CIS 372: Fall 2003
Fall 2003
Individual Programming Assignment 3

Due Date: Thursday, November 13, 2003

1 Objectives

The objective of this assignment is to gain experience in parallel programming with different data distributions, and to observe the resulting effect on performance.

2 The Problem: The Game of Life

The game of life is a simple cellular automaton where the world is a 2D grid of cells which have two states: alive or dead. At each iteration, the new state of a cell is determined by the state of its neighbors at the previous iteration. This includes both the nearest neighbors and the diagonal neighbors, i.e., like a 9 point stencil without using the value of the cell itself as input. The rules for the evolution of the system are:

- If a cell has exactly two alive neighbors, it maintains state.
- If a cell has exactly three alive neighbors, it is alive.
- Otherwise, the cell is dead.

Your code will need to:

1. Initialize the gameboard
2. Start loop:
3. Print the gameboard (temporarily for correctness checking)
4. Calculate number of live neighbors
5. If (live neighbors = 3) then live
6. If (live neighbors < 2) or (live neighbors > 3) then die
7. End loop
8. Print final gameboard (again temporarily for correctness checking)

3 The Assignment

3.1 The Basic Sequential Program:

Details of each part follow:

1. **Initialization:** Use array syntax to initialize a board (an n x n INTEGER array with value 1 for a live cell and 0 for a dead one) with the following pattern: let n = 8, and set the row and column nearest the center (n/2) to be alive with all other cells dead. For n = 8, that means that the live cells will be all of row 4 and all of column 4.
2. **Print board:** The board can be printed as a series of bitmaps which can be animated using xv, by having the board be printed in pgm format, with which you are already familiar. You may use other formats if you like.

3. **Update:** Declare a count array the same size as the board (n x n) to contain the number of live neighbors for each point. Include all nearest neighbors, including diagonal neighbors.

4. **Compilation:** Compile your code with board size n=8 for testing for 10 iterations of evolution, to produce an executable life program.

5. **Test:** Run the code on a single processor for 10 iterations to check that it produces a set of correct state configurations. Use xv to view the resulting arrays to check that your code is working. After checking correctness, you can comment out the print statements inside the loop and keep the final print of the final board.

3.2 Parallel Versions:

Initial Parallel Version with Block Row-wise Data Distribution: For these parallel versions, you may assume that the number of elements in a row of the square matrix is evenly divisible by the number of processes. Add MPI commands into your sequential code to create a parallel program that distributes the board and count matrices equally in a (ROW-BLOCK) row-wise block data distribution where each process receives and performs computations on a set of consecutive rows of the matrices. Process 0 should print the final board matrix and compare with the results of the sequential program to ensure that correctness is achieved in the parallel version. Test this version of the program on 2, 4, and 8 processes.
Implementing Additional Data Distributions: Copy your parallel version of the program and create separate parallel versions that perform data distribution in the following ways:

- ROW-CYCLIC: row-wise cyclic where each row is distributed as a whole row to a given process, and each process is assigned a row in a round robin fashion
- COL-BLOCK: column-wise block distribution where each process receives a set of consecutive columns of the matrix
- CYCLIC-CYCLIC: cyclic-cyclic where each process receives a single element of each row, in a round robin fashion, continuing to the next row in the same round robin fashion
- BLOCK-BLOCK: checkerboard where each matrix is partitioned such that each process receives a single submatrix of the board and count matrices, in a checkerboard fashion.

You should have 1 sequential version and 5 parallel versions of your program. You may use any MPI commands that you see appropriate to accomplish these data distributions. I encourage you to try some of the collective communications to ease your burden! You could also use a topology for the checkerboard version.
Performance Runs: Increase the size of your arrays to n = 2048. Compile and run both the sequential and the parallel programs and check correctness one more time by diff'ing the final files. After you are convinced that your parallel versions all work correctly, by checking the printed final matrices against each other, then comment out the printing part of your programs. If you have not already done so, add timing into the program that includes the data distribution, computation, and gathering phases, to be reported as separate timings and then a total time that includes all three phases.

Run each version with n = 2048 with no printing for 1, 4, 8, and 16 processes. Graph your total timings all on a single graph that includes one line for each of the 6 program versions, where each line contains data points for 4 different numbers of processes. Graph the computation only timings in the same way as another single graph. Graph the distribution only timinings in the same way as another single graph.

4 Experimental Report

Your experimental report should consist of the following sections in this order. It is strongly recommended that you type your report using a word processor rather than handing in a hand-written report.

1. **Cover Page:** Title, author, course number and semester, date.

2. **Project Summary and Hypothesis:** First, state and justify your hypothesis. Which data distribution did you expect to run the best, the worst? Why? Explain your hypothesis in the context of a large number of processes and a small number of processes, and a large matrix versus a small matrix.
3. **Collected Timing Data.** This section includes the graphs of the data collected from the performance runs.
4. **Analysis and Conclusions.** In English, explain your timing results. Describe your observations concerning the different data distribution methods, and how it compares to your hypothesis. Explain why you believe the timings are different or similar between the methods. How does it compare to your hypothesis? This section also includes a discussion of what you learned from the various aspects of the lab. Discuss possible reasons for inconsistencies or discrepancies in your data versus what you would have expected to happen.
5. **General Remarks:** Any comments on the lab itself that you would like to make in terms of suggestions for improvement or problems you experienced.
6. **Appendix:** Your code for each of the parallel versions. You do not need to include your sequential code. Be sure the label each version clearly.

Please staple all parts of your lab together, and label each piece. Be prepared to discuss your results in class.

5 Criteria for Evaluation

Your lab will be evaluated according to the following criteria:

1. 10 pts: Sequential game of life
2. 11 pts: Parallel version 1: row-block
3. 11 pts: Parallel version 2: row-cyclic
4. 11 pts: Parallel version 3: column-block
5. 11 pts: Parallel version 4: cyclic-cyclic
6. 11 pts: Parallel version 5: block-block
7. 10 pts: 3 Graphs of timings
8. 25 pts: Experimental report
   (a) 2 pts: cover page
   (b) 10 pts: hypothesis and project summary
   (c) 13 pts: analysis of timing results (includes points for code in appendix)

6 Extra Credit

Modify your program to work correctly when the number of elements in a row of the square matrix is not evenly divisible by the number of processes. Show that your program works correctly by running each parallel version with a matrix size that fits this description. Hand in the code and the output of the runs.

7 Notes on Multidimensional Arrays

To be able to access consecutives rows of a 2-dimensional array in C, you must allocate space for the entire matrix together with one malloc or statically. Otherwise, it is possible that separate malloc's get memory allocated from nonconsecutive locations.