Shared memory parallelism with threads: each processor accesses a single memory system, rather than each addressing their own memory system.

Distributed computing with messages: Is a model in which components located on networked computers communicate and coordinate their actions by passing messages.

Accelerated computing with GPU hardware: The use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications.

The 6 basic MPI commands:
MPI Init() – Allow command arguments to be modified,MPI Finalize(),MPI Comm size() – Number of MPI processes,MPI Comm rank() – Internal process number,MPI Get processor name() – External processor name,MPI Send(), MPI Recv()

Collective communications:
MPI_Bcast is used to broadcast from one process to all processes.
MPI_Scatter is used to break up a large array on one process, and send pieces to all processes.
MPI_Reduce (and MPI_Allreduce) are used when you have one or more variables that exist on every process, and you want to combine them to your head node (or all nodes) using some simple reduction operation.

block scheduling (if memory locality is your primary performance concern) or round robin scheduling (if load balancing is your primary performance concern). Block scheduling can be performed with fairly simple logic applied to rank and size, and round robin loops are easily implemented by replacing a loop that starts at zero and increments by 1 with a loop that starts at rank and implements by size.

Latency: the delay before a transfer of data begins
Bandwidth: The maximum data transfer rate of a network

CUDA thread organization in terms of threads and blocks:
CUDA allows for you to set up groups of threads called blocks, and you can have multiple blocks. You can arrange your blocks in a 2-D grid, and you can arrange your threads in a 3-D block, though you do not have to. You have a variety of options of setting up your threads--one thread per block and lots of blocks, one block with lots of threads, and everywhere in between. The maximum number of total blocks, total threads, and amount per dimension will depend on the card being used, so a proper algorithm should check for these and take it into account.

Thread-based parallelism using the threadIdx, blockDim, blockIdx, and gridDim variables:
The kernel is run with (num blocks)*(num threads per block) instances on the device, and each instance is differentiated by the threadIdx, blockIdx, and gridIdx variables. In this case we are using blocks only and in 1 dimension only. The device memory is then copied back to the host, and the code continues.

Host designated code: use to initialize memory (cannot run on the “device”)

Device code: this code uses an copied of the host code so that it can be run on the device.

Branching itself is basically free or at least very cheap (handled by a separate unit running parallel to ALU for example), but something to keep in mind is that branching also tends to increase GPR pressure, which in turn lowers the occupancy. Low occupancy means there can be less threads in flight at once which influences GPU's ability to hide memory latencies.

Example Problems
1. Consider the two problems. Explain if the algorithms would map well with CUDA enabled GPGPU. If not give a description of how to parallelize on MPI

2. Matrix-Matrix multiply, given 2 matrices A and B, calculate C where $C_{ij} = \sum_k A_{ik} B_{kj}$

A matrix-matrix multiply operation consists of few CPU operations compared to memory reads and writes, and those reads and writes are going to be in a predictable order lending to a high degree of memory locality, particularly on an architecture like a GPU. This would map well to a GPGPU architecture.

b. A Monte Carlo integration library, that provides general routines that operate on developer supplied functions (i.e. the end user/developer writes a function in C that calculates the value of f(x,y,z), and this is passed to the Monte Carlo integration routine as a function pointer.)

A general purpose Monte Carlo integration library that requires the passing of function pointers will be difficult to implement on a GPU. Only the most recent versions of the CUDA standard allow for kernel function pointers, and only the most recent hardware is capable of running the newer standard. GPU acceleration is generally performed at a much lower level in the code (typically at the level of a single loop), and it is difficult to gauge how well a general purpose library would transfer to accelerators. This would be better handled with a distributed code.

2. Consider the following 3 problems. Assuming you are to write a parallel program implemented using MPI, give a brief description of an algorithm to do so, including how you will implement a parallel solution. Focus on 2 issues related to the parallel implementation, the scheduling method to be used, and the collective communication call that would best fit the problem.

a. Histogram creation – you are to read data from the U.S. census and create histograms of demographic data across the entire U.S. population, in this case number of citizens aged 0-9, 10-19, 20-29, and so on.

Given a dataset of population data, you could use the scatter command to split up the data across nodes, and have each node loop through their portion of the data and count up the number of individuals in each bin range, creating an array of populations by bin range. That array could be joined into a single array on the root process using the reduce command with a sum operation.

b. Creation of a fractal landscape – you are developing fractal landscapes for use in a CGI enhanced movie, where mountain backdrops will be based off of the Mandelbrot set, and you need to be able to rapidly compute different regions of the set at high resolution. For a given point in the set , the Mandelbrot set iterates the function , $x_{i+1} = x_i^2 - y_i^2 + x_0$, $y_{i+1} = 2x_i y_i + y_0$ until the point either “escapes” or is deemed to be a member of the set. The value of the set for escapees is taken to be the number of iterations required to escape.

This is an “embarrassingly parallel” program. Each pixel is completely independent of each other pixel, and as such the problem can be split in whatever way is most convenient. Load balancing will be an issue, so a round-robin breakup should be considered. The data afterwards could then be gathered into a single array.

C. Calculation of the value of pi using a Monte Carlo integration, in which multiple random x-y pairs in a box of side length 1 are calculated, and the number of “hits” (those within a distance of 1 from the lower left corner) is recorded. Pi is approximately the ratio of hits to total attempts times 4.

This is an “embarrassingly parallel” problem. The trails can be computed and added in any order. The end result is a single number giving the number of hits across all processes. This can be combined across processes by using the MPI_Reduce command. Load balancing will not be an issue for this problem.

3. In the following code, will deadlock not occur, definitely occur, or possibly occur. Explain fully. Assume size=2;

```
Other = (rank+1)%2;
if(rank==0) {
MPI_Send(buffer_out_messagelength,MPI_INT,other,999,MPI_COMM_WORLD);
MPI_Recv((buffer_out_messagelength,MPI_INT,other,999,MPI_COMM_WORLD, &status);}
else {
MPI_Recv((buffer_out_messagelength,MPI_INT,other,999,MPI_COMM_WORLD, &status);}
```

MPI_Send(buffer_out_messagelength,MPI_INT,other,999,MPI_COMM_WORLD); }

Logic is in place for this code that will ensure that there is a paired send and receive. Rank 0's send command will occur at the same time as rank 1's receive command. As these commands are completed, rank 0's receive command will occur at the same time as rank 1's send command. No deadlock will occur.

4. Correct the following CUDA kernel so that it is tolerant in the event that the number of blocks and threads created do not match the size of the array being computed.

```
_global_add_arrays(int * np, float * a, float * b, float * c) {
Int n=np[0];
Int ind = threadIdx.x+blockDim.x*blockIdx.x;

while(ind<n) {
c[ind] = a[ind] + b[ind];
Ind += blockDim.x*gridDim.x;
} }
```