CSc 422 — Homework 4, Spring 2005

Due Tuesday, April 26

This assignment is again worth 40 points. The first problem is worth 8 points; the programs are worth 16 points each. For the programs, use MPD, Java with sockets and/or RMI, or C with the MPI library.

Please hand in your answer to problem 1, paper copies of your programs for problems 2 and 3, and output from your programs. Also submit your programs electronically as described at the end of the assignment.

1. **Stable Marriage Problem**, Exercise 7.15, part (a). Use the asynchronous message passing notation employed in class and defined in Chapter 7 (channels, send, receive, and empty). I strongly suggest that you work out a few examples to get a feel for the problem before attempting to write a program. Every process should terminate knowing who its partner is. You may use a coordinator process to detect termination.

2. **Rock/Scissors/Paper.** Write a distributed program to simulate a three-person rock/scissors/paper game. Each player randomly chooses one of rock, scissors, or paper. Then the players compare their choices to see who "won." Rock smashes scissors, scissors cut paper, and paper covers rock. Award a player 2 points if it beats both the others; award two players 1 point each if they both beat the third; otherwise award no points. Then the players play another game.

Use one process for each player. The players must interact directly with each other. Do *not* use an additional coordinator process. There should be one command-line argument, `numGames`, the number of games to play. Print a trace of the results of each game as the program executes. At the end print the total points won by each player. Turn in output for 10 games and for 30 games.

3. **Eight Queens Problem.** Develop a distributed program to find all 92 solutions to the 8-queens problem. The goal of the problem is to place 8 queens on a chessboard in such a way that no queen can attack any other. This means that a solution cannot have two queens in the same row or column or along the same diagonal.

Your program should use the manager/workers paradigm (distributed bag of tasks) described in Section 9.1 of the textbook. There should be one command-line argument: `numWorkers`, the number of worker processes. The manager holds a bag of tasks. Each task specifies one possible placement of queens on a chessboard. Each worker repeatedly requests a task from the manager, determines whether the specified placement is a solution, and if so, tells the manager. The manager prints solutions either as they are found or at the end of the program.

A chess board has 8 rows and 8 columns. Hence, there are 64x63x62x61x60x59x58x57 different board configurations in which queens are on different squares. However, many of these are easy to rule out because a solution cannot have two queens in the same row or column. In particular, suppose we represent a board configuration as an 8-tuple (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8). Each v_i is an integer between 1 and 8. The subscript i indicates a column, and the value of v_i indicates which row in column i a queen occupies. If the values are unique—i.e., the values are a permutation of the integers from 1 to 8—then the tuple specifies a board configuration in which queens are in different rows and columns.

With this representation, there are 8! (40320) tuples that do not have two queens in the same row or column. One such tuple is (1,2,3,4,5,6,7,8), which has queens in row 1 of column 1, row 2 of column 2, row 3 of column 3, and so on. Another such tuple is (3,4,8,1,2,7,6,5), which has queens in row 3 of column 1, row 4 of column 2, row 8 of column 3, and so on.

Each task should be a tuple. You should have the manager generate them as needed—and need to figure out how to do so! It is easiest to start with (1,2,3,4,5,6,7,8), then (1,2,3,4,5,6,8,7), then (1,2,3,4,5,7,6,8), and so on.

To summarize, the manager generates 40,320 tasks one at a time as needed by the workers. The workers analyze tasks and should find 92 unique solutions. Run your program with 4 workers and turn in your output. Print each solution as a tuple, with one tuple per line.

Run your program on Lectura and/or Parallel. However, it must be written so that each component—the manager and each worker—*could* execute on a different machine. In particular, each component should be supported by a different Unix process. This is automatic with MPI, supported in MPD by virtual machines, and supported in Java by starting components individually and having them communicate using sockets or RMI.

Electronic Turnin. Use the `turnin` program on Lectura to turn in your programs. The assignment names are `hw4.rock` and `hw4.queens`. Use whatever file names you wish for your programs. Either put a "usage" comment at the top of each program, or turn in README files explaining how to compile and execute the programs.