Homework 3

Due: Mar 15th @ 11:59 pm (as a typed, pdf file)

1. **OpenMP tasks:** Use OpenMP tasks to parallelize the following sequential code, you can assume that x, y, z, f, g, h, i, and j are all properly defined. Compute each of the variables a-e using a task *specific* to that variable.

```
double x = …;
double y = …;
double z = …;

double a = f(x);
double b = g(y);
double c = h(a,b);
double d = i(x,c);
double e = j(c,d,z);
```

We have provided starting code for you below:

```
double x = …;
double y = …;
double z = …;

double a,b,c,d,e;

#pragma omp parallel
{

// Compute a,b,c,d, and e in parallel using tasks.






}
```

2. **Pthreads**: Write a shared memory program using pthreads that computes the average of a distribution contained in array A in parallel by breaking the array A into smaller chunks. Each part a)-c) is marked in the comments for you to implement.

```
struct thread_info_struct {
      int start;  // start index of chunk for thread num
      int end;    // end index of chunk for thread num
      int* A;     // pointer to array
      pthread_t thread;
      int partial_sum;
}

void main() {
      /* assume thread attributes are initialized here */

      thread_info_struct thread_info[num_threads];
      int* A = (int*)malloc(1000*sizeof(int));
      assert (A != NULL);

      //initialize array
      for (int i = 0; i < 1000; i++) A[i] = rand() % 1000;

      for (int i = 0; i < num_threads; i++) {
            /* a) initialize thread_info_struct and create threads to compute
            partial_sum in parallel */

      }
      // join threads
      for (int i = 0; i < num_threads; i++) {
            pthreads_join(thread_info[i].thread, NULL);
      }

      double average = 0;
      /* b) compute average from partial sums */

      free(local_sum);
}

void* partial_sum(void* my_info) {
      /* c) Add your code here to compute local sum*/

}
```

3. **C++ Atomics:** Here is an incorrect implementation of a barrier, please explain where the error(s) is/are and what can go wrong if this is used.

```
std::atomic<int> x; // initialized to 0
int t = ...; // number of threads
void barrier() {
      int my_x = x.fetch_add(1);
      if (my_x == t) {
            x.store(0);
      } else {
            while (x.load() != t);   // spin-wait
      }
}
```

(Hint: this implementation **will** work if only one barrier is needed throughout the entire program execution. Think about when multiple barriers are needed in the program.)

4. **MPI:** A tree has been constructed out of P processes such that process 0 is the root and the left and right children of a process i are processes 2*i+1 and 2*i+2 respectively. Each process contains an integer my_val.

Please use point-to-point communication **only** (i.e. send/recv, no collectives) to implement the MPI reduction collective such that process 0's sum_val is the sum of each of the P process' my_vals. Your implementation should be equivalent to this MPI call:

```
MPI_Reduce(&my_val, &sum_val, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

```
// sum_val for process with rank 0 will be the sum of all my_val's
void my_reduce(int *my_val, int *sum_val) {
      int my_rank = …; // assume already computed
      int num_ranks = …; // assume already computed

      /* TODO: complete this function */
      int parent, leftchild, rightchild;





}
```