S1

i. How OpenMP, MPI, MapReduce and GPU help in parallel and distributed computing? Explain each.
OpenMP helps parallel programming to create CPU threads using pragmas without remembering threading APIs. MPI make distributed programming paradigm possible by a multi-process model where they can make use of a cluster. MapReduce provides a scatter/gather programming paradigm with predefined workflows to process data stored on a large distributed file system. Programmer only have to specify map and reduce methods. GPU with ~1000 of core and threads scheduling hardware facilitate hybrid CPU/GPU programming paradigm.

ii. How special locality speed-up processing in shared memory systems?
Data caches pre-fetch successive data items in cache lines for each data access. This means depending upon cache line size, many data accesses can be served from cache which is many fold faster than RAM and thus the speedup.

iii. How task and data decompositions are different. Explain.
Data decomposition allows distribution of data on different cores or nodes for processing. Task decomposition identify dependent and independent computing tasks in one big computation.

iv. How mapping different tasks on same processors helps speed-up execution?
Dependent task need intermediate/partial results of previous tasks and if mapped on the same processor/core many memory accesses can be avoided.

v. Is dynamic scheduling possible in Data-Parallel and Master-Slave algorithms models? Explain for OpenMP, GPU programming, MPI and MapReduce
Data-Parallel Model: Same computation is performed on different parts of data, therefore, they are limited in the number of processors (cores) and amount of memory present.
Master-Slave: Master process assigns tasks to different slave processes at runtime and can use some criterial to do dynamic assignment.

vi. List different steps while designing a nontrivial parallel algorithm.
i) identify parallel computation, ii) mapping to processes/cores, iii) how intermediate data are distributed among various tasks, and iv) avoiding race conditions on data.

vii. How message passing on a single bus slower than when cross-bar switch is used? Assume different processors are connect to memory using a bus and cross-bar switch.
Busses allows point to point communication (one sender and one receiver). Each processors access memory in non-overlapping turns thus increasing time of access. Cross-bar switches allows many non-conflicting path between processors and memory depending about their construction. The time taken on a bus is reduce by the amount of parallel connections.

viii. How a 2-way superscalar pipeline remains idle in a processor?
Vertical waste is introduced when the processor issues no instructions in a cycle, horizontal waste when not all issue slots can be filled in a cycle.

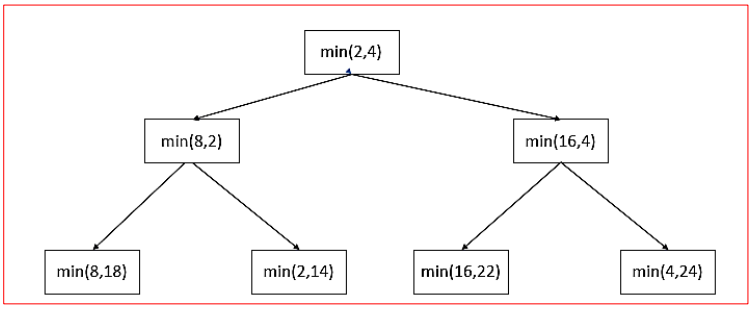
ix. Explain how you can measure and classify computation to communication ratio in your program?
By measuring time taken for computation and communication. Fine grain in case of communication is done more often and course-grain otherwise.

x. How Flynn's classifies parallel and distributed computing? Explain each classification for shared and distributed systems.
i) SIMD: Multi-threading using CPUs and GPUs in shared memory, Master-Slave based in MPI and MapReduce (distributed memory). ii) MIMD: Multi-cores connected to share memory or Multi-Processors connected together using a network. iii) SISD is shared memory but serial, and iv) MISD is not common and is used in special situation where multiple actions need to be performed on same data in a pipelined faction.

S2

Problem #	Shared/Distributed Memory	Task/Data Parallelism
A large number of arrays (size of array N= 1 million) executing a sum of array elements on a single node.	Shared	Data
Transposing a matrix (size = 1000* 1000) in parallel using more than one node.	Distributed	Data
Using multiple threads to generate a thumbnail for each photo in a collection on cluster machines	Distributed	Data
The fork-join array summation application on MPI cluster.	Distributed	Data
The fork-join array summation application on MPI cluster.	Shared	Task

S3




```
1.  procedure RECURSIVE_MIN (A, n)
2.  begin
3.  if (n = 1) then
4.     min := A[0];
5.  else
6.     lmin := RECURSIVE_MIN (A, n/2);
7.     rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
8.     if (lmin < rmin) then
9.        min := lmin;
10.    else
11.       min := rmin;
12.    endelse;
13. endelse;
14. return min;
```

S4

```
#pragma omp parallel
#pragma omp for schedule (static) || schedule (static, 3)
   for(i = 0; i < 24; i++)
     c[i] = a[i] + b[i];

(static)  ****** ****** ****** ******
(static,3) *** *** *** *** *** *** *** ***
```

S5

(a)

- **Step #1:** In non-blocking mode send or receive call returns before: the data is copied out of the buffer (send) or data has been received and copied into the buffer (receive). The transmission and reception of messages can proceed concurrently (using support hardware) with the computations performed by the program upon the return of the above functions.
- **Step # 2:** At a later point in the program, a process that has started a non-blocking send or receive operation must make sure that this operation has completed before it proceeds with its computations (ensuring that the send has taken place OR the data has actually received). This is because a process that has started a non-blocking send operation may want to overwrite the buffer that stores the data that are being sent, or a process that has started a non-blocking receive operation may want to use the data it requested.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request)
```

(b)

MPI provides a pair of functions MPI_Test and MPI_Wait. The first tests whether or not a non-blocking operation has finished and the second waits (i.e., gets blocked) until a non-blocking operation actually finishes.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

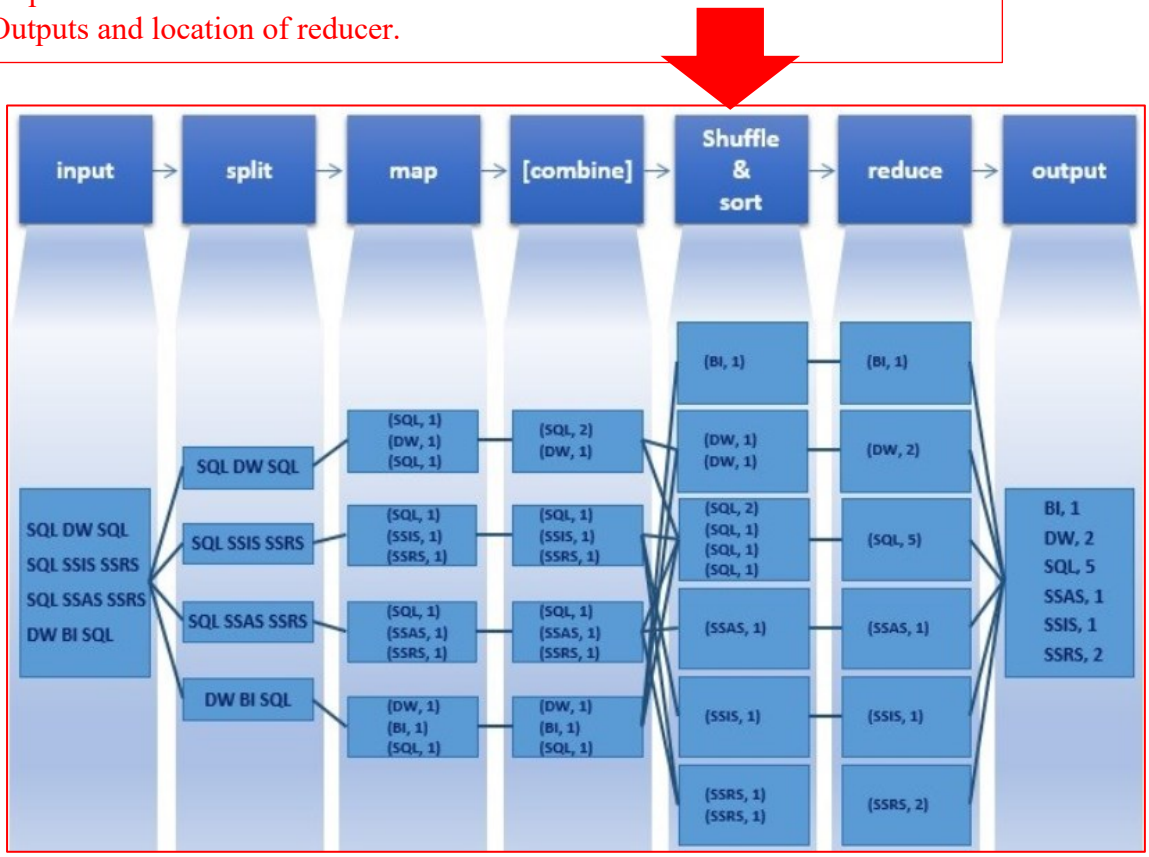
S6

```
#include<iostream.h>
#include<mpi.h>

int main(int argc, char ** argv){
  int mynode, totalnodes;
  int sum,startval,endval,accum;
  MPI_Status status;

  MPI_Init(argc,argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalprocs); // get totalprocs
  MPI_Comm_rank(MPI_COMM_WORLD, &myid); // get myid

  sum = myid + 1; // rank is an integer ranging from 0 to totalprocs-1

  if(myid != 0)
    MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
  else
    for(int j=1;j<totalprocs;j=j+1) {
      MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
      sum = sum + accum;
    }

  if(myid == 0)
    cout << "The sum from 1 to 1000 is: " << sum << endl;
  MPI_Finalize();
}
```

S7

Minimum grading criteria (2 point each).

- Mapper invoked for each line of split.
- Outputs and location of map outputs
- Combiner is reducer code running at each mapper.
- Explain Shuffle and Sort done by the framework before feeding key-wise output to reducer.
- Outputs and location of reducer.



S8.

```
//
//Q8(a):How much data is requested per unique URL? The output file contains URL and total data requested.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"REQ_URL"); // string operation per line
   string& p_value = get_value (str,"DATA_SENT"); // strings operations per line
   Emit(p_key,p_value);
}
Reduce(ReduceInput* input) {
   // Iterate over all entries with the same key and concatenate the values
   int64 value = 0;

   while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
   }
   Emit(value); // Emit sum for input->key()
}
```

```
//
// Q8(b) Which host provided the URL to the user? The output file contains hostname and a list of URLs.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"HOST"); // string operation per line
   string& p_value = get_value (str,"REQ_URL"); // strings operations per line
   Emit(p_key,p_value);
}
Reduce(ReduceInput* input) {
   // Iterate over all entries with the same key and concatenate the values. Also count
   string& p_value_list;
   int64 count = 0;
   while (!input->done()) {
      count++;
      p_value_list = p_value_list + "," + input->value();
      input->NextValue();
      p_value_list = IntToString(count) + " URLs. List: " + p_value_list;
   }
   Emit(p_value_list); // Emit sum for input->key()
}
```

```
//
//c)  Count the number of hosts in the cluster. The output file contains the sentence
//    "This cluster consists of 250 hosts", where 250 is the computed value.
//
Map(const MapInput& input) {
   const string& str = input.value();
   const int n = text.size();

   string& p_key = get_key (str,"HOST"); // string operation per line
   if check_valid_host(p_key) // check validity of host
      Emit("HOST",1); // for counting we ignore host name and use only "HOST" string
}
Reduce(ReduceInput* input) {  // As there is only one key HOST this is the only reducer in the system
   // Iterate over all entries with the same key and concatenate the values
   int64 value = 0;
   string& p_value_list;

   while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
   }
   p_value_list = "This cluster consist of " + IntToString(value) + "hosts";
   Emit(p_value_list); // Emit sum for input->key()
}
```

S9

```
__global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {
// Calculate the row # of the d_Pin and d_Pout element to process
int Row = blockIdx.y*blockDim.y + threadIdx.y;
// Calculate the column # of the d_Pin and d_Pout element to process
int Col = blockIdx.x*blockDim.x + threadIdx.x;
// each thread computes one element of d_Pout if in range
if ((Row < m) && (Col < n)) {
d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
}
```

i) Blocks used 5x4= 20

ii)
```
if ((Row < m) && (Col < n)) {
    d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

S10.

1.
```
cudaMalloc((void **) &d_A, size);
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMalloc((void **) &B_d, size);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

cudaMalloc((void **) &d_C, size);
```

2.
```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

3.
```
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory for A, B, C
cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
```