exercises corresponding to the first fifteen lectures

The exercises below are ordered chronologically. A good understanding of the lecture notes (what was written on the blackboard in class) should suffice to find the answers.
The programs we have seen in class are available on the class web site.

The first element in the double numbering refers to lecture. Exercises with zero number are meant as an addition to your culture and will not be collected.

1.0) Visit www.top500.org.

1.1) How many processors whose clock speed runs at 3.0GHz does one need to build a supercomputer which achieves a theoretical peak performance of at least 4 Tera Flops? Justify your answer.

1.2) Suppose we have a program where 2% of the operations must be executed sequentially. According to Amdahl's law, what is the maximum speedup which can be achieved using 64 processors? Assuming we have an unlimited number of processors, what is the maximal speedup possible?

2.1) Benchmarking of a program running on a 64-processor machine shows that 2% of the operations are done sequentially, i.e.: that 2% of the time only one single processor is working while the rest is idle. Use Gustafson's law to compute the scaled speedup.

2.2) Derive a formula for the number of links in a hypercube with $p = 2^k$ processors, for some positive number k.

3.1) Consider a network of 16 nodes, organized in a 4-by-4 mesh with connecting loops to give it the topology of a torus (or doughnut). Can you find a mapping of the nodes which give it the topology of a hypercube? If so, use 4 bits to assign labels to the nodes. If not, explain why.

3.2) In class we derived an omega network for eight processors (see figure 1.14 on page 21 of the textbook). Give an example of a configuration of the switches which is blocking, i.e.: a case for which the switch configurations prevent some nodes from communicating with each other.

4.0) Run the programs we have seen in class on the cluster.

4.1) Adjust hello world so that you type in your name and every node salutes you, using the name you typed in.

5.1) Modify the program we have seen in class to sum 100 numbers to work with a variable number of processors. Instead of 100, add the first n positive natural numbers where $n = 10p$.

5.2) Rewrite the program to sum 100 numbers using MPI_Send and MPI_Recv instead of MPI_Scatter and MPI_Gather.

5.3) Rewrite the program to square p numbers using MPI_Scatter and MPI_Gather.

5.4) Modify the program for a parallel sum replacing the MPI_Gather by MPI_Reduce to sum up the results of the partial sums.

6.1) Rewrite the `cost_scatter.c` (available from the course web site) using MPI_Send and MPI_Recv. Execute for #processors $= 2, 3, \ldots, 10$, and make a table with wall times. Are the times as bad as for the scatter?

6.2) The second program with fan-out broadcast does not do the same as the scatter because it sends the same block to all processors. Modify the fan-out broadcast program `cost_fan.c` so that both programs achieve the same effect. Check the wall times for eight processors.

6.3) Run the program `cost_fan.c` for a varying number of processors, e.g.: #processors $= 2, 3, \ldots, 8$, and compute start up time for communication via interpolation.

7.1) The program `cost_i1sr.c` shows experimentally that on our cluster in the time it takes to send a 2520-by-2520 matrix of doubles, one can do one million floating point additions. Modify this program, using nonblocking sends and recvs to measure how many floating point additions can be done in the time it takes to scatter a 2520-by-2520 matrix over 10 processors.

8.1) An image is represented by a matrix of values between 0 and 255, the (i, j)th element encoding the grayscale of the (i, j)th pixel. A reflection of the image along its diagonal swaps the grayscale for the (i, j)th pixel with the grayscale for the (j, i)th pixel. Describe a parallel algorithm to perform this reflection, distribute the data in such a way to minimize the communication overhead and to achieve an equal distribution of the number of swaps among the processors.

9.1) Download the MATLAB/Octave script `mtbf.m` to solve a simple instance of the mean time between failures problem using the Monte Carlo method. Translate `mtbf.m` into C and create a parallel version of the program. Use MPI_Walltime to measure the speedup. Do you get more accurate answers as you increase the number of simulations?

10.0) Visit `http://sprng.cs.fsu.edu/`.

10.1) We say that a is a primitive element mod m if $\#\{\ x \mid x = a^i \bmod m,\ i = 0, 1, \ldots, m-1\ \} = m-1$. If the multiplier a in the congruential generator $x_{n+1} = a * x_n \bmod m$ is a primitive element mod m, for a prime modulus m, what is the period of the numbers x_n? Does the period depend on the seed x_0? Justify your answers, either by mathematical proof or/and simulations.

10.2) Write an algorithm to compute, for a given m, the sequence of numbers a of primitive elements mod m. Implement the algorithm in C and create a parallel version of the program.

11.1) A continuous function f over an interval $[a, b]$ has a root in $[a, b]$ if $f(a)f(b) < 0$. The bisection algorithm to approximate a root of f in $[a, b]$ repeatedly bisects $[a, b]$, keeping the interval for which $f(a)f(b) < 0$. The algorithm stops after N bisections, or sooner when either $|b - a| < \epsilon$ or $|f(\frac{a+b}{2})| < \epsilon$, for $\epsilon > 0$.
Generalize this bisection method for p processors, keeping in mind that the computational cost is dominated by the number of function evaluations. Write this parallel algorithm in pseudo code.
Analyze the speedup, computation/communication ratio, and the scaled speedup: if we allow the same number N of iterations, how much more accurate will our approximation be?

11.2) Implement the parallel root finding method of exercise 11.1. Use the MPI_Walltime to measure the speedup for an increasing number of processors, $p = 2, 3, \ldots, 10$. Experiment with $\cos(x)$ over $[0, 3]$. How accurately can you compute $\frac{\pi}{2}$?

11.3) (a) Implement the recursive bisection algorithm to compute the sum of n doubles. You may assume that $n = 2^k$, for some $k > 0$. Illustrate the correctness of your implementation giving the output of summing the first n positive numbers.

(b) Develop a parallel implementation of this recursive bisection summation algorithm, using the fan out algorithm to distribute and to collect the results. Illustrate again the correctness in the same way as before. Execute the parallel algorithm on 8 processors and measure the wall time.
Can you take n large enough to notice a speedup?

12.1) Consider the linear search algorithm to find a given element x which occurs in an unsorted sequence. Write a parallel version of this linear search algorithm in pseudo code, assuming the sequence is already scattered evenly among the processors. Analyze the speedup, counting the number of comparisons in the best case, worst case, and average case. Examine the computation/communication ratio to decide whether and when a parallel linear search algorithm is worthwhile.

12.2) Implement a parallel linear search algorithm for an unsorted sequence of numbers. Test it by generating a sequence of n randomly generated doubles in $[0, 1]$ and picking one x of this sequence to search for. Take random choices for x and specific choices of x to simulate the worst case and best case. Use at least 10 processors and n large enough to get a realistic impression of the computational cost. Use the MPI_Walltime to investigate if the search can terminate p times faster.

12.3) Take the program `bucket_sort.c` available from the class web site and write a parallel implementation for it. To examine the communication overhead, report MPI_Walltime results on 5 and 10 processors, once with and once without taking the cost of the scattering of the data into account, for a sequence of at least one million random doubles. Discuss the results.

12.4) Consider a sequence of n numbers, sorted in increasing order. Binary search to locate an element in this sequence takes $O(\log_2(n))$ comparisons. Assuming the sequence is already distributed evenly among p processors, can one do better? Describe a parallel version of a parallel binary search in pseudo code. Analyze the speedup and computation/communication ratio.

12.5) Implement a parallel binary search algorithm for a sorted sequence of numbers. To avoid the communication overhead, let processor i (for i ranging between 0 and $p-1$) generate an increasing sequence of n evenly spaced numbers between i/p and $(i+1)/p$. Run the parallel binary search for 5 and 10 processors, with n at least one million and record the values returned by MPI_Walltime.

13.1) Extend the traprule function so that we can compute double integrals. Use the following prototype:

```
double traprule ( double (*f)( double x, double y ), double a1, double b1,
                  double (*a2)( double x ), double (*b2) ( double x ), int n );
/* applies the composite Trapezoidal rule to approximate the integral
 * of f(x,y) for x in [a1,b1], and y in [a2(x),b2(x)], using n+1 function
 * evaluations each time in the rule; use only for n > 0 */
```

Use it to compute the approximate the volume under the unit sphere in the first octant, i.e.: for $x \in [0, 1]$, and $y \in [0, \sqrt{1-x^2}]$. Write a parallel version of this program. Examine the speedup, executing the program for various number of processors.

13.2) An application of numerical integration is the computation of Fourier series approximation of a function. Over $[-1, +1]$, the Fourier series for a function $f(x)$ is the sum $a_0/2 + \sum_{k=1}^{\infty}(a_k \sin(k\pi t) + b_k \cos(k\pi t))$, where the coefficients are computed by the integrals $a_0 = \int_{-1}^{+1} f(t)dt$, $a_k = \int_{-1}^{+1} f(t)\sin(k\pi t)dt$, and $b_k = \int_{-1}^{+1} f(t)\cos(k\pi t)dt$. Write the pseudo code of a parallel algorithm to compute the first n integrals in the Fourier series. Analyze the speedup and computation/communication ratio.

13.3) Implement the parallel version of the algorithm of exercise 13.2, using the composite trapezoidal rule. Test your program for $f(x) = e^x$ (compare with Maple to ensure the correctness of the coefficients). Use MPI_Walltime to measure the speedup.

14.1) The program `nbody.c` shown in class to outline a very basic algorithm to simulate the n-body problem did not use any realistic data. The gravitational constant G is 6.67e-11 with units $\mathrm{m^3 s^{-2} kg^{-1}}$. To simulate the sun-earth-moon system, one would need to use the following data. The mass of the sun, earth, and moon are respectively 2.0e+30 kg, 6.0e+24 kg, and 7.35e+22 kg. The earth is about 1.5e+8 km away from the sun and the moon moon is about 384,400 km away from earth. To make a simulation numerically stable, one could scale as follows: take 1 for the mass of the sun and 1 for the distance between earth and the sun. Change the gravitational constant G accordingly and run a simulation for one year. Experiment with different time steps and judge whether the results are meaningful (after one month, the moon should have completed an orbit around the earth).

14.2) In class we stated (without proof) the following technical lemma about quadtrees for point sets: the depth of a quadtree is at most $\log_2(s/c) + 3/2$, where s is the length of one side of the square which contains all points of the set and where c is the smallest distance between any two points. Prove this.

14.3) Develop an algorithm to create an Octtree for a given list of coordinates in 3-space. Implement the algorithm in C and test is for lists of random points with coordinates distributed uniformly between -1 and $+1$. Run the algorithm many times and record the depth of the Octtree and the smallest distance between two points in the list. Based on these experimental results, try to formulate a general statement about the depth of the Octtree and the distance between the points in the list.

15.1) Consider the evaluation of a polynomial f of degree n given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_n)$ at some x, i.e.: compute $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Give the pseudo code of an algorithm to compute $f(x)$ by a p-stage pipeline. Illustrate the algorithm with a space-time diagram for $n = 32$ and $p = 4$. What is the speedup for this case?

15.2) Consider the evaluation of a polynomial f of degree n given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_n)$ at some x, using Horner's method, e.g., for $n = 4$: $f(x) = (((a_4x + a_3)x + a_2)x + a_1)x + a_0$. Give the pseudo code of an algorithm to compute $f(x)$ by a p-stage pipeline. Illustrate the algorithm with a space-time diagram for $n = 32$ and $p = 4$. What is the speedup for this case?

Homework collection is every Friday, at 1PM. Homework (unlike projects) must be completed individually. Feel free to contact me with questions about these assignments.