Thread Interaction (Bakery, Mutual Exclusion)

Created: *29/04/2018 1:42 PM*
Updated: *29/04/2018 2:01 PM*

Week 3: Thread Interaction
Two properties in concurrent programs that need to be correct:

1. Safety: means that synchronization & mutual exclusion are enforced.
2. Liveliness: should not die before it is meant to. Live-lock, deadlock and starvation shouldn't be enforced.

Mutual Exclusion: One thread should be in a critical section at a time.
Four methods introduced in lectures.
Spinlocks

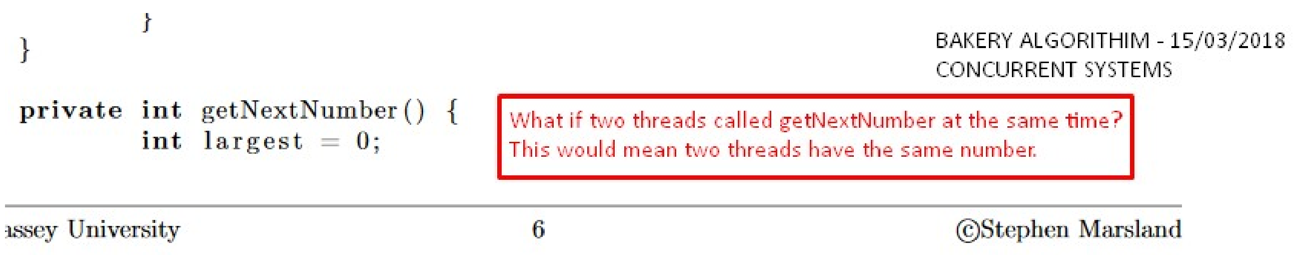
```
global boolean lock
thread A {
   while(lock)
   //--/ Critical Section
}
thread B {
   lock = true;
   //--/ Critical section
   lock = false;
}
```

- A thread may still be executing the critical section before it is locked.
- Busy-waits: the thread stays on the professor in an infinite loop.

In addition to mutual exclusion, assure:

- Each thread must eventually be able to enter the critical section.
- Any thread may halt in any noncritical section.

Bakery Algorithm

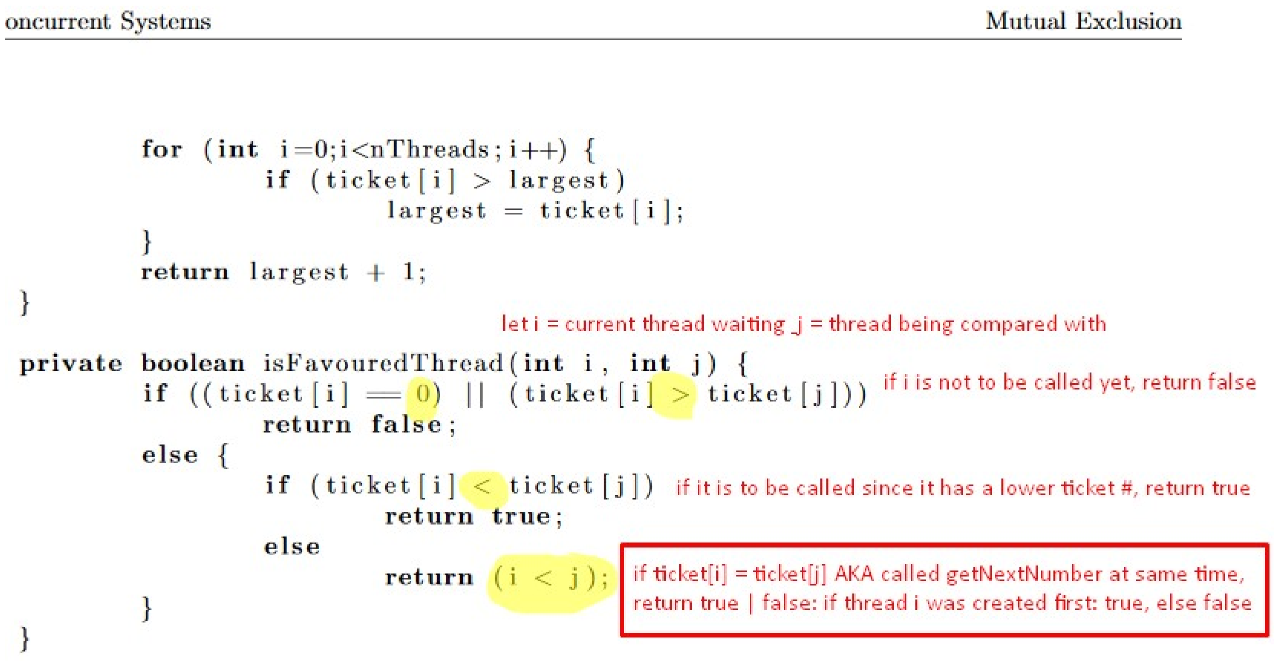


```
        }
}

private int getNextNumber() {
        int largest = 0;
```

BAKERY ALGORITHIM - 15/03/2018
CONCURRENT SYSTEMS

What if two threads called getNextNumber at the same time?
This would mean two threads have the same number.

assey University 6 ©Stephen Marsland



oncurrent Systems Mutual Exclusion

```
        for (int i=0;i<nThreads;i++) {
                if (ticket[i] > largest)
                        largest = ticket[i];
        }
        return largest + 1;
}

private boolean isFavouredThread(int i, int j) {
        if ((ticket[i] == 0) || (ticket[i] > ticket[j]))
                return false;
        else {
                if (ticket[i] < ticket[j])
                        return true;
                else
                        return (i < j);
        }
}
```

let i = current thread waiting j = thread being compared with

if i is not to be called yet, return false

if it is to be called since it has a lower ticket #, return true

if ticket[i] = ticket[j] AKA called getNextNumber at same time,
return true | false: if thread i was created first: true, else false

Additional keywords:
Race condition: When a thread always wins the race to the critical section.
Starvation: When a thread waits for access to a critical section but never gets it.
Deadlock: All threads are blocked waiting for something that'll never happen.
Live-lock: When both threads are running, but are stuck in loops waiting for x condition.

Reentrant Locks

Created: *29/04/2018 2:49 PM*
Updated: *29/04/2018 2:57 PM*

Reentrant Locks

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    //../ Critical section
}
finally {
    lock.unlock();
}
```

For **fair (FIFO: first in first out)**, that is: give threads access in the order that they request it,

```
new ReetrantLock(true);
```

Fairness rarely required, since JVM guarantees that threads will not be starved.

Reentrant Read/Write Locks
Allows multiple simultaneous readers. However, writing must be mutually exclusive.

ReentrantReadWriteLock has two locks:
readLock: used to avoid anyone writing.
writeLock: prevents any other access.

Synchronization

Created: *29/04/2018 2:07 PM*
Updated: *29/04/2018 3:10 PM*

Synchronization
Is a mutex (mutually exclusive) lock. It permits only one thread in its critical section. Focuses on the *object*, rather than the thread itself.
Synchronization solves the following issues:

- **Thread interference:** when there are two operations running in different threads but acting on the same data, interleave. For example, two threads write to the variable *a* at the same time: they both have their own version of *a*, and only one wins. Likewise with reading, may read at same time of read: but what'll the variable contain at that time?

- **Memory consistency:** No guarantee of variables between threads, of the actual value.

Happens-Before Relationship: when a thread exits a synchronized block, it updates the modified variables so all threads have the same version.

Two methods of implementing synchronization:
Method level:

```
synchronized void function() {
    //../ critical section
}
```

Block level:

```
synchronized(this) { //this: the object that should be locked
    //../ critical section
}
```

A thread, when calling a synchronized block:

- if lock is free, go
- otherwise go to sleep & awaken when lock is free

Implicit (synchronized) -

- cannot interrupt threads waiting
- can't check if free before requesting access
- can't stop waiting
- they're not fair

Refer to Lab Three

Explicit Lock VS Implicit Lock

Created: *29/04/2018 3:00 PM*
Updated: *29/04/2018 3:07 PM*

Which lock is ideal?

- Use synchronous if limitations don't matter since it's easy to use.

- If need more functionality, use reentrant locks such as having simultaneous readers, or checking if a lock is in lock state before accessing it.

Ultimate goal: Avoid critical sections wherever possible otherwise, keep size minimal.

Volatile

- These variables are placed into main memory. This makes writing to the variable *atomic*.
- However, there may still be memory consistency problems where threads try to access the same variable at the same time when writing: only one will win access to the atomic operation.
- Implement compare & swap (see Atomic Classes for more) to fix this.

Semaphores

Created: *17/04/2018 7:26 PM*
Updated: *20/04/2018 6:45 PM*

Semaphores

- Semaphores were founded by Edsger W. Dijkstra.
- Two methods: acquire() and release()
- A semaphore is basically an atomic integer.
- Acquire() checks if the integer is greater than zero. If it is, it's decreased by one and the thread is allowed in the critical section.
- Release() checks if there are any suspended process waiting, and if there is it wakes only one up. Otherwise, increases the semaphore.
- Generic (or counting) semaphores allow multiple threads into the critical section. Often good for controlling access to a limited amount of resources: such as items in a buffer being added/removed.

Refer to BarberShop for implementation (src folder).

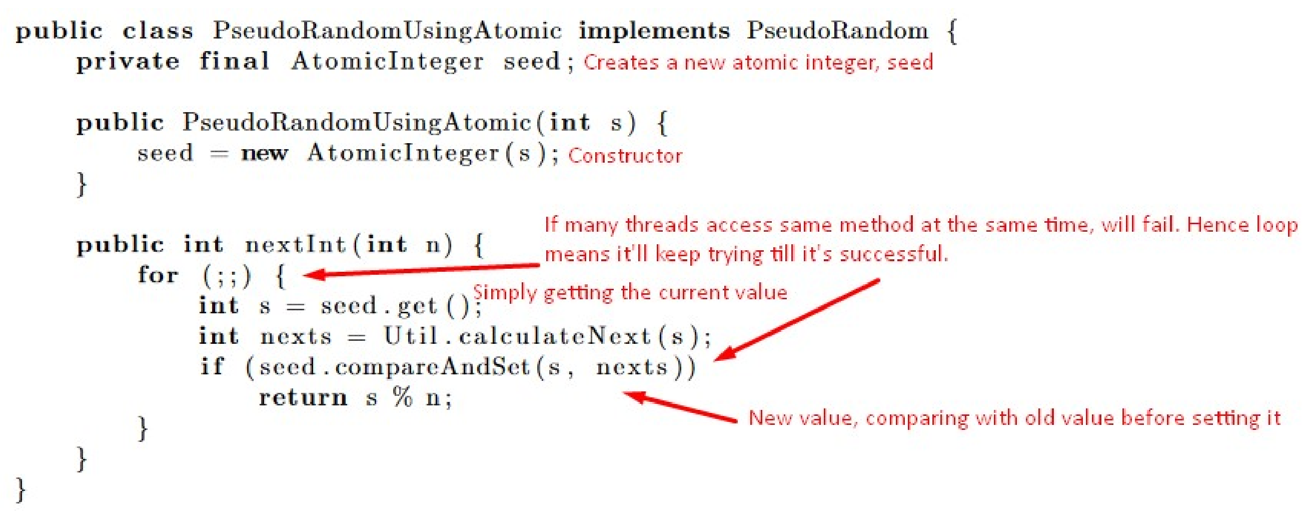
Atomic Classes

Created: *17/04/2018 7:34 PM*
Updated: *20/04/2018 6:45 PM*

Atomic Classes

Compare & Swap
Is a machine instruction which means it's atomic (cannot be interrupted). It's used to check if a variable's old value is still the same before assigning a new value: checking if another thread is accessing it.

java.util.concurrent.atomic
Provides the atomic classes. They have methods: get() & set() which is equivalent to volatile variables.



```
public class PseudoRandomUsingAtomic implements PseudoRandom {
    private final AtomicInteger seed; Creates a new atomic integer, seed

    public PseudoRandomUsingAtomic(int s) {
        seed = new AtomicInteger(s); Constructor
    }

    public int nextInt(int n) {
        for (;;) {
            int s = seed.get();Simply getting the current value
            int nexts = Util.calculateNext(s);
            if (seed.compareAndSet(s, nexts))
                return s % n;
        }
    }
}
```

If many threads access same method at the same time, will fail. Hence loop means it'll keep trying till it's successful.

New value, comparing with old value before setting it

Monitors

Created: *19/04/2018 6:48 PM*

Updated: *2/05/2018 12:24 PM*

Monitors

- Monitors use a 'wait and notify' method to communicate between other threads.
- That is, wait() and notify() methods corresponding to sleep and wake up.
- wait() is a must, sleep() does not enforce locks. Wait() is implemented with locking features.
- A monitor '*is essentially a **shared class** with **explicit queues***'.
- **A shared class** refers to a toolkit accessible by multiple threads, while all methods are synchronized but attributes remain encapsulated.
- Using a monitor to protect a class (a data structure?) and assure it is mutex. *Java has implemented this in the root Object class, so all objects have it.*

Methods in Java.lang.Object
notify() - wakes up a single chosen thread waiting on this object's monitor.
notifyAll() - wakes up all threads waiting on this object's monitors.
Threads awakened in the above method will not be able to proceed until the current thread holding the object's monitor relinquishes the lock on the object.
wait() - waits until it has been notified.

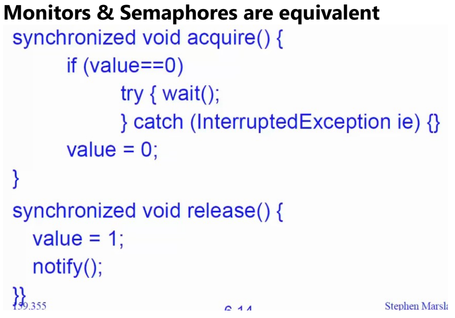
wait() should be guarded, in the event a thread is awakened when it shouldn't be. For example,

```
synchronized(obj) {
    while(<condition does not hold>)
        obj.wait();
}
```

This makes the thread currently accessing the object, wait until some condition is met.

Refer to ProducerProblem in monitors package.

Monitors & Semaphores are equivalent



```
synchronized void acquire() {
        if (value==0)
                try { wait();
                } catch (InterruptedException ie) {}
        value = 0;
}
synchronized void release() {
    value = 1;
    notify();
}}
```

159.355 6.14 Stephen Marsh

Thread Pools

Created: *2/05/2018 12:31 PM*
Updated: *3/05/2018 1:10 PM*

Thread Pools

1. Threads are created.
2. Threads wait until they are given tasks.
3. Once finished, they return to the thread pool.

- Pools can be created using *ThreadPoolExecutor*
- *LinkedBlockingQueue* is used to store any tasks that are waiting to run.
- The *ThreadPoolExecutor.execute()* method is used to call a run() method of a thread implementation.
- Thread pools are terminated using *ThreadPoolExecutor.shutdown()*. All tasks will be completed, but no new tasks will be accepted. *shutdownNow*() will attempt to stop all current tasks as well. Returns list of tasks waiting for execution.
- The pool shrinks to its specified minimum if threads go unused.

Different types of blocking queues
There are different types of blocking queues (which store the awaiting tasks).

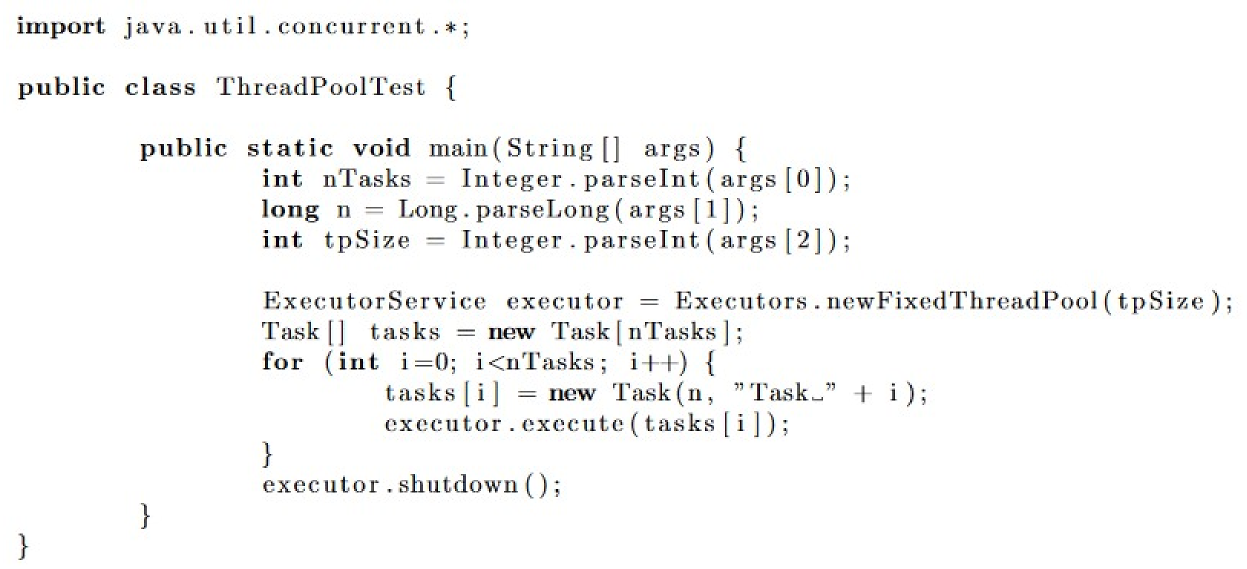
- *ArrayBlockingQueue* (FIFO, bounded queue)
- *LinkedBlockingQueue*: linked list (FIFO, unbounded queue)
- *PriorityBlockingQueue*: orders tasks by priority
- *SynchronousQueue*: doesn't maintain a list, but forwards directly to a thread.

Thread Factory
Creates threads on demand.

The Rejected Execution Handler
Decides how to deal with tasks that are rejected by the execute() method.

- *AbortPolicy*: throws a *RejectedExecutionException*
- *CallerRunsPolicy*: executes the new task outside the thread pool.
- *DiscordPolicy* discards the task.
- *DiscardOldestPolicy*: discards the oldest task in the queue.

Executors
There are classes which makes the job easier. *newCachedThreadPool*() - reuses cached threads, *newFixedThreadPool()* - reuses a fixed number of threads. Also, *newScheduledThreadPool*().



```
import java.util.concurrent.*;

public class ThreadPoolTest {

        public static void main(String[] args) {
                int nTasks = Integer.parseInt(args[0]);
                long n = Long.parseLong(args[1]);
                int tpSize = Integer.parseInt(args[2]);

                ExecutorService executor = Executors.newFixedThreadPool(tpSize);
                Task[] tasks = new Task[nTasks];
                for (int i=0; i<nTasks; i++) {
                        tasks[i] = new Task(n, "Task␣" + i);
                        executor.execute(tasks[i]);
                }
                executor.shutdown();
        }
}
```

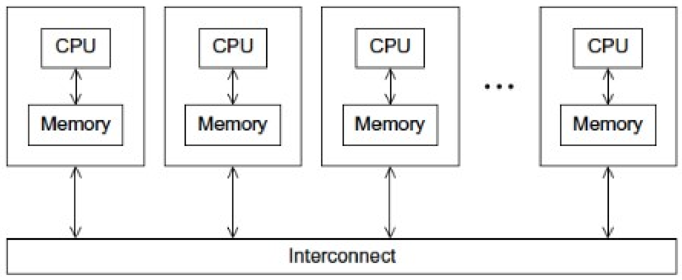
Message Passing Interface (MPI)

Created: *8/05/2018 1:17 PM*
Updated: *14/05/2018 5:50 PM*

Message Passing

Shared memory space is fine, until you have processes or threads across the network. This requires some sort of communication protocol.

Such system is called **Distributed Memory**. Since each thread or process has its own memory, there is no worry regarding race conditions.



Two methods for communication

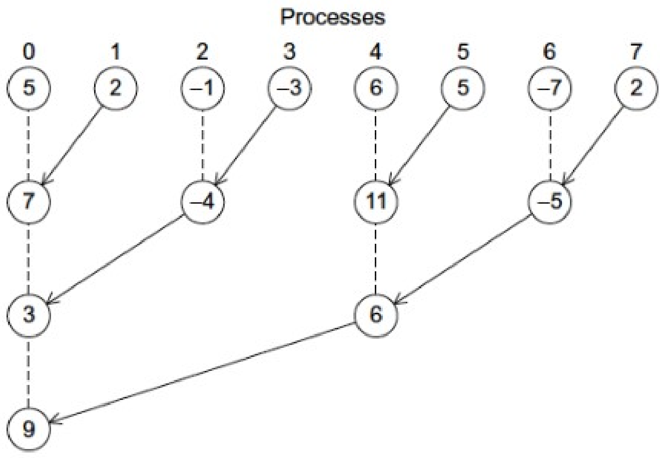
- *Synchronous*: threads sending & receiving messages are blocked until the message is sent.
- *Asynchronous:* threads will not block.
- *Rendezvous*: implements Request-Reply. Asynchronous receiving and synchronous reply. So a thread waits until the other thread replies.

MPI in Java using MPJ

MPJ offers the following methods:

1. *Send*: blocks while sending the message to the process.
2. *Recv*: blocks until it receives.
3. *Isend*: non-blocking send.
4. *Irecv*: non-blocking receiving.
5. *Bcast*: sends a global message.
6. MPI.Wtime() returns the current time stamp.

Collective Communication



Processes can combine messages to create a structure. **The load is furthermore balanced** across processes speeding up the process since the system is using parallelism.

There are functions such as *MPI.SUM* used to combine data from many processes. *MPI.Allreduce* transmits messages to all processes. *MPI.reduce* passes the content onto a single process.

Safety
Deadlock is possible. If unsure if process can deadlock, use *synchronized*. *MPI.Ssend* enforces *synchronized*.

Message Passing Java (MPJ) ...more

Created: *22/05/2018 2:59 PM*
Updated: *22/05/2018 6:02 PM*

Some more on Message Passing
The rank is the current process' ID. Each process has its own ID which uniquely identifies it. The size is the total number of processes specified when running with ***-np {NUMBER OF PROCESSES}*** as a parameter.

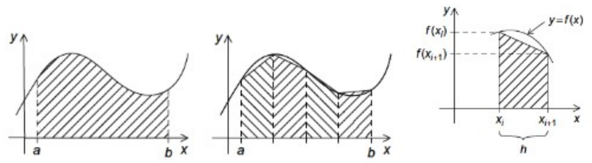
```
import mpi.MPI;

public class HelloWorld {
                public static void main(String[] args) throws Exception {

                        MPI.Init(args) ;

                        int rank = MPI.COMM_WORLD.Rank();
                        int size = MPI.COMM_WORLD.Size();
                        System.out.println(”I␣am␣process␣<”+rank+”>␣of␣total
␣␣␣␣␣␣␣␣ ␣␣␣␣␣␣␣␣ ␣␣␣␣␣␣␣␣ ␣␣␣␣␣␣␣␣<”+ size+”>␣processes.”);
                        MPI.Finalize();
                }
}
```

The MPI.COMM_WORLD class is the communicator which handles the parallelism behind the communication between the processes. This class contains the methods *Send, Recv, Isend, Irecv, Bcast, Reduce, Gather, Scatter, and Barrier. I at beginning is non-blocking.*

Example usage in terms of scientific programming
The trapezoidal rule for example approximates the area underneath a curve. The number, n, is how many times the area underneath the curve is divided. This is also the number of processes for example, that will calculate their own individual areas.

$$\text{Area of 1 trapezoid} = \frac{h}{2}(f(x_i) + f(x_{i+1}))$$

$$h = \frac{b-a}{n}$$



1. partition the solution into tasks.
    1. find the area of an individual trapezoid.
    2. sum the areas.
2. identify the communication channels between the tasks: pass the final areas to the process which sums the lot.
3. aggregate the tasks into composite tasks.
    1. how many processes are needed? n
    2. split the range (b - a) by n.

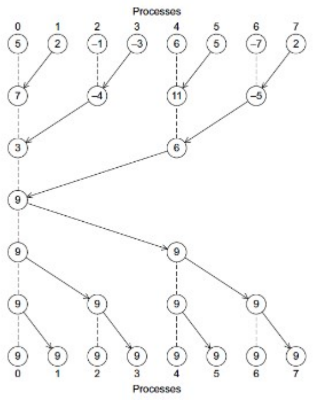
Each process will run the same implemented function. Therefore, the rank can be used to determine the value of x for performing the calculation. While rank == 0, is the root process summing, the other process send to the root process. Note MPI.DOUBLE is simply the data-type declaration.

```
if (rank != 0) {
MPI.COMM_WORLD.Send(local_int,0,1,MPI.DOUBLE,0,99);
} else {
total_int = local_int[0];
for (source = 1; source < size; source++) {
MPI.COMM_WORLD.Recv(local_int,0,1,MPI.DOUBLE,source,99
total_int += local_int[0];
}
}
```

WARNING (Deadlock)
Deadlock can occur with the BLOCKING version of COMM_WORLD.Send, so use ISend for non-blocking. There are also **synchronized** versions:

MPI.COMM_WORLD.Ssend

Efficiency (Scatter, Gather, Collective Communication, Reduce, Allreduce)
In these example, process rank = 0 deals with all the messages. However, this isn't exactly efficient. This is when *Collective Communication* plays a role. It's a tree-based structure, where messages are passed along as they are finished. The load is somewhat more balanced between processes. This implies that there is more work done in parallel, which makes the program more efficient.

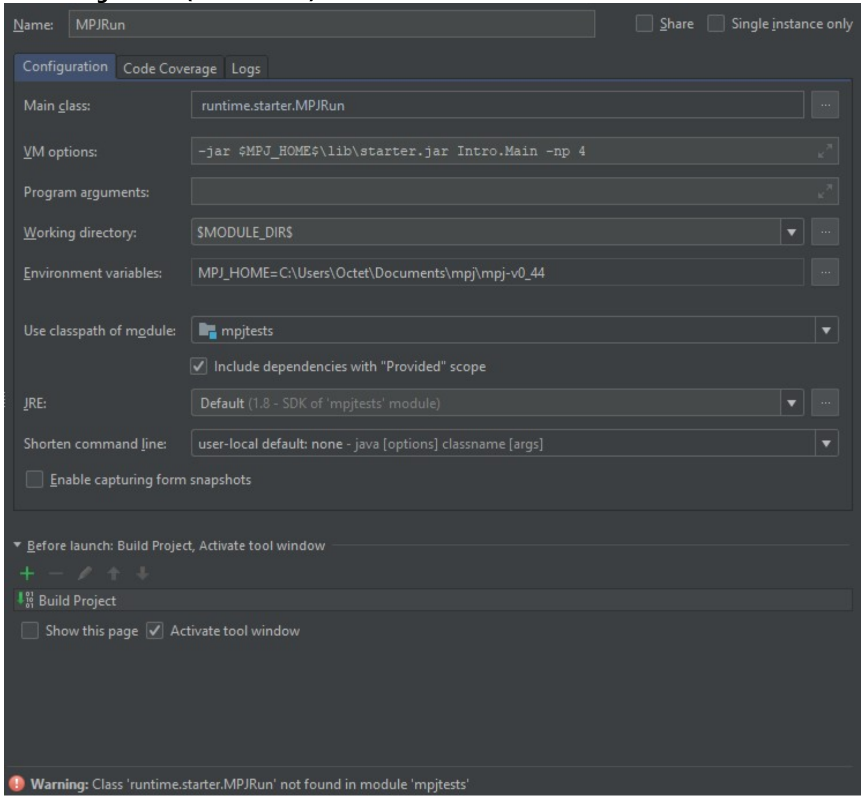


There is a method in MPI.COMM_WORLD, Reduce (the following is for C).

```
//Function prototype
MPI_Reduce(
    void* send_data,     //An array of data to send
    void* recv_data,     //Needed for Root process, to recv data. size: sizeof(datatype) * count
    int count,           //Total # of processes
    MPI_Datatype datatype,    //The data type of the data (i.e MPI.DOUBLE)
    MPI_Op op,            //The operation to perform, i.e MPI.SUM
    int root,            //The Rank of the root process
    MPI_Comm communicator
)
```

Refer to ./MPI/mpj_reduce.pdf
MPI.COMM_WORLD.Allreduce transmits to all processes, rather than just root.
MPI.COMM_WORLD.Scatter and MPI.COMM_WORLD.Gather scatter and gather data arrays from multiple processes. Scatter to give, gather to collect the results.

Run configuration (IntelliJ IDEA)



Name: MPJRun
Main class: runtime.starter.MPJRun
VM options: -jar MPJ_HOME\lib\starter.jar Intro.Main -np 4
Program arguments:
Working directory: $MODULE_DIR$
Environment variables: MPJ_HOME=C:\Users\Octet\Documents\mpj\mpj-v0_44
Use classpath of module: mpjtests
[✓] Include dependencies with "Provided" scope
JRE: Default (1.8 - SDK of 'mpjtests' module)
Shorten command line: user-local default: none - java [options] classname [args]
[ ] Enable capturing form snapshots
Build Project
[ ] Show this page [✓] Activate tool window

Warning: Class 'runtime.starter.MPJRun' not found in module 'mpjtests'

Map-Reduce

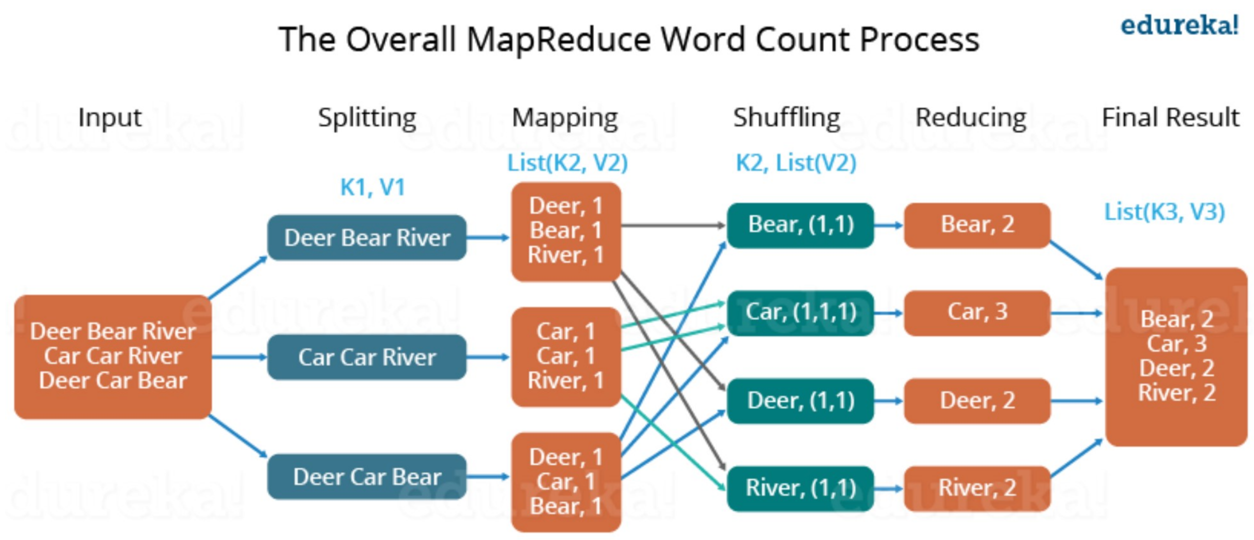
Created: *22/05/2018 6:09 PM*
Updated: *31/05/2018 10:41 AM*

Map

- Extract something you care about from each record.
- Shuffle and sort.

Reduce

- Aggregate, summarize, filter or transform.
- Write the results.



For example, this can be used to calculate the frequency of words in some array (shown above).

MapReduce can be scaled to clusters of servers to speed up the computation, with easy configuration.

MPI v MapReduce v OpenCL

Created: *30/05/2018 1:18 PM*
Updated: *30/05/2018 1:18 PM*

OpenCL

Created: *31/05/2018 10:43 AM*
Updated: *31/05/2018 11:52 PM*

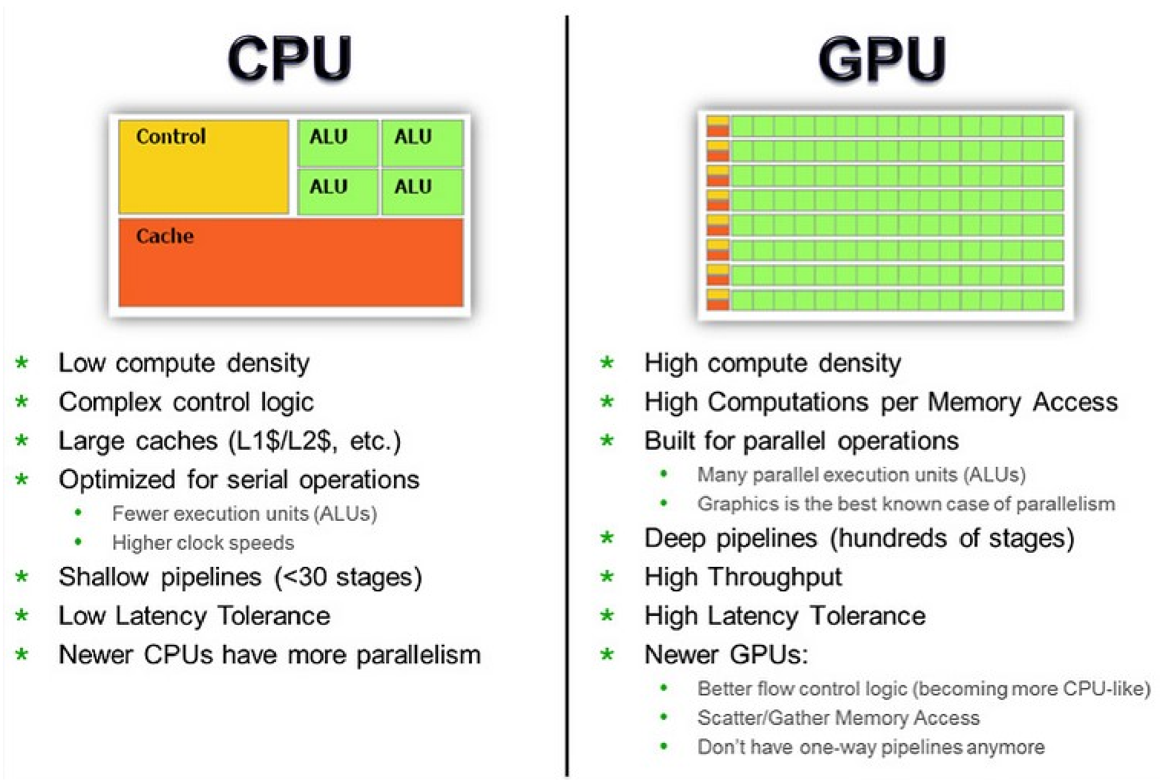
Open Computing Language
Introduction

- It is a framework that allows you to write programs that execute across heterogeneous platforms.
- It provides a standard interface for parallel computing using task-based & data-based parallelism.
- OpenCL can communicate with a large range of devices.

Let the Host be the desktop system.
Let the Compute Device be: CPU or GPU

Use cases -- when to use GPU for computation

- When devices move memory faster than host.
- Changing from one data format to another.
- Devices calculator faster than the Host, big chunks of data. That is, more computation, using the ALU.



CPU

- Low compute density
- Complex control logic
- Large caches (L1$/L2$, etc.)
- Optimized for serial operations
  - Fewer execution units (ALUs)
  - Higher clock speeds
- Shallow pipelines (<30 stages)
- Low Latency Tolerance
- Newer CPUs have more parallelism

GPU

- High compute density
- High Computations per Memory Access
- Built for parallel operations
  - Many parallel execution units (ALUs)
  - Graphics is the best known case of parallelism
- Deep pipelines (hundreds of stages)
- High Throughput
- High Latency Tolerance
- Newer GPUs:
  - Better flow control logic (becoming more CPU-like)
  - Scatter/Gather Memory Access
  - Don't have one-way pipelines anymore

Double precision maths is slower on a GPU, by *2x*. It's because it requires more data, it doubles the amount of required data.

WHEN TO USE GPU
GPUs have less computation power per thread. While it has more threads it is able to provide better parallelism. It has on-board memory making it ideal f or large data sets. GPUs are often quick, since they do render screens in real time, just like how one would speak through the microphone, and someone over the network would almost receive that in real time. CPUs are more ideal for lower amount of threads, or for tasks which are computationally heavy.

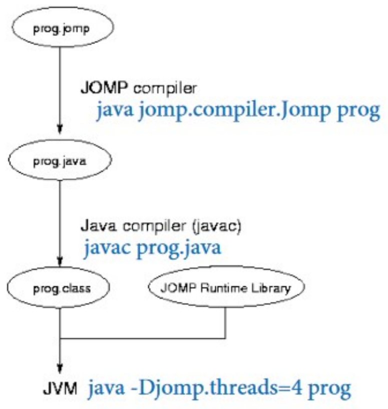
OpenMP

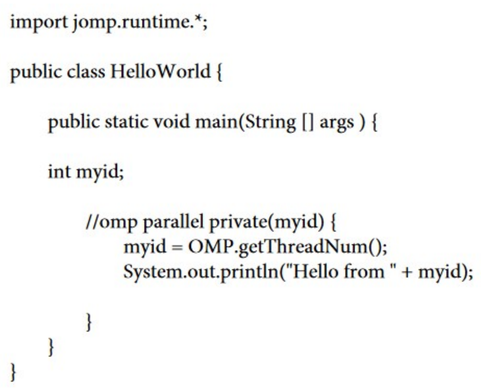
Created: *31/05/2018 11:12 AM*
Updated: *31/05/2018 11:59 AM*

Open Multi Processing

A master thread can spawn a team of threads. These teams form clutters called parallel regions. In Java, there is a port for OpenMP called Jomp (since OpenMP is designed for C/C++ & Fortran).

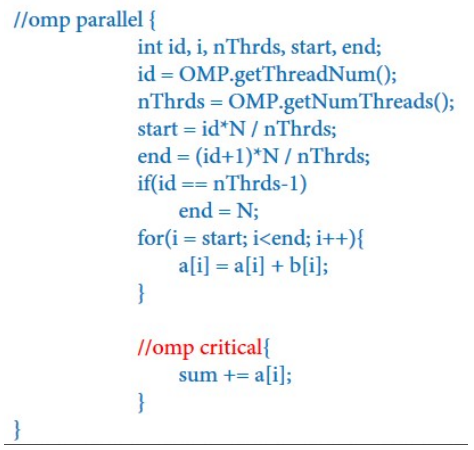
Rather than using #pragma, like in C to tell the compiler to do something... Java & Fortran uses the keyword, *omp*. Jomp has it's own compiler, which then after being compiled, compiles down to Java.





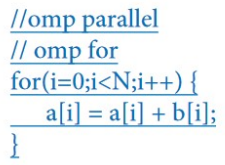
```
import jomp.runtime.*;

public class HelloWorld {

    public static void main(String [] args ) {

    int myid;

        //omp parallel private(myid) {
            myid = OMP.getThreadNum();
            System.out.println("Hello from " + myid);

        }
    }
}
```

//omp parallel, defines a portion of code that should be executed in parallel.
While, ***//omp barrier***, each thread waits at the barrier until all the threads arrive.
For synchronization & critical sections, ***//omp critical***.



```
//omp parallel {
        int id, i, nThrds, start, end;
        id = OMP.getThreadNum();
        nThrds = OMP.getNumThreads();
        start = id*N / nThrds;
        end = (id+1)*N / nThrds;
        if(id == nThrds-1)
            end = N;
        for(i = start; i<end; i++){
            a[i] = a[i] + b[i];
        }

        //omp critical{
            sum += a[i];
        }
}
```

Parallel Loops
OpenMP allows for *for loops* to be executed in parallel. Only works for for loops, and only if the number of iterations is known. At the end of parallel constructs, there is an implicit barrier.



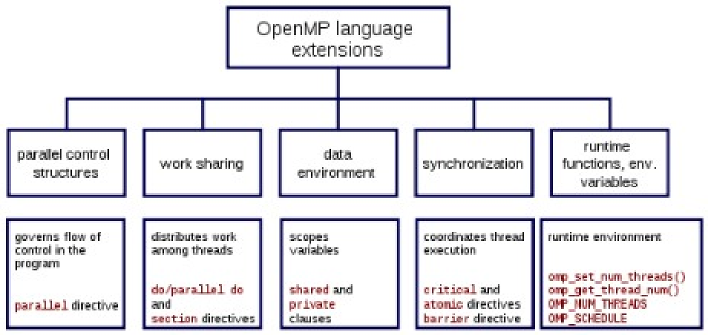
```
//omp parallel
// omp for
for(i=0;i<N;i++) {
    a[i] = a[i] + b[i];
}
```

A race condition exists in the following example, since the iterations are performed in a range of threads, local variables differ between threads inside the loop.

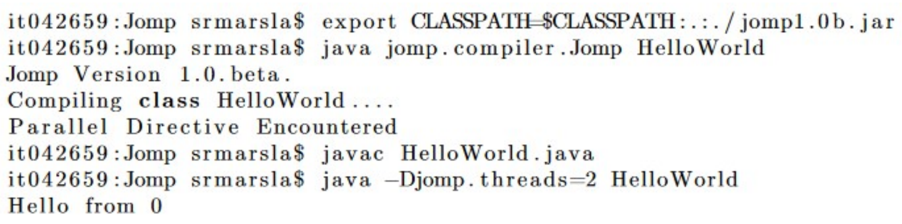
```
long sum = 0;
    //omp parallel for
        for (int i=1; i<1000000; i++)
            for (int j=1; j<100; j++)
                sum += i % j;
```

Therefore, a reduction variable must be used: basically declares a global variable that is shared across threads. Results from each thread are combined. A reduction variable can be declared using ***//omp parallel reduction(+: varname)*** where + is the operation being performed on the variable.

```
//omp parallel for reduction(+:sum)
        for (int i=1; i<1000000; i++)
            for (int j=1; j<100; j++)
                sum += i % j;
```


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Compiling & Runtime
To compile, must use jomp.compiler.Jomp someFile (with .jomp extension). This will produce a Java file, which can then be compiled. Javac someFile.java. Then running, the number of threads must be specified, java -Djomp.threads=n someFile. jomp1.Ob.jar must be in the CLASSPATH.


```
it042659:Jomp srmarsla$ export CLASSPATH=$CLASSPATH:.:./jomp1.0b.jar
it042659:Jomp srmarsla$ java jomp.compiler.Jomp HelloWorld
Jomp Version 1.0.beta.
Compiling class HelloWorld....
Parallel Directive Encountered
it042659:Jomp srmarsla$ javac HelloWorld.java
it042659:Jomp srmarsla$ java -Djomp.threads=2 HelloWorld
Hello from 0
```

Notes
In the notes, it uses an array, result[] to fetch the results. The reason it cannot return a result, is because it's being performed in parallel. Likewise, if it was a simple integer, it would not correspond to the same memory address. Therefore, an array is used. ***//omp sections { ... }*** followed by ***//omp section*** for specifying what should go on in the first & second threads, is useful.

Assignment

Created: *31/05/2018 11:34 PM*
Updated: *9/06/2018 12:44 PM*

What will need to be done initially?

1. Load entire Iris data set (*training* set).
2. Define y, := test instance.
3. Define k, the number of neighbors (or threads).

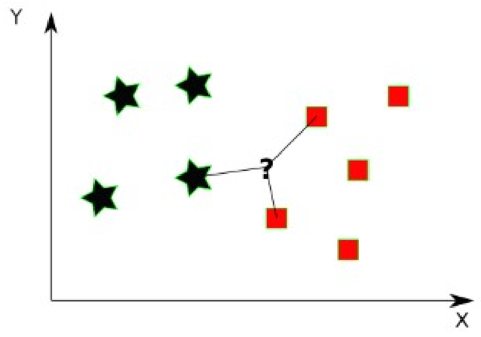
What will need to be done, in parallel?

1. Calculate distance between test & training vectors.
2. Update list of k closest neighbors.

Barrier

1. At the end, the class with majority of k neighbors is selected/output.

In other words, finding the closest k neighbors, then the test data point will belong to the class which has the most neighbors.



So, the class with the most neighbors is the 🟥 (2 v 1). **?** then belongs to 🟥

1. Let k, be the number of processes, but also the number of closest neighbors to compute.
2. Each thread finds the closest neighbor for... problem: there may be a second point in another data set closer than a point in another data set.

DESIRE: Minimize communication, MPI is the choice since for larger data sets, they require more memory while there isn't as much computation being carried out. This enables the ability to use many nodes to have their individual memory (i.e servers), rather than having it on a single node. This makes the system expandable.

There is a testing set consisting of fifteen vectors.
Now, there is a training set which contains all the data minus the testing set. This will be 135 vectors.