CS484: Parallel Programming - Spring 2017
MP2
Due Date : Monday, March 20, 11:59PM
Submission Method : SVN

March 4, 2017

1 Assignment

1.1 Queue Implementation

Implement a lock-free circular queue using c++11 atomics, that can support multiple producers and consumers. Compare the performance of your implementation against an equivalent version that uses locks for synchronisation. You will be using the c++ interface to pthreads for this assignment.

The queue should support the following functions :

- Insert an element
- Delete an element
- Check if the queue is full
- Check if the queue is empty

Answer the following questions :

- What are the atomics you used for your implementation and why?
- Do the following experiments with your implementation :
  - For a fixed number of insertions and deletions, measure the performance of the two implementations for varying number of threads
  - For a fixed number of threads, measure the performance of the two implementations by varying the total number of insertions and deletions
- Tabulate your results and plot graphs for both experiments. Clearly indicate the parameters, i.e total number of operations (insertions and deletions) and the number of threads. You have to perform all your experiments on Taub.

We have provided you with starter code for both versions, you can find it under the folder *queue* in the MP2_new directory once you extract the tar file. Here is a good reference for multi-producer, multi-consumer queues with pseudo code `https://www.research.ibm.com/people/m/michael/podc-1996.pdf`

Here is a link to the list of c/c++11 atomics, that might be useful to you.

`http://en.cppreference.com/w/cpp/atomic`

1.2 Parallelizing WaTor

For this MP, you will be using MPI to implement a parallel WaTor simulation. WaTor is a population dynamics simulation implemented on a 2D toroidal (wrap-around) grid; for more information, see here.

We have provided you with a reference sequential implementation in "wator.cpp", under the folder *wator* which includes support functions. In *main()*, there are calls to *wator()* and *wator_parallel()*. *wator()* runs the sequential version of the Wator simulation, and is fully-functional. *wator_parallel()* contains the same code as *wator()*, but the calls to *update_sharks()* and *update_fish()* are replaced by *update_sharks_in_parallel()* and *update_fish_in_parallel()* respectively.

Your task is to write the program for *update_sharks_in_parallel()* and *update_fish_in_parallel()*, using the sequential versions of the methods as references. You should make use of MPI's message passing features to add parallelism to the update methods. The sequential program discretizes the domain into cells, which contain sharks and fish. The locations of sharks and fish are updated in a doubly nested loop based on the contents of its neighboring cells. For the parallel version, you can divide the domain into blocks/tiles with each tile consisting of a number of rows and columns. You can assign a tile to an MPI process. During an update phase, the elements of a tile will need rows and columns from its four nearest neighbors - north, south, east and west. For your convenience, you can assume the domain is toroidal where the cells wrap around, i.e the northern neighbor of row 0, lies on row N-1, for an NXN domain. Similarly, the western neighbor of column 0, lies on column N-1.

You have to implement the following versions :

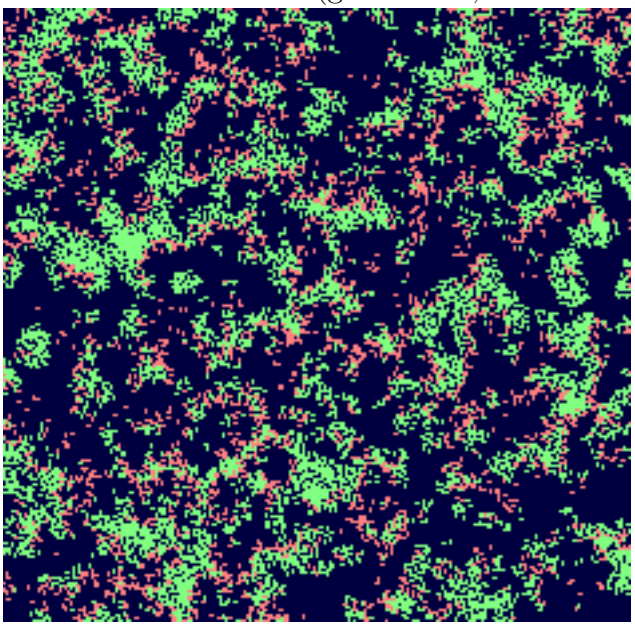
- Perform nearest neighbor exchange using blocking send/receive, preferrably MPI_SendRecv (single call that exchanges messages between a communicating pair).
- Use non-blocking MPI calls (MPI_Irecv and MPI_Isend) for nearest neighbor exchange

For each version, do strong scaling experiments by changing the number of MPI processes, fixing the size of the mesh (NXN) and the number of iterations. Measure the time taken for each experiment and plot a graph using the sequential time as baseline. Explain your parallelization strategy and compare it with the results you obtained.

Verify your implementation is correct by comparing the number of fish and sharks per chronon to the reference implementation (the algorithm is completely deterministic). Test your code locally first, then on Taub using a batch script (please avoid using Taub's login nodes).

It should not be necessary to edit any methods other than *update_sharks_in_parallel()*, *update_fish_in_parallel()*, and *main()*, though you are free to edit the rest of wator.cpp as you

Figure 1: Section of a Wator world (green=fish, red=shark, blue-water)



see fit. You will also need to add includes for MPI. Feel free to edit any of the constants at the top of wator.cpp to vary the parameters of your world. You may also find it useful to comment / uncomment the *#define ASCII_ART* line at the top of the file to toggle printing a snippet of the Wator world to your terminal.

1.3 Submission

Submission is via svn. Create a folder in your svn account named "MP2". Please have separate folders for each section - list implementation and Wator. The folders should contain all the files necessary for executing your code, **including makefiles and instructions for compiling / running**. You should also submit a neatly typed report, preferrably in pdf, which contains the graphs and analyses of your results.