Q1.

a) Parallel tasks often produce some quantity that needs to be summed together. One such condition exists in the code shown in Figure 1 where result variable is summed. i) State the problem in the code (one sentence only), and ii) show new code by adding/correcting statements which produces correct results. Use OMP constructs and clauses only.

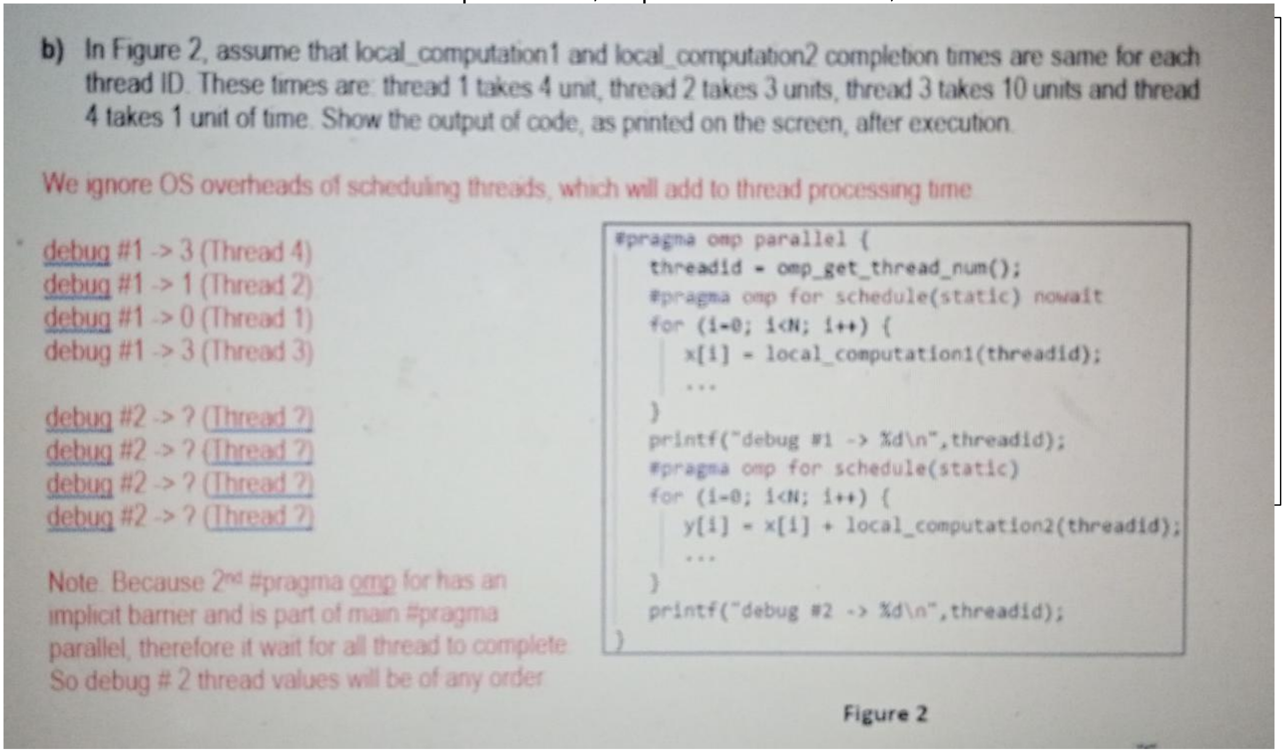
```
double result = 0;
#pragma omp parallel {
    double local_result;
    int num = omp_get_thread_num();
    if (num==0) local_result = f(x);
    else if (num==1) local_result = g(x);
    else if (num==2) local_result = h(x);
    result += local_result;
}
```

Figure 1

i) Multiple threads will try to update variable “result” by adding their private variable “local_result” overwriting previous value.

ii) Any one of the following:
#pragma omp parallel reduction (+: result)
OR
#pragma omp critical
result +=local_result;

b) In Figure 2, assume that local_computation1 and local_computation2 completion times are same for each thread ID. These times are: thread 1 takes 4 unit, thread 2 takes 3 units, thread 3 takes 10 units and thread 4 takes 1 unit of time. Show the output of code, as printed on the screen, after execution.



b) In Figure 2, assume that local_computation1 and local_computation2 completion times are same for each thread ID. These times are: thread 1 takes 4 unit, thread 2 takes 3 units, thread 3 takes 10 units and thread 4 takes 1 unit of time. Show the output of code, as printed on the screen, after execution.

We ignore OS overheads of scheduling threads, which will add to thread processing time

debug #1 -> 3 (Thread 4)
debug #1 -> 1 (Thread 2)
debug #1 -> 0 (Thread 1)
debug #1 -> 3 (Thread 3)

debug #2 -> ? (Thread ?)
debug #2 -> ? (Thread ?)
debug #2 -> ? (Thread ?)
debug #2 -> ? (Thread ?)

Note: Because 2nd #pragma omp for has an implicit barrier and is part of main #pragma parallel, therefore it wait for all thread to complete So debug # 2 thread values will be of any order

```
#pragma omp parallel {
    threadid = omp_get_thread_num();
    #pragma omp for schedule(static) nowait
    for (i=0; i<N; i++) {
        x[i] = local_computation1(threadid);
        ...
    }
    printf("debug #1 -> %d\n",threadid);
    #pragma omp for schedule(static)
    for (i=0; i<N; i++) {
        y[i] = x[i] + local_computation2(threadid);
        ...
    }
    printf("debug #2 -> %d\n",threadid);
}
```

Figure 2

Q2.

a) Show an MPI program where each process computes a random number, and the last process finds and prints the maximum generated value. Each process should also print the value of its random number, so that you can check the correctness of your program.

```c
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv) {
    int ierr, num_procs, my_id, root_id;
    float p_random,sum_random;

    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    root_id = num_procs-1;
    p_random = rand();
    printf("Random # %6.9f is generated by process # %d\n",p_random, my_id);

    sum_random = 0;
    MPI_Reduce(&p_random,&sum_random,1,MPI_FLOAT,MPI_MAX, root_id, MPI_COMM_WORLD);

    if (my_id == root_id) printf("Sum is individual random number are %6.9f\n",sum_random);

    ierr = MPI_Finalize();
}
```

b) Write an MPI pseudocode of a SPMD program where process 0 first reads 20K values of integer from a csv file and distribute them equally to all other processes using MPI send/receive primitives. Process 0 also reserve same size chunk of values for itself. Each process calculates the mean of the received chunk of integer number and write output using a printf statement as "Rank # %d: My mean is %6.3f\n", rankid"
Assumptions:

- Assume number of processes = last two digits of your non-zero roll number [e.g k19-12**34**] OR first two digits of your roll number [e.g. k19-**12**34] is last digits are zero.
- Assume buffered semantics when using send and receive primitives.

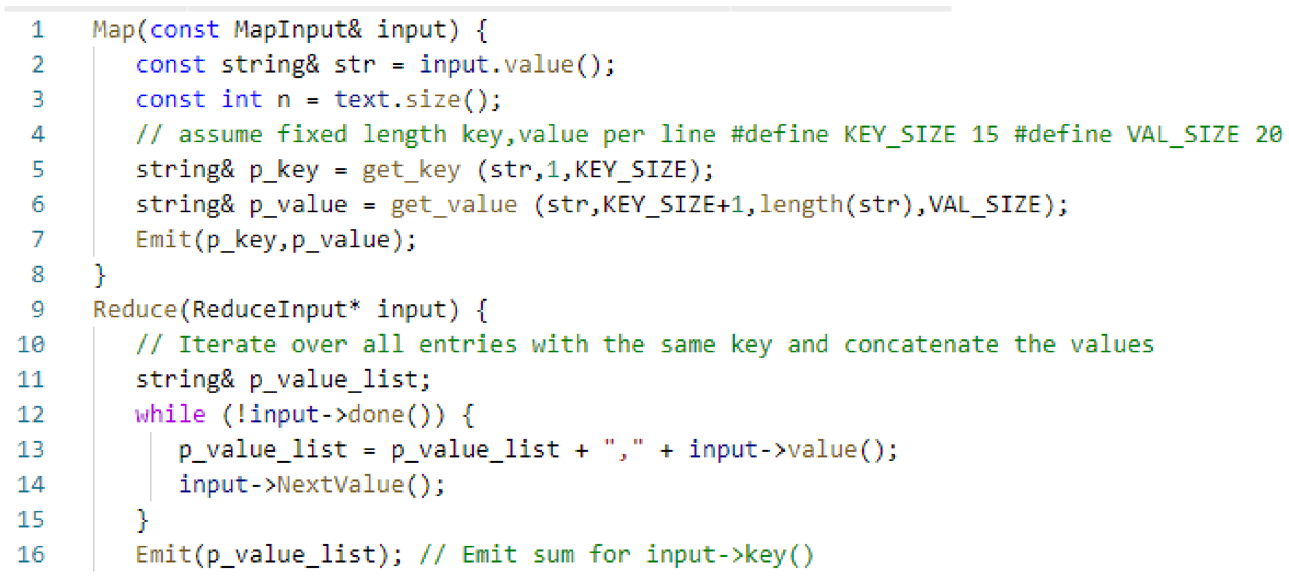
```c
#define MAX_BUFFER 20000
int buffer[MAX_BUFFER], send_buf[MAX_BUFFER], recv_buff[MAX_BUFFER], p;
float p_mean;
MPI_Request request;

// MPI initialization code

// As per assumption. my roll number is K09-0982. mpiexec should be launch with -n 82
root_id = 0;
p_size = MAX_BUFFER/num_procs; //num_procs is 82 (0 - 81). Ignoring round-off problems

if (root_id == 0) {
   read_from_csvfile("%PATH%\mydata\intcsv.txt",MAX_BUFFER, &buffer); // user define function

   int p = 1;
   for (i=p_size; i<= p_size*num_procs, i+=p_size) {
      copy_buffer(&send_buf, &buffer, i,p_size);
      MPI_SEND(&send_buf,p_size,MPI_INT,p,100,MPI_COMM_WORLD,&request); //Tag = 100, send to all processes
      p += 1;
   }
   p_mean = 0.0;
   p_mean = cal_mean(&send_buf,p_size); // calculate mean
   printf("Rank # %d: My mean is %6.3f\n", my_id, p_mean);
}
else {

   MPI_RECV(&recv_buf,p_size,MPI_INT,0,100,MPI_COMM_WORLD,&request); //Tag = 100, recieve from process 0
   p_mean = 0.0;
   p_mean = cal_mean(&recv_buf,p_size); // calculate mean
   printf("Rank # %d: My mean is %6.3f\n", my_id, p_mean);

}
```

Q3

a) How replication of 128MB file chunks in HDFS helps the MapReduce programming paradigm when facing node failures? Explain your answer using the word count example.

Each chunk will be replicated 3 or 5 times on many nodes using rack awareness. In case of failure of a node on which a word count mapper for a given chunk is running, the master can consult the HDFS Namenode to get node ID which have the replicated same chunk and schedule same mapper on that node.

b) Assume a 5TB csv file containing <key, value> pairs is uploaded on an HDFS cluster. The key field contains an alpha numeric sensor name (e.g. sensor1, oiltempration23, etc.) and the value is a 20 digit floating point values (e.g. 76253464.56495). Therefore the <key, value> pair becomes <sensor1, 76253464.56495>. Assume 50 unique sensor names in the csv file.
Show commented pseudocode of Mapper and Reducer functions that will produce outputs files containing sensor wise list of values.



```
Map(const MapInput& input) {
    const string& str = input.value();
    const int n = text.size();
    // assume fixed length key,value per line #define KEY_SIZE 15 #define VAL_SIZE 20
    string& p_key = get_key (str,1,KEY_SIZE);
    string& p_value = get_value (str,KEY_SIZE+1,length(str),VAL_SIZE);
    Emit(p_key,p_value);
}
Reduce(ReduceInput* input) {
    // Iterate over all entries with the same key and concatenate the values
    string& p_value_list;
    while (!input->done()) {
        p_value_list = p_value_list + "," + input->value();
        input->NextValue();
    }
    Emit(p_value_list); // Emit sum for input->key()
```