Homework 3

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

Nathan Nard (nnard2)

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
#pragma omp single
    {
        #pragma omp task shared(a) depend(out: a)
        a = f(x);
        #pragma omp task shared(b) depend(out: b)
        b = g(y);
        #pragma omp task shared(a,b,c) depend(in:a,b) depend(out:c)
        c = h(a, b);
        #pragma omp task shared(c,d) depend(in:c) depend(out:d)
        d = i(x, c);
        #pragma omp task shared(c,d,e) depend(in:c,d)
        e = j(c, d, z);
    }
```

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 */
            thread_info[i].thread = (pthread_t)i;
            thread_info[i].start  = i*1000/num_threads;
            thread_info[i].end    = (i+1)*1000/num_threads;
            thread_info[i].A      = A;
            thread_info[i].partial_sum = 0;
            pthread_create(&thread_info[i].thread, NULL, partial_sum,
                           (void*)&thread_info[i]);
      }
      // 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 */
      double average = 0;
      for (int i = 0; i < num_threads; i++){
          average += thread_info[i].partial_sum;
      }
      average /= 1000;

      // Didn't need to malloc a "local_sum" variable, so it is commented out.
      // free(local_sum);
}
// continued on next page
```

```
void* partial_sum(void* my_info) {
      /* c) Add your code here to compute local sum*/
    thread_info_struct* contents = (thread_info_struct*) my_info;

    for (int i = contents->start; i < contents->end; i++){
        contents->partial_sum += contents->A[i];
    }

    return nullptr;
}
```

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.)

As mentioned in the hint, this will work for just one call to barrier(), but will fail for each subsequent call. The reason why it fails for each subsequent call is the if block that contains the x.store(0) command doesn't get evaluated until the next call to barrier. The result of this causes the first thread in the next barrier() call to reset the x value to 0, but that thread then never increments x nor does it get caught by the spin-wait while loop. So it will continue on past the barrier while the other t-1 threads get stuck in the spin-wait while loop, causing the program to hang as the value of x gets stuck at t-1. One could change x.store(0) to x.store(1) to stop the program from hanging, but then the barrier() method would still not be correct as one thread (the first one that comes in and sets x to 1) would be able to get past the barrier while the others get caught up in it. So then we can remove the else block and place the while loop outside it so every thread will run into it. A more correct refactored code:

```
void barrier() {
    int my_x = x.fetch_add(1);
    if (my_x == t) {
        x.store(1);
    }
    while (x.load() != t); //spin-wait
}
```

However, this is still incorrect. The comparison in the if block is not atomic (ie, not thread safe), so it is possible for a thread to get into the if block, but then not set x to 1 before another thread increments x in the previous fetch_add() line, thereby causing x to never increment all the way to t. The best solution is to not reset the x value each time you use the barrier() method, but instead increment x "indefinitely" and just check to see if x is a multiple of t in order to terminate the while loop. Hopefully the x value never exceeds the max integer size during your program. Here is the best implementation for barrier():

```
void barrier() {
   x.fetch_add(1);
   while (x.load()%t != 0); //spin-wait
}
```

There may a way to use atomic_compare_exchange_strong to supplant the comparison above, but at present I am unable to construct a working implementation using that method.

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, num_ranks;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

    // all but process 0 have a parent
    int parent     = (my_rank == 0) ? NULL : (my_rank-1)/2;
    int leftchild  = 2*my_rank + 1;
    int rightchild = 2*my_rank + 2;

    /************ handle recvs *************/
    int leftchild_recv  = 0;
    int rightchild_recv = 0;

    // power2Ceiling needed for non-full binary trees.
    int power2Ceiling = pow(2, (int)(log2(num_ranks)+1))-1;

    // processes in last row of binary tree do not have children,
    // so they don't receive anything
    if (my_rank < (power2Ceiling - 1)/2){
        if (leftchild < num_ranks)
            MPI_Recv(&leftchild_recv , 1, MPI_INT, leftchild , 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (rightchild < num_ranks)
            MPI_Recv(&rightchild_recv, 1, MPI_INT, rightchild, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    /************ handle local_sum *********/
    int local_sum = 0;
    local_sum += leftchild_recv;
    local_sum += rightchild_recv;
    local_sum += *my_val;

    /************ handle sends *************/
    if (my_rank != 0){
        MPI_Send(&local_sum, 1, MPI_INT, parent, 0, MPI_COMM_WORLD);
    } else { // my_rank == 0

        // process 0 has no parent to send to,
        // thus we're done so return local_sum.
        *sum_val = local_sum;
    }
}
```