Problem 1 (10 pts): Recall the **Selection Sort** algorithm. Retrieve source code for the serial version from somewhere convenient.
Create a parallel version of this algorithm which is targeted at a **Distributed Memory Machine** using an appropriate programming paradigm we discussed in class. Presume each processor in the distributed memory machine starts with some portion of the entire array of numbers and at the end of the algorithm, each process should have some of the numbers in sorted order in an array along and the smallest such numbers should be on processor 0, second smallest on processor 1, and so forth as shown below.

```
START:
P0: A[] = { 84 32 21 28 }
P1: A[] = { 17 20 25 85 }
P2: A[] = { 24 11 31 99 }
P3: A[] = { 13 32 26 75 }

GOAL
P0: A[] = { 11 13 17 20 }
P1: A[] = { 21 24 25 26 }
P2: A[] = { 28 31 32 33 }
P3: A[] = { 75 84 85 99 }
```

Problem 2 (10 pts): Recall the **Selection Sort** algorithm. Retrieve source code for the serial version from somewhere convenient.
Create a parallel version of this algorithm which is targeted at a **Shared Memory Machine** using an appropriate programming paradigm we discussed in class. Presume that the array of numbers must be initially read from a single file and can fit into the main memory of the shared memory machine.

Problem 3 (10 pts): Consider the code below which computes and returns the unique numbers which appear in a parameter array.

```
// Compute unique elements of data by checking
void unique1(double *data, int ndata,
            double **set_uniques, int *set_nuniques)
{
  int uniques_size = 5;
  double *uniques = malloc(uniques_size * sizeof(double));
  int nuniques = 0;

  for(int i=0; i<ndata; i++){
    double x = data[i];
    int present = 0;
    for(int j=0; j<nuniques; j++){ // Check if x is
      if(x == uniques[j]) {        // already present
        present = 1;
        break;
      }
    }
    if(present==0){                 // x is a new element
      if(uniques_size == nuniques){ // expand array if needed
        uniques_size *= 2;          // by doubling its capacity
        uniques = realloc(uniques, uniques_size * sizeof(double));
      }
      uniques[nuniques] = x;    // append x to the end
      nuniques++;
    }
  }

  *set_uniques = uniques;       // set the output array
  *set_nuniques = nuniques;     // set the output count
  return;
}
```

Describe a parallel version of this algorithm for a **Shared Memory Architecture** an appropriate programming paradigm we discussed in class.
Assess whether the parallel algorithm will yield much speedup over the serial version and if so, what kind of input data benefit most from the parallelism. If little speedup is to be gained, explain why and suggest alternatives.

Problem 4 (10 pts): A common serial algorithm to compute the minimum spanning tree of a graph takes

$$T_{serial} = 4N^2$$

operations to complete. A standard parallelization of this algorithm uses P processors and takes

$$T_{parallel} = 4\frac{N^2}{P} + 2N \log_2 P$$

operations to complete.
Fill in the table below to calculate various performance metrics for this parallel algorithm for different sizes of inputs. Give fractional values as **decimal numbers with 2 digits of accuracy.**

N	P	T_{serial}	$T_{parallel}$	Speedup	Efficiency	Cost
8	4					
16	4					
32	4					
32	8					

Construct an expression for the Parallel Overhead T_o of the parallel algorithm.
Also describe the Isoefficiency for this function to indicate how much the problem size needs to increase to keep efficiency constant for a larger number of processors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdarg.h>
#include <mpi.h>

int total_procs, proc_id, name_len;
char proc_name[256];

int main(int argc, char **argv){
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &proc_id);
  MPI_Comm_size (MPI_COMM_WORLD, &total_procs);

  int my_size;
  int *rbuf;

  if(proc_id == 0){
    int N = atoi(argv[1]);
    printf("Running with arg %d\n",N);
    int *sbuf = malloc(N * sizeof(int));
    int *all=NULL, i;
    for(int i=0; i<N; i++){
      sbuf[i] = (i+1) * (i+1);
    }

    my_size = N/total_procs;
    if(i < (N % total_procs)){
      my_size++;
    }
    rbuf = malloc(my_size * sizeof(int));

    for(int i=0; i<my_size; i++){
      rbuf[i] = sbuf[i];
    }

    int pos = my_size;
    for(int i=1; i<total_procs; i++){
      int size = N/total_procs;
      if(i < (N % total_procs)){
        size++;
      }
      MPI_Send(&size, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
      MPI_Send(sbuf+pos, size, MPI_INT, i, 1, MPI_COMM_WORLD);
      pos += size;
    }
    free(sbuf);
  }
  else{
    MPI_Recv(&my_size, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    rbuf = malloc(my_size * sizeof(int));
    MPI_Recv(rbuf, my_size, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  for(int i=0; i<my_size; i++){
    printf("P%2d: rbuf[%d] = %d\n",proc_id,i,rbuf[i]);
  }

  free(rbuf);
  MPI_Finalize();
  return 0;
}
```

Problem 5 (10 pts): The following MPI program demonstrates a certain communication pattern which is done inefficiently. Reorganize the code and replace inefficient calls with an equivalent an more efficient collective communication pattern.