Name: ____________________________

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

**1.** (15) The online help for the **clusterApply()** function in R's **parallel** package says,

> clusterApplyLB is a load balancing version of clusterApply. If the length p of seq is not greater than the number of nodes n, then a job is sent to p nodes. Otherwise the first n jobs are placed in order on the n nodes. When the first job completes, the next job is placed on the node that has become free; this continues until all jobs are complete. Using clusterApplyLB can result in better cluster utilization than using clusterApply, but increased communication can reduce performance. Furthermore, the node that executes a particular job is non-deterministic.

Fill in the blanks: This is similar to the ____________ option in ____________ programming, with chunk size ____________.

**2.** (65) Here you will work on a Thrust version of the CUDA code in our last quiz, which solved a problem similar to the root finding example in Section 4.11. It finds the root of a user-supplied function **f()**, which is increasing on (0,1) and has a root somewhere inside. The initial search interval is (0,1), but the interval gets smaller with each iteration. At any iteration, the current interval is divided in subintervals, with each thread handling one subinterval. Fill in the blanks.

```
// Thrust example:  find the root of an
// increasing function on (0,1); not
// assumed efficient

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/remove.h>
#include <thrust/sequence.h>

__host__ __device__ float f(float x) {
   return x*x - 0.5;
}

struct signchange {
   float width;
   thrust::device_vector<float>::iterator ab;
   signchange(
      ________,  // blank (a)
      float _width):
         ab(_dab),width(_width) {}
   __host__ __device__
   bool operator()(int i)
   {  if (_________)  // blank (b)
            return true;
      else return false;
   }
};

// do niters iterations, with nsubintervals
// checked each time; typically would want
// nsubintervals = number of threads
float throot(int niters, int nsubintervals)
{  int iter;
   thrust::host_vector<float> hab(2);
   hab[0] = 0.0;
   hab[1] = 1.0;
   float width;  // subinterval width
   thrust::device_vector<float> dab(hab);
   thrust::host_vector<int> hfoundit(1);
   thrust::device_vector<int> dfoundit(1);
   thrust::device_vector<int>
      seq(nsubintervals);
   thrust::sequence(seq.begin(),seq.end(),0);
   for (iter = 0; iter < niters; iter++) {
      width =
         (hab[1] - hab[0]) / nsubintervals;
      __________(  // blank (c)
         ____________  // blank(d), contains
                       // .begin(), .end()
         ________________  // blank (e)
         signchange(dab.begin(),width));
      thrust::copy(dfoundit.begin(),
         dfoundit.end(),hfoundit.begin());
      hab[0] = _____________  // blank (f)
      hab[1] = hab[0] + width;
      thrust::copy(hab.begin(),hab.end(),
         dab.begin());
   }
   return _________; // blank (g)
}

// test case
int main(int argc, char **argv)
{  float root;
   int niters = atoi(argv[1]),
       nsubintervals = atoi(argv[2]);
   root = throot(niters,nsubintervals);
   printf("%f\n",root);
}
```

**3.** Suppose we wish to use Thrust to compress an upper-triangular matrix, storing only the upper-triangular portion, column by column. For instance, the matrix

$$\begin{pmatrix} 5 & 12 & 13 \\ 0 & 168 & 8 \\ 0 & 0 & 1 \end{pmatrix}$$

would be stored as (5,12,168,13,8,1).

(a) (10) Which would be appropriate here, a Thrust scatter or gather operation?

(b) (10) For a $4 \times 4$ input matrix, what would be the appropriate map vector, given your answer in (a)? Assume row-major order. Answer in vector form, e.g. (8,88,-2,-6).

**Solutions:**

**1.** dynamic; OpenMP; 1

**2.**

```
// Thrust example:  find the root of an increasing function on (0,1)

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/remove.h>
#include <thrust/sequence.h>

__host__ __device__ float f(float x) {
   return x*x - 0.5;
}

struct signchange {
   float width;
   thrust::device_vector<float>::iterator ab;
   signchange(thrust::device_vector<float>::iterator _dab,
              float _width):
         ab(_dab),width(_width) {}
   __host__ __device__
   bool operator()(int i)
   {  if (f(ab[0]+i*width) < 0 &&
         f(ab[0]+(i+1)*width) > 0)
            return true;
      else return false;
   }
};

// do niters iterations, with nsubintervals checked each time; typically
// would want nsubintervals = number of threads
float throot(int niters, int nsubintervals)
{  int iter;
   thrust::host_vector<float> hab(2);
   hab[0] = 0.0;
   hab[1] = 1.0;
   float width;  // subinterval width
   thrust::device_vector<float> dab(hab);
   // index of subinterval where sign change is found
   thrust::host_vector<int> hfoundit(1);
   thrust::device_vector<int> dfoundit(1);
   thrust::device_vector<int> seq(nsubintervals);
   thrust::sequence(seq.begin(),seq.end(),0);
   for (iter = 0; iter < niters; iter++) {
      width = (hab[1] - hab[0]) / nsubintervals;
      thrust::copy_if(seq.begin(),seq.end(),
         dfoundit.begin(),
            signchange(dab.begin(),width));
      thrust::copy(dfoundit.begin(),dfoundit.end(),
         hfoundit.begin());
      hab[0] = hab[0] + hfoundit[0] * width;
      hab[1] = hab[0] + width;
      thrust::copy(hab.begin(),hab.end(),dab.begin());
   }
   return hab[0];
}

// test case
int main(int argc, char **argv)
{  float root;
   int niters = atoi(argv[1]),
       nsubintervals = atoi(argv[2]);
   root = throot(niters,nsubintervals);
   printf(”%f\n”,root);
}
```

**3a.** gather

**3b.** (0,1,5,2,6,10,3,7,11,15)