Parallel and Distributed Computing
Programming Assignment 1

Due Friday, February 7
(Note Change in Due Date)

For programming assignment 1, you should write two C programs. One should provide an estimate of the performance of "ping-pong" on the penguin cluster using various MPI implementations. The second should provide an estimate of the cost of a floating point operation.

1 Ping-Pong

A ping-pong program uses two processes and simply sends a message from (say) process 0 to process 1, and then process 1 sends the message back to process 0. We'll use this approach to evaluate the performance of message-passing using MPI.

In its simplest form, the ping-pong algorithm can be described by the following pseudo-code.

```
/* Use two processes */
if (my_rank == 0) {
   start timer;
   send message;
   receive message;
   stop timer;
   print time;
} else /* my_rank == 1 */ {
   receive message;
   send message;
}
```

Since the times are taken on a single node, ping-pong avoids the problems we might encounter with different clocks on different nodes.

Since the elapsed time will depend on the length of the message, your program should take timings for various size messages. At a *minimum* you should take timings for messages with lengths 0 bytes, 1024 bytes, 2048 bytes, 3072 bytes, . . . , 131,072 bytes.

After taking your timings you should use least squares to fit a line to your data. The reciprocal of the slope is often called the *bandwidth,* and the intercept the *latency.* (You can use a software package such as Matlab or Excel to do the least-squares calculations.)

The default timer provided by our MPI implementations (`MPI_Wtime()`) provides a resolution of 1 microsecond, which may not be much less than the cost of a single ping-pong for small messages. So instead of sending a single message, you should do something like this:

```
if (my_rank == 0) {
   start timer;
   for (i = 0; i < MESSAGE_COUNT; i++) {
      send message;
      receive message;
   }
   stop timer;
   print Average message time = elapsed time/(2*MESSAGE_COUNT);
} else {
   for (i = 0; i < MESSAGE_COUNT; i++) {
      receive message;
      send message;
   }
}
```

This formulation adds in the loop overhead, but it's still better than the single ping-pong. A `MESSAGE_COUNT` of 100 isn't too big.

2 Cost of Floating Point Operations

The performance of floating point operations is *highly* dependent on the code that's executing them. For example, if most of the operations are carried out on operands in level one cache, then the average performance will be far superior to that of code in which many operations are carried out on operands that need to be loaded from main memory. So we can't give a definitive answer such as "on machine X, the average runtime of a floating point operation is Y seconds."

Undeterred by such practical considerations, though, we'll persevere. For our purposes, let's define the cost of a floating point operation to be the average run time of one of the operations in the multiplication of two 1000×1000 matrices. That is, suppose we compute the elapsed time of the following calculation:

```
for (i = 0; i < 1000; i++)
   for (j = 0; j < 1000; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < 1000; k++)
         C[i][j] += A[i][k]*B[k][j];
   }
```

Then, since this code executes 1000^3 floating point multiplies and 1000^3 floating point adds, we'll define the cost of a floating point operation to be

$$\text{Cost of floating point operation} = (\text{Elapsed time})/(2 \times 10^9)$$

3 Caveats and Details

- The two programs should take no input. The ping-pong program should print out average times in seconds for the various messages sizes: 0 bytes, 1024 bytes, etc. The matrix-multiplication program should print out the average time in seconds per floating-point operation.

- In the ping-pong program be sure to initialize your buffers. There are implementations of MPI that crash when an uninitialized buffer is sent.

- For MPI, you should test messages using three different "implementations:" messages sent using Infiniband with MVAPICH, messages sent using ethernet with OpenMPI, and "shared memory" messages sent using OpenMPI. For the messages using Infiniband or ethernet you should use two *nodes* for your final data collection. If you use two nodes, all the MPI implementations will place one process on one node and one process on the other. So for the shared memory messages, you should use OpenMPI with a single node. Note that the same code that you used for Infiniband and ethernet should work without modification for shared memory messages: the implementation should detect that the two processes are on the same node and use its shared memory protocol.

  See the handout on the class website on the penguin cluster for more details on using the cluster.

- For MPI, use `MPI_Wtime()` for timing:

```
double start, finish, elapsed;
if (my_rank == 0) {
   start = MPI_Wtime();
   loop of pingpongs;
   finish = MPI_Wtime();
   average elapsed = (finish-start)/(2*number of iterations);
}
```

- Some implementations of MPI do some "on-the-fly" set up when messages are initially sent. These messages may cause your program to report somewhat slower times. This doesn't seem to be an issue on the penguin cluster, but if you're going to be running the program on other systems it may be useful to add 10-15 ping-pongs before starting the timed ping-pongs.

- Use the macro `GET_TIME()` defined in `timer.h` for timing matrix multiplication. It takes a double (*not* a pointer to a double) as an argument, and it returns the number of seconds since some time in the past as a double.

- It may happen that the runtime system on the cluster will give a segmentation fault if you try using a declaration such as

  ```
  double A[1000][1000];
  ```

  The matrix is too big. You're more likely to have success if you allocate the matrices from the heap:

  ```
  double* A;
  . . .
  A = malloc(1000*1000*sizeof(double));
  . . .
  free(A);
  ```

  If you do this, you'll need to take care of converting a subscript with two indexes into a single subscript. An easy way to do this is to use `A[i*1000 + j]` instead of `A[i][j]`.

4 Analysis

You should include an analysis of your results with your source code. The discussion should include estimates of the slope and intercept for the least-squares regression lines approximating your data for the MPI program.

Here are some other issues you should consider.

1. How well did the least squares regression line fit the data? What was the correlation coefficient?

2. Instead of using elapsed or wall-clock time, we might have used CPU time. Would this have made a difference to the times? Would it have given more accurate or less accurate times? Would it have had different effects on the reported times for the two different programs?

3. Was there much variation in the ping-pong times for messages of a fixed size? If so, can you offer a suggestion about why this might be the case? (You may need to generate more data to answer this question.)

4. Assuming your estimates are correct, what (if anything) can be said about the relative costs of communication and computation on the penguin cluster?

5 Coding and Debugging

When you write a parallel program, you should be extremely careful to follow all the good programming practices you learned in your previous classes — top-down design, modular code, incremental development. Parallel programs are far more complex than serial ones; so good program design is even more important now.

With all of our MPI implementations all of the processes can print to `stdout`. So the simplest approach to debugging is to add `printf` statements at critical points in the code. Note that in order to be sure you're seeing the output as it occurs, you should follow each debugging `printf` by a `fflush(stdout)`. Also, since multiple processes can produce output, it's essential that each statement be preceded by its process rank. For example, your debug output statements might have the form

```
printf("Proc %d > . . . (message) . . .\n", my_rank, . . .);
fflush(stdout);
```

Finally, note that all the processes are writing to a single shared buffer. So the order of the output from different processes is meaningless: only the order of a single process' output is meaningful.

6 Documentation

Follow the standard rules for documentation. Describe the purpose and algorithm, the parameters and variables, and the input and output of each routine. You should also include your analysis with the documentation.

7 Subversion

You should put copies of your source files and any makefiles in your `prog1` subversion directory by 11 am on Friday, February 7. (See the document `http://www.cs.usfca.edu/peter/cs625/svn.html` on the class website.) You should also turn in a print-out of your source to me by 2 pm on the 7th.

8 Grading

Correctness will be 60% of your grade. Does your program correctly ping-pong for the various required message sizes? Is the matrix-multiplication correct? Are the numbers the programs generate reasonable?

Static features will be 10% of your grade. Is the source code nicely formatted and easy to read? Is the source well-documented?

Your analysis will be 30% of your grade. Did you answer the required questions for the analysis? Does your data support your conclusions?

9 Collaboration

You may *discuss* all aspects of the program with your classmates. However, you should never show any of your code to another student, and you should never copy anyone else's code without an explicit acknowledgment.