1. (a) Both message passing and data sharing are mechanisms that allow parallelised programs to exchange information at runtime in order to dynamically coordinate their behaviour.

   Data sharing requires a shared address space between the communicating execution streams (usually threads). Both threads will have access to a common space in memory, and will usually communicate through it with the judicious use of locks, barriers and other synchronisation constructs. Data sharing is suited to applications running on a single machine with multiple processors or multiple cores.

   Message passing does not require a shared address space, but instead the parallel portions of the program must be able to communicate by sending and receiving messages. This avoids the need for synchronisation constructs, and is often more suited to programs that are executing on multiple seperate machines.

   Data sharing is often a more convenient form of communication between parallel portions of a program since communication is usually reliable and fast, while message passing often occurs through some sub-system which can vary wildly in its reliablilty, latency characteristics and bandwidth. For example, message passing could be done over the internet, which could lead to a latency of several hundred milliseconds and a bandwidth of just a few megabits per second, while writes to a shared portion of memory would have a latency of nanoseconds and have a bandwdth of multiple gigabits.

   An example of a data-sharing implementation is OpenMP, and an example of a message passing implementation is MPI.

   (b) Data-sharing, shared memory; Data sharing maps very easily onto a share memory architecture, since all threads have access to the same address space by default. All that is required is for some scheme to by created in order to control the semantics of each thread's accesses to shared memory. One approach could be to designate some memory areas as private for each thread (for local calculations etc that don't require communication or synchronisation), but then use synchronisation control structures such as locks, barriers, atomic operations etc to coordinate access to the shared memory locations. Performance should scale relative to the amount of synchronisation that is required by the parallel threads, though the overhead of data transfer and synchronisation will be very low since all memory operations are fast.

   Data-sharing, distributed memory; Since by default, each core can only access the memory that is attached to that core, a mechanism where cores can request values from another core's memory will be required to give the illusion of a shared address space. Managing the physical locations of values in the shared address will be a target for optimisation, since if one thread is continually using data that is on another core, then this will incur a large overhead. If the commu-

nication between threads is minimal, then this approach could give the hardware designer more flexibility to do things like improve the clock speed (due to more simple circuitary) or have a larger CPU cache relative to the memory size (if the size of each CPU cache was the same, but the memory accessible by each CPU was $\frac{1}{n}$, then more cache hits would be achieved). However, the program must be very architecture aware here, and avoid constantly fetching and writing to locations in the memory of another core.

Message-passing, shared memory; Here, we can implement a layer of abstraction that stops each thread from writing to memory outside its own address space, but provide a mechanism whereby threads can 'send a message' which is then written to a region of the shared memory space to be 'recieved' (i.e. read) by another thread. This is a less efficient use of shared memory than with data sharing, since the overhead of the abstraction is introduced, but this overhead can be minimalised. One main advantage of this approach is that the programmer does not have to consider many synchronisation operations such as locks or atomic operations (unless perhaps accessing some shared resource) because the abstraction layer would handle this for the programmer.

Message-passing, distributed memory; Message passing is the natural form of communication for distributed memory systems. There would be little abstraction between the programmer and the underlying memory model, which could lead to efficient implementations of programs. Memory accesses to locations owned by other threads would likely be slower than with the shared memory message passing approach, but accesses to memory on the same core as the thread is running would likely be more efficient since there is no software monitoring the memory acesses.

The performance of all the message-passing approaches will also depend on the topological arrangement of the cores. A fully connected network would give the best performance (at perhaps an increased cost in terms of wiring, which may not be feasible since the wires would grow order of n^2). Other interesting topologies could be a torus/grid which balance connectivity with wiring complexity or a circular network or centeral bus, which are the most simple approaches but where communication can take the longest.

(c) i. There are two main approaches to loop scheduling; static and dynamic.

When the loop is scheduled statically, the number of threads used and the iterations assigned to each thread is set at compile time, which reduces the runtime overhead of the threading, but is inflexible if the size of the data is not known in advance.

Since we know how many iterations there are (N) and that the work is constant for each iteration, we can statically assign which

threads get which iterations (split the number of iterations by the number of threads, and assign each thread a consecutive chunk). This will produce optimal performance since the loop overhead is minimal at runtime, and all the threads would be expected to finish at the same time.

ii. Since the work done for each iteration would vary, all of the iterations can be held in a queue and requested by a thread when the thread requires more work.
This means that if one thread was assigned a very long iteration, and another was assigned a short iteration, then the latter thread could be assiged more work from the queue before the former was finished, thereby ensuring that thread utalisation is high.
To decrease the overhead of the queue, threads could take multiple iterations at once from the queue, and since N is large, this would by asymptotically similar in terms of the distribution of work between the threads (assuming the distribution of long and short iterations was random), but would require less interation between the queue and the threads (which could potentntially be a bottleneck).
Of course, dynamic loop scheduling involves a runtime overhead that is not present for statically allocated loops, and is therefore (in a sense) less efficient. On the other hand, an intelligent loop scheduler could potentially adapt to the 'shape' of the workload and assign iterations in such a way to evenly distribute the load of multithreading at runtime in a manner impossible for a statically analysed method.

2. (a) Control structures such as barriers, locks, semaphores and monitors are important because Java (and most other programming languages) let threads access the same address space but these accesses are not coordinated by default. These control structures utalise special instructions that are supported at the hardware level to ensure consistency.

These are implemented in the `java.util.concurrency` package, and include `CyclicBarrier`, atomic variables such as `AtomicInteger`, and locks such as `ReentrantLock`.

To give an example, if two threads were incrementing a variable:

```
LDR R1, x
ADD R1, R1, 1
STR R1, x
```

If two threads entered this section of code at the same time, then it could go:

```
1. LDR R1, x
```

```
1. ADD R1, R1, 1
2. LDR R1, x
1. STR R1, x
2. ADD R1, R1, 1
2. STR R1, x
```

Now the value of `x` would be incremented by one, but not by two. An atomic operation such as that proided by `AtomicInteger` or a lock such as a `ReentrantLock` would be needed to ensure that the shared variable `x` was updated correctly.

(b) The wait operation on the semaphore needs to be atomic. It can be implemented in a naive manner like so:

```
loop LDR R1, semaphore
     CMP R1, 1
     BLT loop
     SUB R1, 1
     STR R1, semaphore
     ...
```

The trouble with this is that if two threads are active in this block at the same time, then they could both exit the wait loop by reading the semaphore when it is 1. This means that two threads would be in the critical section at the same time. The solution is to use an atomic operation, which can test to see if the variable is 1 and update the variable at the same time if it is:

```
loop TAS R1
     BNZ loop
     ...
```

Here, the `TAS` instruction (depending on the implemention) zeros the memory location in R1 if it was 1 (indicating that the semaphore is taken). If the TAS instruction failed then the branch instruction would work and it would be tried again.

(c) CISC style instructions such as Test and Set require lots of work to be done for that single instruction. This does not map well onto modern RISC pipelined processors, since long running instructions result in long pipeline stalls which slash the IPC of the computation.

Furthermore, TAS instructions must lock a memory location, which requires synchronization and bus communication, which is often at a premium on modern multi-core machines. In some cases, the whole snoopy bus must be locked while the instruction executes.

(d) Load linked and store conditional can make the code between them execute as if it was executed atomically. This works by loading a memory address and setting a load linked flag in the processor, and moving the memory address that was loaded into a special register.

If the snoopy bus sees a write that has the same address as the one in the special register, then it unsets the load linked flag in the processor.

The code loops on a load link operation until it sees that the semaphore is free, then it enters the atomic section. When the atomic section is done, the code tries to store the results of the computation, but will not do so if the store conditional command fails (i.e. the load linked flag was not set).

```
loop LDL R1, R2
     CMP R1, #0
     BNC loop
     MOV R1, #1
     STC R1, R2
     CMP R1, #1
     BNE loop
```

(e)
```
wait PUSH R1
     PUSH R2
     MOV  R2, barrier
decr LDL  R1, R2
     SUB  R1, R1, #1
     STC  R1, R2
     CMP  R1, #1
     BNE  decr
loop LDR  R1, R2
     CMP  R1, #0
     BNE  loop
     POP  R2
     POP  R1
```