This assignment has three objectives: first, to become familiar with programming using the MPI (Message Passing Interface) library, second, to develop an understanding of static and dynamic load-balancing strategies, and finally, to observe the impact of communication overheads on overall parallel program performance.

You should use the Linux Beowulf cluster for all of your program development and debugging work. The assignment requires you to conduct measurements on both the Beowulf cluster and the Origin 2000 machine at NCSA.

**Particulars**

For this assignment you will parallelize a cellular automaton problem called WaTor using the MPI library. The class of problems known as cellular automata is characterized by a problem space consisting of a number of cells, each of which can take on one of a set of states. Each cell in the problem space interacts with neighboring cells following a set of rules to produce the next generation of cells in which the states have evolved using the rules. From a parallelism point of view, cellular automata are representative of a much broader class of problems, which include N-body interactions (e.g., interactions between drug molecules in a solvent, matter in space, etc.). All of these problems are characterized by the uneven distribution of work across the problem space, necessitating either static or dynamic load balancing.

The WaTor problem (sometimes also referred to as the “Sharks and Fishes” problem) models evolution over time of a two-dimensional toroidal (wrap-around) ocean inhabited by two species: fishes and sharks. The ocean is modeled as a set of grid points, with each point only being inhabited by one animal at a time. Fishes are assumed to swim around at random (eating abundant plankton which is not modeled in the problem). Sharks swim around looking for fish to eat. Sharks that have not eaten for some number of time steps will starve. The fishes and sharks of WaTor are capable of producing a single offspring every specified number of time steps if there is room in the world to support it.

More precisely, the sharks and fishes behave as described below:

- Each time step, a fish will move randomly to any of its four adjacent cells that is empty. If they have been alive long enough to reproduce, they leave behind a single offspring in the cell that they are vacating. If no empty cells exist around a fish, it simply stays put.
- Sharks first look for neighboring fish to eat. If more than one of the four neighboring cells is occupied by a fish, then the shark will choose one at random to eat. If there are no fish in neighboring cells, then the shark will move just like the fish do (to a random empty location). Like fishes, sharks also leave behind offspring if they are old enough to reproduce. If the shark has not eaten in a specified number of iterations, it dies.

Note that the age of both sharks and fishes is reset when they reproduce.

To help you get started, I have provided a sequential version of the WaTor program accessible on bm.scs.cs.nyu.edu at the following location: /home/vijayk/Homework3/saf.c.

Run saf without any arguments to see what command line arguments it expects: the X and Y dimensions of the ocean, the initial number of fishes and sharks, their breeding ages, the starvation threshold for sharks, and the number of timesteps to simulate. The program dumps out the state of the ocean every DUMP_INTERVAL timesteps (a constant defined in the program) into a file named ocean.data. In the
same directory, you will also find a simple Java program, \textit{WatorGUI}, which allows you to visualize the simulation. The Linux cluster does not have any Java binaries, so you will have to run the program on your own machine. Note also that the dump file can grow pretty large, so please run moderate sized simulations. The following is a relatively stable configuration, which you can use for testing purposes:

\texttt{> ./saf 128 128 300 30 4 12 8 200}

Given this background, the assignment consists of three parts (all parts are required):

1. \textbf{(6 points)} Parallelize the Wator program for execution on multiple processors in the Single Program Multiple Data (SPMD) style using MPI primitives for communication and synchronization. Specifically, perform the following steps:
   - Partition the ocean among the processors using a \textit{scatter decomposition} of blocks. Your partitioning scheme must be parameterizable, so that it is possible to experiment with different block sizes and different assignments of blocks to processors. In particular, you should \textit{not} assume that blocks are square.
     A simple way of doing this may be to take as arguments to your parallel program, the X and Y dimensions of a block, and then cyclically distribute the resulting blocks among processors. You can assume that the grid and block dimensions are powers of 2.
   - Allocate storage for the block(s) a processor will be responsible for, and randomly seed this subgrid with the portion of the sharks and fishes necessary. Note that each processor should have a different starting seed for its random number generator. Note also that you will need to allocate “ghost region” space along the grid boundaries to store cells that map into neighboring blocks.
   - The time step loop starts off with processors exchanging the boundary data for their assigned blocks with neighbors and using this data to fill out the ghost regions.
   - Each processor then calculates the movement of the fishes and sharks in its blocks. Note that conflicts can arise if two processors independently decide to move an animal into the same grid point.
   - Each processor exchanges the animals it wishes to move across block boundaries with its neighbors. Note that conflict resolution needs to happen here. For purposes of this lab, let us assume that one of the animals is thrown away. Note that the results of the conflict resolution need to be propagated back to the source processor so it can update its view of the world.
   - Repeat the previous two steps until the desired number of iterations has completed, \textit{or until all shark and fish die}. Note that the latter is not implemented in the sequential program, and requires a reduction operation in the parallel program.

Your program should behave correctly for any input and block size specification. Note that your parallel program will produce somewhat different ocean states than your sequential program because of the random selection of moves. However, these are only likely to cause local perturbations as opposed to global changes in simulation behavior.

You may want to use MPI support for communication data types and reduction operations to preserve the natural structure of the code (instead of flattening out data structures into a byte array sent by a message). You should try to minimize communication operations (by aggregating locally first) and overlap whatever remain as much as possible.

2. \textbf{(4 points)} For specific inputs, which will be posted on the mailing list in a few days, characterize the performance achieved by your parallel program for different block sizes on both the Linux Beowulf
cluster and the Origin 2000. For a given number of processors and a fixed set of inputs, identify the block size that yields the best overall speedup. Explain your observations in terms of the overheads of communication operations and load imbalance in your best performing partition.

3. (5 points) In Part 2 above, you will observe that a scatter decomposition (where multiple non-contiguous blocks are assigned to the same processor) would yield better overall performance. However, this improved performance comes at the cost of increased communication overheads because of the need to exchange block perimeter regions with neighboring processors.

An alternative load balancing scheme, which is effective in systems with slow-changing loads (e.g., a sparse ocean), is to periodically compute a near-optimal partitioning of the grid among processors based on an estimate of the work involved in each grid region. One example of such a technique, called *orthogonal recursive bisection* (ORB), divides up the grid into the same number of regions as the number of processors; however, regions although rectangular in nature can have widely differing dimensions.

A simplified version of the ORB strategy (with a central coordinator) works as below:

- At the end of every *k* iterations (*k* is a prespecified parameter), all the processors send their subgrid data to a coordinating processor.
- This coordinating processor runs the algorithm described below and parcels out new (balanced) regions to each processor, which continue working on it for the next *k* iterations.

The coordinating processor recursively computes a set of *bisectors* (alternating between the X and Y dimensions), resulting in the domain being represented as a (possibly unbalanced) quad tree whose leaves have roughly the same number of life forms. One way of computing the bisectors is to compute the median lines through each region of the grid (starting from the entire grid). See the following figure for an example.

Modify your program of Part 1 so that each processor receives its work assignment from a coordinating processor that runs the ORB algorithm. For a set of inputs that will be posted to the mailing list in a few days and a given number of processors, compare the performance of this scheme against that of the scatter decomposition scheme developed earlier on both the Beowulf cluster and the Origin 2000. Your measurements should separate out the costs of computing the new region assignment from the time spent by processors working on the loop iterations.
Guidelines

You are expected to hand in a write-up that should contain the speedup measurements, and answers to the analysis questions asked above. Your answers should be detailed enough to convince me that you understand the reasons for the observed performance. Please also include a code listing of your programs.

One final word of advice: you may want to start off with a relatively straightforward implementation of your parallel program first before you put in any optimizations. This assignment admits several optimizations, some of which are fairly tricky to get right.