Sian Lennon CA400
CASE4 16343896

1(a)

HPC or High performing computing is about using large amounts of computing power for short periods of time to do work in a parallel fashion; HPC systems tend to focus on tightly coupled parallel jobs and hence they should work with low latency interconnects. Being tightly each component has a lot of knowledge of every other component and often they are working in tandem. Early HPC systems weren't distributed and often were referred to supercomputers, but with the rise of cloud computing that was shifted and HPC can be distributed and often.
In contrast High Throughput processing/computing is all about how many jobs can be done over a set period of time and is usually measured over a long time. Again in stark contrast components in a HTP system are loosely coupled and therefore each component has, or makes use of, little or no knowledge of the definitions of other separate components, the complete opposite situation of the HPC system.

In regards to Gustafson's law it has more implications for the HPC system, when each component communicates with one another it will create overhead.

HPC systems should leverage MISD(Multiple-instruction, single-data) system architectures if possible. As mentioned the system is designed to do work quicker on a set problem, the systems should be working on the same dataset.

HTP systems should leverage MIMD(Multiple-instruction, multiple-data). The components are often working on their own thing and with separate data there is no overhead with communication into the critical sections

Q1(b)

60 * 1000 = 60000(Minute into milliseconds)
10000 * 100 = 1000000 (Latency in milliseconds)
60000/1000000 = .06
10000 * .06 = 600 requests need to be buffered

Q1(c)

Add more consumers ie more end transaction systems, easiest implementation would be more queues with a load balancer before the queue system to balance the loads between our queues. I would have chosen the above option as it is an effective solution that leverages an already built system, just duplicated and with a load balancer added. Other factors such as an exponential growth in transactions could still be an issue and having just one point of fault on the load balancer could be a major issue down the road.

Q2(a) What are the requirements for deadlock to occur in concurrent code? Which requirement has been the most important in your studies or projects to date? Explain, using an example, why this is the case

1. Mutex - one resource at least should be non-sharable..
2. No pre-emption - No tasks can be interrupted.
3. Hold and wait - When a process holds a resource and requests another.
4. Circular wait (e.g process a is waiting on process b and process b is waiting on process c who is waiting on process a)

The most important I would say has been the hold and wait requirement. Initially, I remember having a few issues within the first assignment with circular wait (robot threads were waiting on the aircraft to be free in a circular motion). But I realised the bigger problem later when my robots were holding an aircraft resource and waiting on the parts resource, this caused issues with starvation and possible deadlock and the easiest fix for me was for the robot to first access the parts, take them and then release the inventory and wait on an aircraft.

Q2(b)

i)
Critical section

```
int balance = this.balance.get();
this.balance.set(balance};
```

ii)

AtomicIntegers are thread safe, that means that they protect the bank balance as thread safe objects behave correctly when accessed by multiple threads.

iii)

The code overall is not thread safe as it is possible for two threads to access the same bank account, and change the balance at the same time as a thread can get the atomic integer in the space between where the other thread getting and setting.

iv)

```
Public class Account {
        private AtomicInteger balance = new AtomicInteger(0);
        public int withdraw (int amount){
                Int val = balance.addandGet(-amount);
        }
}
```

Q2(c)

```
class Transaction {
        Public void transfer (Account from, Account to, int amount) {

                acquire(MyBalanceLockfrom)
                If from.balance >= amount {
                        acquire(MyBalanceLockTo)
                        From.balance = from.balance - amount;
                        To.balance = to.balance + amount;
                }
                release(MyBalanceLockfrom;)
                release(MyBalanceLockTo);


        }
}
```

Q3(a)

Monitors are an Abstract data type that support controlled access to shared resources. A process such as A, B and C must access shared resources through procedures. As below, A and B are queing to get into the CS while C is in there as monitors only allow one process in a critical section at a time. The monitor is encapsulating a segment of code. The condition variables are only accessed from within the monitor. They replace wait/signal from semaphores but have a different way of doing things. The condition variables are used to queue threads if they are waiting on some condition to be true.

Q3(b)

<u>Features</u>

- Semantics of wait() & notify or notifyAll()
- Used on a block of code rather that variables like locks/semaphores

- No code was need to attempt to gain access to the semaphore

My issue with using synchronised in my assignment one was that it only allows for one process to enter the critical section exclusively. This was a problem as I tried to implement them on the parts inventory as I thought only one thread would be able to access this. However, even when the thread was just reading from the parts inventory (and therefore would not cause any overwitting), it was placed in the queue and treated the same as the others.

3(c)

Assumptions by saying Lock & Condition, it is okay to use Reentrant Locks

```
Class BarberShop {
	Lock lock = new ReentrantLock();
	Condition barber_available = lock.newCondition();
	Condition chair_occupied = lock.newCondition();
	Condition door_open = lock.newCondition();
	Int barber, chair, open;

	Public void get_haircut() throws
			InterruptedException {
		lock.lock();
		Try {
			// No barber so wait
			while (barber == 0)
				barber_available.await();

			// barber is yours
			Barber--;
			// Someone in chair
			chair++;

			chair_occupied.signal;

			// Wait for door to open for you
			While (open==0)
				door_open.await();
			}
		Finally {
			// release
			lock.unlock();
```

```
        }
    }

    Public void get_next_customer() throws
            InterruptedException {
        lock.lock();
        Try {
            // Barber available
            Barber++;
            barber_Available.signal;

            // No one to cut hair
            While (chair == 0)
                chair_occupied.await()

            Chair--;

        Finally {
            lock.unlock();
        }
    }

    public void finished_cut () throws
            InterruptedException {
        lock.lock();
        Try {
            // Open door for custy
            open++;
            door_open.signal;

            // Wait for them to leave
            While (open==0)
                customer_left.await();

        Finally {
            lock.unlock();
        }
    }
```

Q5(a)

Examples of commercial: WWW or telecommunications such as Vodafone

Architecture

How the overall system is designed so that the client/server can interact.

The choices available are:

- What kind of interface (is it of generic ie void or type)
- WAN or LAN
- Peer to peer
- TCP/IP other protocols

Complications/issues of DS:

- Considerations of the type o

Design problems:

- The client and server often need to have the same or similar architecture in order to interact.
- Security issues if architecture is not implemented well.

Fault tolerance

Refers to the need to not succumb to faults if they present themselves. How tolerant they are to faults.

Choices:

- Replicate the data on multiple servers.
- Replicate the requests. (at least once in RPC)

Complication/issues of Ds:

- Need to be able to recover if a server goes down or could cause outages.
- Must keep consistency which might be harder to do with replicating servers or requests. E.g if data is updated on one server and not another.

Design problems

- Cost of duplicating servers.
- Duplicate requests need to be handled.

An example of a distributed design choice used in our project was a duplicate server for the chunking of text in the backend. We wanted to allow for fault tolerance but also system performance improvement by decreasing the latency in requests using parallelism. The idea behind it was good and it did work, however the amount of work on the frontend slowed down the process anyway. Going back to Amdahl's law, that makes sense, the program would never be faster than the amount of serial work that has to be done.

Q5(b)

In the UML diagram a RPC method from the client is being called to a remote server in an at-least-once fashion.

The client sends a request as normal that seems to them to be a local method and a timer of expiration is set. This method invokes a client side stub (acts as a client side interface) which then builds the message and sends it to the OS (in the form request(function_id, call_id and params)). Os then sends it to the Server-Dispatcher seen in the diagram. Server-Dispatcher uses the fn_id to find the function from a table and sends it to server with the parameters. After that the server does its work and returns the result to the server dispatcher. The server-dispatcher writes the result to its disk with the id of the call and crashes so the result is not returned to the client.

This is when the timer expires and the client tries again due to the at-least-once semantics. So the client replicates the request exactly and everything happens the same until it gets to the server-dispatcher who has remembered the result of the call. Therefore all it was to do is look up the m_id key and find the result. It then returns to the stub and then to the client.

ii) At-least-once

iii)

<u>Client code</u>

Client:

```
// Set timeout
timeout = random.time();

// Loop to retry the function call after the timeout
Z = retry_after_timeout(fn(x), timeout);
```

Server interface/handler:

```
Fn (x) {
    Rst = Compute x
```

```
            return rst;
        }
```

Server-Dispatcher:

```
        // Save;
        Save(rst)
        // Send on
```