3.26. Serial odd-even transposition sort of an n-element list can sort the list in considerably fewer than n phases. As an extreme example, if the input list is already sorted, the algorithm requires 0 phases.

a. Write a serial `Is_sorted` function that determines whether a list is sorted.

b. Modify the serial odd-even transposition sort program so that it checks whether the list is sorted after each phase.

c. If this program is tested on a random collection of n-element lists, roughly what fraction get improved performance by checking whether the list is sorted?

3.27. Find the speedups and efficiencies of the parallel odd-even sort. Does the program obtain linear speedups? Is it scalable? Is it strongly scalable? Is it weakly scalable?

3.28. Modify the parallel odd-even transposition sort so that the `Merge` functions simply swap array pointers after finding the smallest or largest elements. What effect does this change have on the overall run-time?

3.10 PROGRAMMING ASSIGNMENTS

3.1. Use MPI to implement the histogram program discussed in Section 2.7.1. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between −1 and 1;
   y = random double between −1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write an MPI program that uses a Monte Carlo method to estimate π. Process 0 should read in the total number of tosses and broadcast it to the other processes. Use `MPI_Reduce` to find the global sum of the local variable `number_in_circle`, and have process 0 print the result. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which `comm_sz` is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any `comm_sz`.

3.4. Write an MPI program that computes a global sum using a butterfly. First write your program for the special case in which `comm_sz` is a power of two. Can you modify your program so that it will handle any number of processes?

3.5. Implement matrix-vector multiplication using a block-column distribution of the matrix. You can have process 0 read in the matrix and simply use a loop of sends to distribute it among the processes. Assume the matrix is square of order n and that n is evenly divisible by `comm_sz`. You may want to look at the MPI function `MPI_Reduce_scatter`.

3.6. Implement matrix-vector multiplication using a block-submatrix distribution of the matrix. Assume that the vectors are distributed among the diagonal processes. Once again, you can have process 0 read in the matrix and aggregate the sub-matrices before sending them to the processes. Assume `comm_sz` is a perfect square and that $\sqrt{\texttt{comm\_sz}}$ evenly divides the order of the matrix.

3.7. A **ping-pong** is a communication in which two messages are sent, first from process A to process B (ping) and then from process B back to process A (pong). Timing blocks of repeated ping-pongs is a common way to estimate the cost of sending messages. Time a ping-pong program using the C `clock` function on your system. How long does the code have to run before `clock` gives a nonzero run-time? How do the times you got with the `clock` function compare to times taken with `MPI_Wtime`?

3.8. Parallel merge sort starts with $n/$`comm_sz` keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. To achieve this, it uses the same tree-structured communication that we used to implement a global sum. However, when a process receives another process' keys, it merges the new keys into its already sorted list of keys. Write a program that implements parallel mergesort. Process 0 should read in n and broadcast it to the other processes. Each process should use a random number generator to create a local list of `n/comm_sz ints`. Each process should then sort its local list, and