - A way to use different models of parallelization in a way that takes advantage **<u>of the good points of each</u>**.
  - Hybrid MPI/OpenMP paradigm is *used for* clusters comprised of SMP machines.
  - Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes.
  - Good usage of shared memory system resource (memory, latency, and bandwidth).

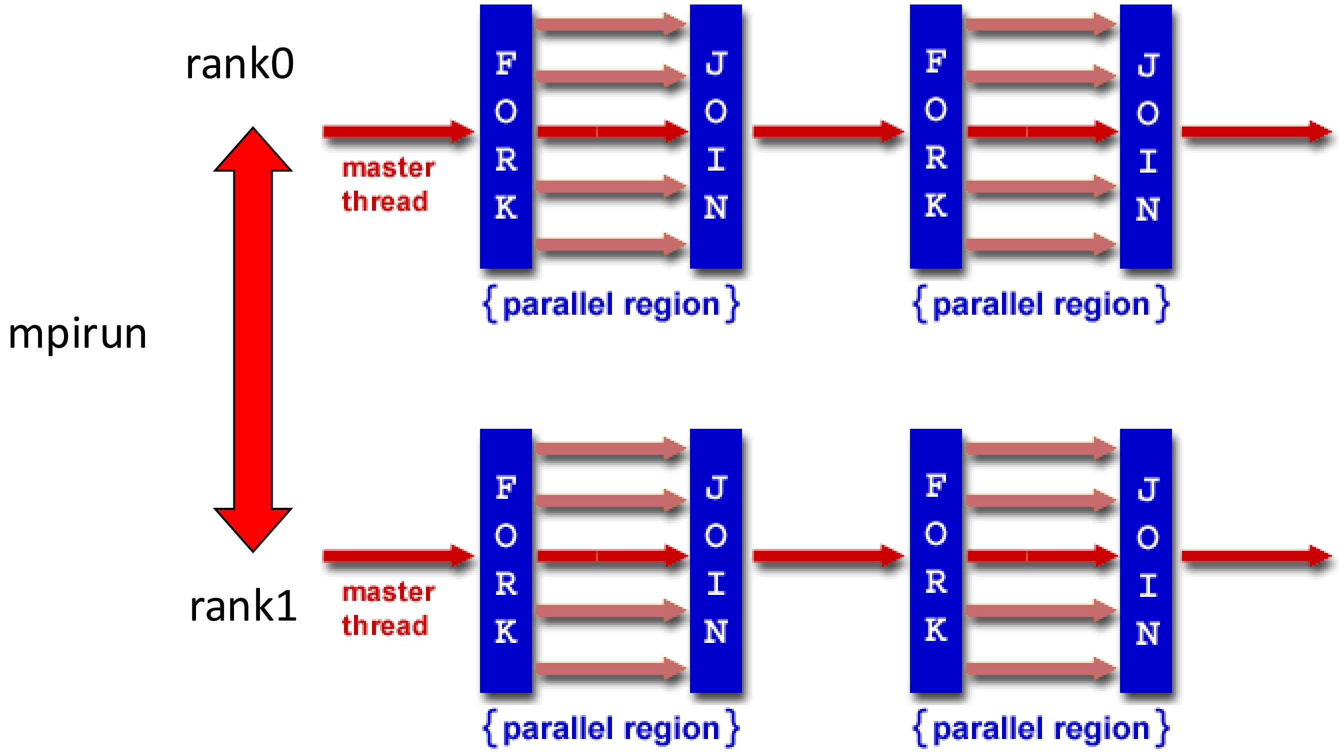
Hybridization: How it Helps

- ***Generalities:***
  - Some problems have **two-level parallelism** naturally.
  - *Could have* better scalability than both pure MPI and pure OpenMP.

- ***How it helps:***
  - Adding MPI to OpenMP code can help scale across multiple (typically SMP) nodes;
  - Adding OpenMP to MPI code can use shared memory on nodes more efficiently, and **reduce explicit intra-node communication** needs;
  - Adding MPI & OpenMP in design/coding of a program can help maximize efficiency, performance, and scaling;
  - Ultimately, avoiding the extra communication overhead with MPI within node.

Hybrid MPI + OpenMP programming

- Each MPI process spawns multiple OpenMP threads



Hybrid MPI + OpenMP Example 1:

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>
int main(int argc, char *argv[]) {
 int numprocs, rank, namelen;
 char processor_name[MPI_MAX_PROCESSOR_NAME];
 int iam, np;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Get_processor_name(processor_name, &namelen);
 #pragma omp parallel default(shared) private(iam, np)
 {
       np = omp_get_num_threads();
       iam = omp_get_thread_num();
       printf("Hello from thread %d out of %d from process %d out of %d
       on %s\n", iam, np, rank, numprocs, processor_name);
 }
 MPI_Finalize();
}
```

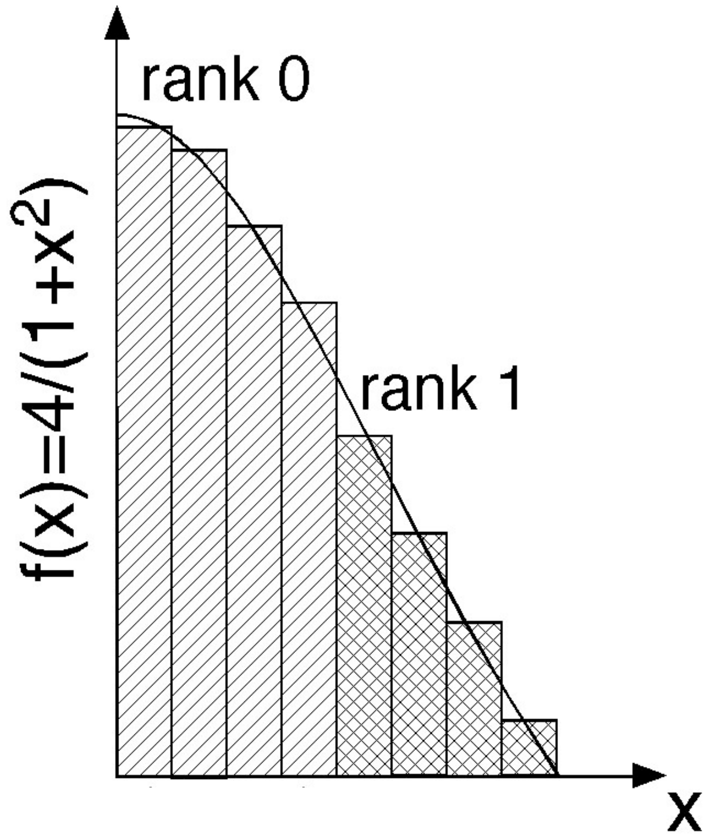

177

Hybrid MPI + OpenMP Example 1 (/2): Sample Output

```
Hello from thread 0 out of 4 from process 0 out of 2 on xt1
Hello from thread 1 out of 4 from process 0 out of 2 on xt1
Hello from thread 2 out of 4 from process 0 out of 2 on xt1
Hello from thread 3 out of 4 from process 0 out of 2 on xt1
Hello from thread 0 out of 4 from process 1 out of 2 on xt2
Hello from thread 3 out of 4 from process 1 out of 2 on xt2
Hello from thread 1 out of 4 from process 1 out of 2 on xt2
Hello from thread 2 out of 4 from process 1 out of 2 on xt2
```

Hybrid MPI + OpenMP Example 2: Calculate π

- $\int_0^1 \frac{dx}{1+x^2} = \tan^{-1} 1 = \frac{\pi}{4}$
- Integrating the function f(x) from [0,1] will give approximation to π
- Each MPI process integrates over a range of width **`1/numproc`**, as a discrete sum of **`num_steps`** steps, each of width **`step`**
- In each MPI process, **`num_steps`** OpenMP threads perform part of the sum in OPENMP alone.



Hybrid MPI + OpenMP Example 2 (/2): **ompi.c** code

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel for reduction(+:sum) private(x)
   for (i = 0; i < num_steps; i++){
        x = i*step;             // scales x in terms of step
        sum = sum + 4.0/(1.0+x*x); // sum is private til threads done
   }
   pi = step * sum;
}
```

This is the OpenMP version
Of the Monte-Carlo
calculation on its own

Hybrid MPI + OpenMP Example 2 (/3)

mpipi.c

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;
        long num_steps = 100000;
        double x, pi, my_steps, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ; // divides num_steps among numprocs

        // each will get a bit of the range to do in its part of for loop
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = i*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
        MPI_Finalize();
}
```

This is the MPI version
Of the Monte-Carlo
calculation on its own

Hybrid MPI + OpenMP Example 2 (/4): mixpi.c

```
#include <mpi.h>
#include "omp.h"
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;
        long my_steps, num_steps = 100000;
        double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        #pragma omp parallel for private(x) reduction(+:sum)
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = i*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi,1,MPI_DOUBLE, MPI_SUM,0,MPI_COMM_WORLD);
        if (my_id==0) printf("Pi is %f\n", pi);
        MPI_Finalize(); /* Terminate MPI */
}
```

Get the MPI part done first, then add OpenMP pragma

Lecture Summary

- 2 Message Passing Primitives: Send & Receive with many different combos of each synch/asynch, persistent/transient
- Things to remember in MPI:
  - MPI programs need specific compilers (e.g. **`mpicc`**), **`MPD`**, **`mpirun`**.
  - Functions for point-to-point comms, 6 more advanced ones, to synchronise, and perform collective comms, ...
- Unlike MPI, OpenMP:
  - Facilitates incremental parallelization of a serial program, so doesn't require 'all or nothing' approach to parallelization,
  - MPI scales well but is non-trivial to implement for codes originally written for serial machines & not good for shared memory
  - Can implement both coarse-grain & fine-grain parallelism.
- Together, they can form a good team!