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.

1. (40) Write an R function that computes two-dimensional Discrete Fourier Transforms (DFTs) in parallel. You must use the approach that makes use of separability, described on p.167, and use **snow** as your vehicle for parallelization. Your function will call R's one-dimensional DFT function `fft()`; the call **fft(v)** on a vector **v** returns the DFT of **v**. (That function is also capable of two-dimensional DFTs, but you will not use that here, in order to parallelize the operation.)

The form of your function's call will be **fft2(m)**, where **m** is a matrix. The return value will be a matrix of the same size. Write full code, with clear comments, and **WRITE LEGIBLY**.

2. (30) Below is OpenMP code to compute prefix sums in parallel, using the approach outlined at the bottom of p.130 and top of p.131.

Here is a sample call:

```
int main()
{  int i,u[9] = {5,12,13,5,4,3,8,6,1}, z[3];
   parprfsum(u,9,z);
   for (i = 0; i < 9; i++) printf("%d\n",u[i]);
}
```

The result will be (5,17,30,35,39,42,50,56,57).

Fill in the blanks in the code:

```
#include <omp.h>

// calculates prefix sums sequentially on u, where u is an
// m-element array

void seqprfsum(int *u,int m)
{  int i,s=u[0];
   for (i = 1; i < m; i++) {
      u[i] += s;
      s = u[i];
   }
}

// OMP example, calculating prefix sums in parallel on the
// n-element array x, in-place; for simplicity, assume that n is
// divisible by the number of threads; z is for intermediate
// storage, an array with length equal to the number of threads; x
// and z point to global arrays
void parprfsum(int *x, int n, int *z)
{
   #pragma omp parallel
   {  int i,j,me = omp_get_thread_num(),
                                       // one blank line
          chunksize = n /              // blank
          start =                      // blank
      seqprfsum(&x[start],chunksize);
      #pragma omp                      // blank
      #pragma omp                      // blank
      {
      for (i = 0; i < nth-1; i++)
         z[i] =                        // blank
      seqprfsum(z,nth-1);
      }
      if (                  ) {        // blank
         for (j = start; j < start + chunksize; j++) {
            x[j]                       // blank
         }
      }
   }
}
```

3. (30) Below is CUDA code to solve a linear system of equations via Gaussian elimination. It uses the approach of p.143, except that it reduces to the form $(I|x)$, where **I** is the identity matrix and **x** is the solution to the system. The difference from p.143 is that line 3 in the pseudocode is now

```
for r = 0 to n-1, r != i
```

There is no pivoting, i.e. no swapping of rows if 0s or near-0 values are encountered.

Fill in the blanks:

```
// linear index for matrix element at row i, column j, in an m-column
// matrix
__device__ int onedim(int i,int j,int m) {return i*m+j;}

// replace u by c* u; vector of length m
__device__ void cvec(float *u, int m, float c)
{  for (int i = 0; i < m; i++) u[i] = c * u[i];  }

// multiply the vector u of length m by the constant c (not changing u)
// and add the result to v
__device__ void vplscu(float *u, float *v, int m, float c)
{  for (int i = 0; i < m; i++) v[i] += c * u[i];  }

// copy the vector u of length m to v
__device__ void cpuv(float *u, float *v, int m)
{  for (int i = 0; i < m; i++) v[i] = u[i];  }

// solve matrix equation Ax = b; straight Gaussian elimination, no
// pivoting etc.; the matrix ab is (A|b), n rows; ab is destroyed, with
// x placed in the last column; one block, with thread i handling row i
__global__ void gauss(float *ab, int n)
{  int i,n1=n+1,abii,abme;
   extern __shared__ float iirow[];
   int me = threadIdx.x;
   for (i = 0; i < n; i++) {
      if (                      ) {            // blank
         abii =                                // blank
         cvec(&ab[abii],n1-i,1/ab[abii]);
         cpuv(                   );            // blank
      }
                                               // one blank line
      if (                     ) {             // blank
         abme = onedim(me,i,n1);
         vplscu(iirow,                         // blank
      }
      __syncthreads();
   }
}
```

Solutions:

1. If you have discovered how **parApply()** works on vector-valued functions—placing the result of each row *or* column of the input into a *column* of the output, you can write the code this way:

```
fft2 <- function(cls,m) {
   tmp <- parApply(cls,m,1,fft)
   return(parApply(cls,tmp,1,fft))
}
```

The code using only fundamental ops would run along the following lines:

```
l <- list()
for each row r in m
   add r to l
call clusterApply() on l with fft(), result mm
l <- list()
for each column c in m
   add c to l
call clusterApply() on l with fft(), return result
```

2.

```
#include <omp.h>

// calculates prefix sums sequentially in-place on u, where u is an
// m-element array
void seqprfsum(int *u,int m)
{  int i,s=u[0];
   for (i = 1; i < m; i++) {
      u[i] += s;
      s = u[i];
   }
}

// OMP example, calculating prefix sums in parallel on the n-element
```

```
// array x, in-place; for simplicity, assume that n is divisible by the
// number of threads; z is for intermediate storage, an array with length
// equal to the number of threads; x and z point to global arrays
void parprfsum(int *x, int n, int *z)
{
   #pragma omp parallel
   {  int i,j,me = omp_get_thread_num(),
          nth = omp_get_num_threads(),
          chunksize = n / nth,
          start = me * chunksize;
      seqprfsum(&x[start],chunksize);
      #pragma omp barrier
      #pragma omp single
      {
      for (i = 0; i < nth-1; i++)
         z[i] = x[(i+1)*chunksize - 1];
      seqprfsum(z,nth-1);
      }
      if (me > 0) {
         for (j = start; j < start + chunksize; j++) {
            x[j] += z[me - 1];
         }
      }
   }
}
```

3.

```
#include <stdio.h>

// linear index for matrix element at row i, column j, in an m-column
// matrix
__device__ int onedim(int i,int j,int m) {return i*m+j;}

// replace u by c* u; vector of length m
__device__ void cvec(float *u, int m, float c)
{  for (int i = 0; i < m; i++) u[i] = c * u[i];  }

// multiply the vector u of length m by the constant c (not changing u)
// and add the result to v
__device__ void vplscu(float *u, float *v, int m, float c)
{  for (int i = 0; i < m; i++) v[i] += c * u[i];  }

// copy the vector u of length m to v
__device__ void cpuv(float *u, float *v, int m)
{  for (int i = 0; i < m; i++) v[i] = u[i];  }

// solve matrix equation Ax = b; straight Gaussian elimination, no
// pivoting etc.; the matrix ab is (A|b), n rows; ab is destroyed, with
// x placed in the last column; one block, with thread i handling row i
__global__ void gauss(float *ab, int n)
{  int i,n1=n+1,abii,abme;
   extern __shared__ float iirow[];
   int me = threadIdx.x;
   for (i = 0; i < n; i++) {  // seq through the diagonal for pivots
      if (i == me) {
         abii = onedim(i,i,n1);
         cvec(&ab[abii],n1-i,1/ab[abii]);
         cpuv(&ab[abii],iirow,n1-i);
      }
      __syncthreads();
      if (i != me) {
         abme = onedim(me,i,n1);
         vplscu(iirow,&ab[abme],n1-i,-ab[abme]);
      }
      __syncthreads();
   }
}
```