Name: ____________________________

Directions: **Work only on this sheet** (on both sides, if needed). MAKE SURE TO COPY YOUR ANSWERS TO A SEPARATE SHEET FOR SENDING ME AN ELECTRONIC COPY LATER.

**Important note:** Remember that in problems calling for R code, you are allowed to use any built-in R function, e.g. **choose()**, **sum()**, **combn()** etc.

**1.** There are various high-level threads access systems other than OpenMP. One of them is a language called Cilk++, developed at MIT and purchased by Intel. Judging from the names of the Cilk++ constructs below, give names of OpenMP or pthreads constructs that should roughly correspond.

(a) (5) **cilk_for**

(b) (5) **cilk::reducer_opadd**

(c) (5) **cilk::mutex**

(d) (5) **cilk_spawn** (a dictionary definition of *spawn*: "to produce or create something")

**2.** (10) Explain briefly why R's **snow** library would be a poor choice—essentially an impossible one—for pipelined parallel algorithms such as in our MPI example, Sec. 1.3.3.2.

**3.** Consider the in-place matrix transposition code in Sec. 4.3.4.

(a) (10) Fill in the blank: Since each thread works on completely separate elements of the matrix, there "should" not be a lot of cache coherency transactions. But there probably will be, due to the problem of __________________.

(b) (10) Give a potential improvement to one (1) line of the code.

**4.** (10) Consider the Dijkstra example, pp.71ff. Suppose that in the end, shortest distances from vertex 0 to vertex i are roughly correlated with i, i.e. vertices with larger values of i tend to be further from 0. Comment on performance issues that would likely arise. Cite certain variables and/or lines so that it is clear that you understand the issues, but be brief. As usual, you are limited to a single, hopefully not very long, line.

**5.** (40) The function below is written in R, but could be applied to any scheduling situation; it is merely an analytical tool. The call form is **statictime(tasktms,nth)**, where we have **nth** threads, and **tasktms** are the task times, assumed here to be known in advance.

Say for instance we have 6 tasks, needing times **tasktms[1]** through **tasktms[6]**, and 2 threads. We would have some kind of parallel **for** loop, iterating **i** through 1 to 6; one thread would handle some values of **i**, and the other thread would handle the others.

Here we assume the **static** scheduling algorithm used by OpenMP, and the function will return the time needed to complete all the tasks. Fill in the blanks.

```
statictime <- function(tasktms,nth) {
   n <- length(tasktms)
   a1ton <- 1:n
   endtimes <- vector(length=nth)
   for (i in 1:nth) {
      # determine which tasks thread i will handle
      if (i != nth) {
         this1does <-
            # blank (a)
      } else
         this1does <-
            # blank (b)
      # blank (c)
   }
   # blank (d)
}
```

**Solutions:**

**1.a** OMP **for** pragma

**1.b** OMP **reduction** clause, with '+'

**1.c pthread_mutex_lock()**

**1.d pthread_create()**

**2.** It would be impossible to get parallelism this way, without direct communication between the workers.

**3.a** false sharing

**3.b** One could try special scheduling, say **dynamic**, on line 12.

**4.** Entry of the vertices into the **nondone** array will roughly occur in order of i, so that soon the low-numbered threads have little or no work to do in line 54.

**5.** The problem was in part incorrectly specified, as it assumed (without saying so) a chunk size of 1. The code below is written under that assumption.

In actuality, the default for **static** scheduling is to divide the iterations in approximately equal-sized chunks. In our situation here, we could fill blank (a) with

```
((i-1) * floor(n/nth) + 1):(i * floor(n/nth)
```

and do something similar for blank (b).

This does not affect the answers to blanks (c) and (d).

```
statictime <- function(tasktms,nth) {
   n <- length(tasktms)
   a1ton <- 1:n
   endtimes <- vector(length=nth)
   for (i in 1:nth) {
      # determine which tasks thread i will handle
      if (i != nth) {
         this1does <-
            which(a1ton %% nth == i)
      } else
         this1does <-
            which(a1ton %% nth == 0)
      endtimes[i] <- sum(tasktms[this1does])
   }
   max(endtimes)
}
```