Name: ___________________________

Directions: MAKE SURE TO COPY YOUR ANSWERS TO A SEPARATE SHEET FOR SENDING ME AN ELECTRONIC COPY LATER.

1. (45) The code below counts primes, in a manner similar to our **pthreads** example. The function **crossout()**, not shown, is the same as in that example, except for the obvious new arguments. Fill in the blanks.

```
// OMP code program to find the number
// of primes between 2 and n; not
// claimed to be efficient

int primecounter(int n) {
   int *prime;
   int nextbase=3,tot=0;
   blank (a)
   {  int nth = omp_get_num_threads();
      int me = omp_get_thread_num();
      int i,base;
      blank (b)
      {
         blank (c)
         for (i = 3; i <= n; i += 2)
            prime[i] = 1;
      }
      while(1) {
         blank (d)
         {
            base = nextbase;
            nextbase += 2;
         }
         if (base > sqrt(n)) break;
         if (prime[base])
            crossout(prime,n,base);
      }
      blank (e)
      int mytot = 0;
      blank (f)
      for (i = 3; i < n; i += 2)
         mytot += prime[i];
      blank (g)
      blank (h)
   }
   return blank (i)
}
```

2. (45) Here we use R **snow** to count primes. We break the vector 3,5,7,9,... into chunks, one chunk for each cluster node, and have each cluster node count primes in its chunk. To do the latter, we divide by all the numbers in **divvec**, which is a vector of all the primes up through $\sqrt{n}$, which we find serially. For instance, say n is 1000. Then we find the primes up to $\sqrt{1000}$, which turn out to be 2,3,5,7,11,13,17,19,23,29,31; those 11 numbers form **divvec**, which we apply to finding primes up through 1000. Fill in the blanks.

```
# parallel prime counter, not claimed efficient
# serial prime finder; can be used to generate
# divisor vector
serprime <- function(n) {
   ...
}

# apply divvec to one chunk of the prime vector,
# return count of primes there
processchunk <- function(primechunk,divvec) {
   count <- 0
   for (i in primechunk) {
      # note blank (a)!
      if (all(i %% divvec blank (a) ))
         count <- count + 1
   }
   count
}

primecount <- function(cls,n) {
   # generate the vector 3,5,7,9..., through n
   prime <- seq(3,n,2)
   # serially find the primes up through sqrt(n)
   divvec <- serprime(ceiling(sqrt(n)))
   # remove those from our prime vector
   prime <- setdiff(prime,divvec)
   # break prime vector into chunks
   ixchunks <-  blank (b)
   getchunk <- function(ixchunk) blank (c)
   primechunks <- Map(getchunk,ixchunks)
   # send those chunks to the cluster nodes,
   # calling processchunk() at each node
   counts <- blank (d)
   # put it all together
   blank (e)
}
```

3. (10) Consider the pipelined prime-finding MPI code, pp.18ff. Say we have 10 nodes. Fill in the blanks: Node 8 will spend blank (a) time in line blank (b) than will Node 1.

Solutions:

1.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>  // required for threads usage

// OMP code program to find the number of primes between 2 and n; not
// claimed to be efficient

void crossout(int *prime, int n, int k)
{  int i;
   for (i = 3; i*k <= n; i += 2)  {
      prime[i*k] = 0;
   }
}

int primecounter(int n) {
   int *prime;
   int nextbase=3,tot=0;
   #pragma omp parallel
   {  int nth = omp_get_num_threads();
      int me = omp_get_thread_num();
      int i,base;
      #pragma omp single
      {
         prime = malloc((n+1)*sizeof(int));
         for (i = 3; i <= n; i += 2)
            prime[i] = 1;
      }
      while(1) {
         #pragma omp critical
         {
            base = nextbase;
            nextbase += 2;
         }
         if (base > sqrt(n)) break;
         if (prime[base])
            crossout(prime,n,base);
      }
      #pragma omp barrier
      int mytot = 0;
      #pragma omp for
      for (i = 3; i < n; i += 2)
         mytot += prime[i];
      #pragma omp critical
      tot += mytot;
   }
   return tot + 1;
}

main(int argc, char **argv)
{  int n = atoi(argv[1]);
   printf("%d\n",primecounter(n));
}
```

2.

```
# Snow code to count primes through n;
# not efficient

# serial prime finder; can be used to generate
# divisor list
serprime <- function(n) {
   nums <- 1:n
   # all in nums assumed prime until shown otherwise
   prime <- rep(1,n)
   maxdiv <- ceiling(sqrt(n))
   for (d in 2:maxdiv) {
      # don't bother dividing by nonprimes
      if (prime[d])
```

```
            # try divisor d on numbers not yet
            # found nonprime
            tmp <- prime !=0 & nums > d & nums %% d == 0
            prime[tmp] <- 0
        }
        nums[prime != 0 & nums >= 2]
    }

    # apply divvec to one chunk of the prime vector,
    # return count of primes theree
    processchunk <- function(primechunk,divvec) {
        count <- 0
        for (i in primechunk) {
            if (all(i %% divvec > 0))
                count <- count + 1
        }
        count
    }

    primecount <- function(cls,n) {
        # generate the vector 3,5,7,9..., through n
        prime <- seq(3,n,2)
        # serially find the primes up through sqrt(n)
        divvec <- serprime(ceiling(sqrt(n)))
        # remove those from our prime vector
        prime <- setdiff(prime,divvec)
        # break prime vector into chunks
        ixchunks <-
            splitIndices(length(prime),length(cls))
        getchunk <- function(ixchunk)
            prime[ixchunk]
        primechunks <- Map(getchunk,ixchunks)
        # send those chunks to the cluster nodes,
        # calling processchunk() at each node
        counts <- clusterApply(cls,primechunks,
            processchunk,divvec)
        # put it all together
        Reduce(sum,counts) + length(divvec)
    }
```

3. As we go deeper into the pipe, each node has less work to do. That means the later nodes wait more time from one data receipt to the next, i.e. more time on line 83.