Name: _____

Directions: Work only on this sheet (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing. In order to get full credit, SHOW YOUR WORK.

1. (10) Fill in the blank: The **struct** field **cycle** in the complex barrier code presented in discussion section corresponds to the field _____ in the barrier code in our PLN.

2. (15) In our PLN, the way we applied a lock operation on a lock **L** before entering a critical section **C** was as follows:

```
TRY:  TAS R,L
      JNZ TRY
C:    ...
      MOV L,0
```

Consider this variant:

```
      TAS R,L
      JNZ DO_SOMETHING_ELSE
C:    ...
      MOV L,0
```

What Pthreads call is this variant analogous to?

3. Consider the discussion on read/write locks in Sec. 7.8.1 in Grama.

(a) (5) By using read/write locks instead of ordinary locks, the mean time a read must wait to execute will _____. (Answer *increase* or *decrease*.)

(b) (5) Answer [(b)] for writes.

(c) (5) Suppose reads are currently in progress and suddenly some thread requests a write. Choose one: (i) The write will pre-empt the reads. (ii) The write will not pre-empt the reads, and will not run as long as there are reads to be done. (iii) The write and the reads will take turns running, a quantum (or timeslice) at a time. (iv) One can't predict what will occur.

(d) (5) Suppose a write is currently in progress, and suddenly some thread requests a read and some other thread requests a write. Choose one: (i) When the first write completes, the read will run. (ii) When the first write completes, the second write will run. (iii) When the first write completes, the read and the second write will take turns running. (iv) The read and the two writes will take turns running. (v) One can't predict what will occur.

4. (15) Consider the graph in the section of our PLN on shortest-path computation. Add one more vertex v5, which is a distance 2 away from each of v1 and v4. Suppose we apply the MPI code in Sec. 6.6.9 of Grama to this problem, running on two nodes. (So our **my.pg** file would consist of two entries.) Show the entire contents of the (one-dimensional) array **wgt** for the node having rank 1.

5. We are interested in finding the root of a strictly monotone (strictly increasing or strictly decreasing) function **g()**, in an iterative manner, by inspecting narrower and narrower intervals. The initial interval to inspect is [**left**,**right**], and we iterate until the interval width is less than **eps** (say 0.0001). At each iteration, the current interval is divided into equal subintervals, one for each node. Exactly one of the nodes will discover a sign change in **g()** across its subinterval. That subinterval becomes the new interval to be used in the next iteration. You may wish to read part (b) first, to get a more concrete idea of how the root finding works. Your code is required to be reasonably efficient in terms of speed.

(a) (20) Write a JIAJIA function to find the root:

```
float findroot(float (g)(float), float
   left, float right, float eps,
   int numnodes, int me, int *winner, int *workdone)
```

The arguments **winner** and **workdone** are the identity of the node which found the subinterval in which the sign change occurred, and a shared array which keeps track of work done.[1] At the end of execution, **workdone**[**i**] will contain a count of the number of times node **i** was the winner. Use only the shared-memory functions of JIAJIA, not the "MPI-like" message-passing ones.

[1] The **winner** argument was not present in the original version of the exam, and students directly accessed a global shared variable.

(b) (20) The MPI code for root finding shown below has a gap, missing one statement. Fill in the gap in the code with a single statement, consisting of a single fully-specified MPI call. The arguments **numnodes** and **me** are the number of nodes and the node number of this machine.

```
float findroot(float (g)(float), float
   left, float right, float eps,
   int numnodes, int me, MPI_Comm comm)
{  float a=left,b=right,wid,t1,t2;
   int winner,y[2],z[2];
   MPI_Status status;  // see below

   while (1)  {
      wid = (b-a)/numnodes;
      t1 = g(a+me*wid); t2 = g(a+(me+1)*wid);
      y[0] = floor(t1*t2); y[1] = me;
      // gap:  place a single MPI call here
      winner = z[1];
      a = a+winner*wid;  b = a+(winner+1)*wid;
      if (b-a < eps) return (a+b)/2.0;
   }
}
```

Solutions:

1. EvenOdd

2. pthread_mutex_trylock()

3. decrease; increase; (ii); (ii)

4. The array stores the last threee columns of the one-hop distance matrix, in row-major order. So its contents are: $(\infty, \infty, \infty, 10, 5, 2, 100, \infty, \infty, 0, \infty, \infty, \infty, 0, 2, \infty, 2, 0)$

5.a

```
...
wid = (b-a)/numnodes;
t1 = g(a+me*wid); t2 = g(a+(me+1)*wid);
if (t1*t2 < 0) {
   *winner = me;
   workdone[me]++;
}
jia_barrier();
a = a+*winner*wid;  b = a+(*winner+1)*wid;
if (b-a < eps) return (a+b)/2.0;
```

5.b

```
status = MPI_Allreduce(y,z,1,MPI_2INT,MPI_MINLOC,comm);
```