Scientific Computing 9505
Assignment1: Intro to MPI
Due: 2 November

1. **MISD**: Generate an array of numbers (something interesting so you can tell if the moments are reasonable) and calculate the moments (up to the number of processors) as defined in class using a Multiple Instruction Single Data set paradigm: Generate the array of numbers on the master processor, send all the data to all the processors and have each processor calculate one of the moments. Output all the moments. Note that moment *k* of a set of *n* numbers is $M_k = \frac{1}{n}\sum_{i=1}^{n} x_i^k$.

2. **SIMD:** Generate an array of numbers on the master processor. Send a different part of the array to each processor and have each processor work out the 4th moment of its part of the array. Once done, collect the results as a sum on the master processor and output the results. You can use the dot product code we discussed in class, and posted on OWL as a starting point.

3. **Message Passing:** There were a number of upgrades for the dot product program that were suggested in lecture. We will implement some upgrades in this problem.
   a) The Boss node used more memory than the other nodes. We could fix this by reading segments of the file and then sending before reading the next segment. In this case the Boss node should work out the dot product for the *last* segment (not the first).
   b) Allow any size of array to work by having all the worker nodes work on equal sized segments and the Boss node to work on the remaining segment at the end (which might be a bit shorter than the others).

4. **Syncronization**: a) Implement the 1st "Controlling Synchronization" example from class. You will need to come up with your own implementation for the "waste_your_time" function (for example, you could just make it count to a million, or do something more interesting like compute digits of pi). Does your output look anything like the example? Do you think any of the comments from lecture on this example apply to your code? Why/Why not. b), c), d), e) Implement each of the variations suggested in lecture and describe what they do/do not do.

What to hand in for each problem: Do not just hand in a code and some output, provide some analysis of what you did. On the other hand, don't overdo it. This assignment is meant to be fairly short and straightforward, not something you need to write a huge amount about. A paragraph would probably be enough in most cases. E.g:
i) A very brief **written** description of what the code does.
ii) A printout of the computer code you wrote for the problem. The code should include appropriate comments in it, including a comment at the top describing what it does. You can also print out the code and add diagrammatic comments by hand (e.g. arrows to important parts, highlighting, etc, similar to what was done in the lecture notes). The code should also be appropriately indented.

iii) Sample input and output that demonstrates that your program works as advertised (a screen shot demonstrating your actual program running that includes the window/MAC header, and with the fonts large enough to be easily legible. If you are unsure of how to do this google "screenshot"). For the screen shots, the sample I/O should be short and only long enough to demonstrate that the code works. Longer I/O can be summarized. You should also describe why you picked a given input, for example: Does it test something interesting?
iv) Written comments (again ***brief*** and to the point) on things like whether your code worked as expected, answers/comments about anything the question asks about in the order in which the questions are asked and labeled so I can easily see where you are answering or commenting on anything. Do not just say things like "The code works", you need to persuade me that it works for instance by including things like test cases where the answer is known (where you also say where you got the "known result" from). An analysis of the results should be included such as, are the moments what one would expect (how accurate is the code, does this depend on the size of the array of numbers?), what order do the results come out in? (why?), how long did it take to run? (is this reasonable?) (Note: On the command line, you can use: `time <command>` , to obtain the length of time `<command>` takes to run. You may need to do this a couple of times to ensure you don't get something odd).

Make sure you make some effort to test the code in a reasonable limit (e.g. there would be little point to write a parallel program to work out the sum of 100 numbers. How many numbers before it becomes reasonable to use a parallel code?)

A Sample Solution similar to what is expected is provided on the next pages.

Sample Solution based on an example from another course:

Problem 1.6.

(i) The code for this example invites the user to enter a nonnegative number, and returns the square root of this number. Before the square root is calculated, we check that the number really is nonnegative through the assert statement.

(ii) Code for Problem 1.6:

```
#include <iostream>
#include <cassert>
#include <cmath>

/*********************************************************/
/* Example 1.6 from the text.                            */
/* This program computes the square root of a number 'a'.  */
/* The number 'a' is obtained from the user by asking for  */
/* it to be entered on the command line.  The number is    */
/* then checked to ensure it is positive and, if so, the   */
/* sqrt is then printed to the screen.                     */
/*********************************************************/
int main(int argc, char* argv[])
{
  double a;
```

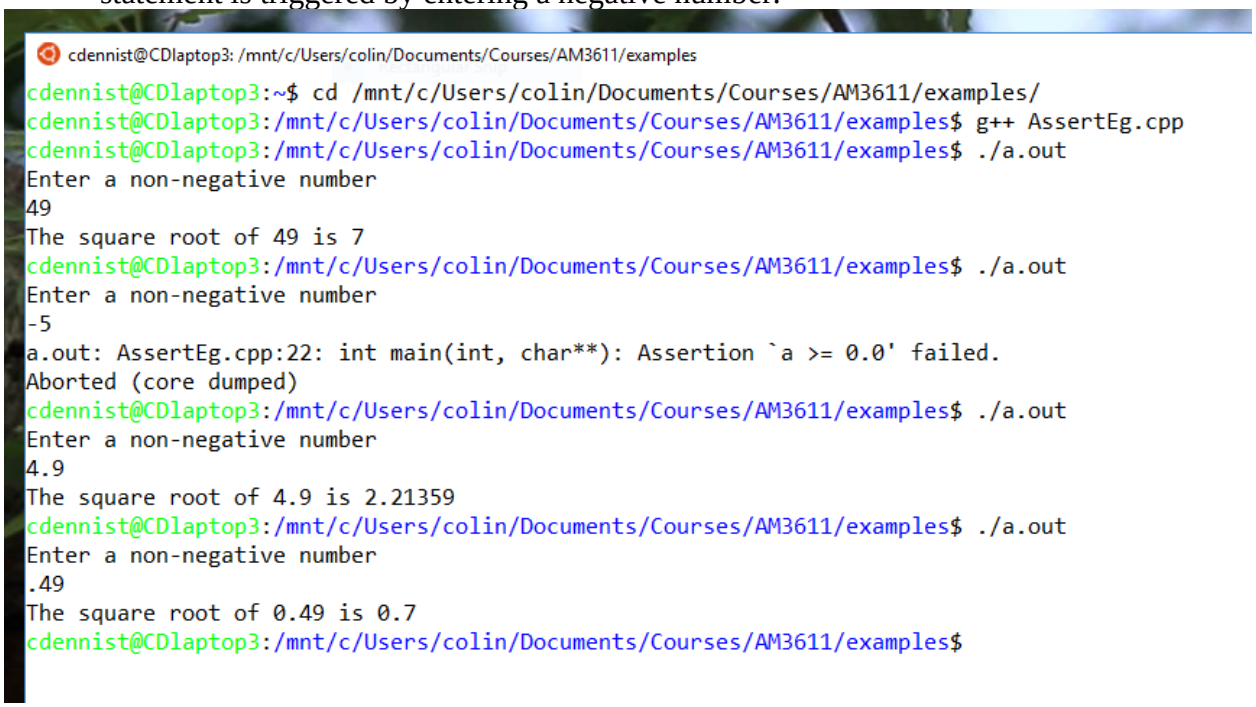
```
    // Get the number from the user
    std::cout << "Enter a non-negative number" << std::endl;
    std::cin >> a;

    // Check that a is positive
    assert(a >= 0.0);

    // Output the sqrt of 'a'
    std::cout << "The square root of "<< a;
    std::cout << " is " << sqrt(a) << "\n";

    return 0;
}
```

(iii) Sample input and output for Problem 1.6 is below and was obtained using the Windows snipping tool. We illustrate both the successful completion of the program where the square root is output and the unsuccessful completion where the assert statement is triggered by entering a negative number.



(iv) Comments: From (iii) we can see that the program correctly gave 7 as the square root of 49. When we incorrectly entered -5 when asked for a non-negative number the assert statement on line 22 of the program was triggered and the program terminated. To ensure the code also worked for floating point numbers we also did two additional tests, both of which gave the correct square root of the number based on comparison to a calculator. There is the caveat that the square root was only given to 6 digits. For the case where we took the square root of 4.9, the result was truncated to six digits and thus is not exact.

These examples illustrate the expected behaviour of the code, including the expected truncation of floating point numbers.