15-418/618, Spring 2018
Assignment 4
GraphRats: MPI Edition

Assigned:	Wed., March 7
Due:	Wed., Mar. 28, 11:59 pm
Last day to handin:	Sat., Mar. 31

1 Overview

Before you begin, please take the time to review the course policy on academic integrity at:

http://www.cs.cmu.edu/~418/academicintegrity.html

Download the Assignment 4 starter code from the course Github using:

```
linux> git clone https://github.com/cmu15418/asst4-s18.git
```

In order to add support for MPI compilation for the GHC machines, do one of the following:

- Add the following line to your file `~/.cshrc`:

  ```
  setenv PATH $PATH\:/usr/lib64/openmpi/bin
  ```

- Add the following line to your file `~/.bashrc`:

```
export PATH=$PATH:/usr/lib64/openmpi/bin
```

Assignment Objectives

In this assignment, you will explore the use of the MPI standard to implement a program consisting of a number of independent processes that communicate and coordinate with one another via message passing. The application is typical of the *bulk synchronous* execution model seen in many scientific applications. Although the application is the same as you had in Assignment 3, you will find that your implementation is very different. OpenMP provides a data-parallel programming model, where the program consists of sequence of steps, each of which performs many operations in parallel. By contrast, an MPI program describes the behavior of an autonomous process that periodically communicates with other processes running the same code.

Machines

The MPI (for "Message-Passing Interface") standard provides a way to write parallel programs that can run on collections of machines that communicate with one another via message passing. This approach has the advantage that it can scale to very large machines, with 1000 or more processors. For this lab, you will run on single, multicore processors, but using message passing, rather than any form of shared memory communication or synchronization.

You can test and evaluate your programs on any multicore processor, including the GHC machines. For performance evaluation, you will run your programs on the Latedays cluster. We recently discovered significant performance among the processors in this cluster, especially when running parallel applications. To get more predictable measurements, we have identified eight of the nodes as having comparable performance to one another. We refer to these as the "normal" nodes. Other nodes are consistently faster or slower.

To ensure that your program only runs on one of the normal nodes, we have set up a special queue. Submit your job via the command

```
linux> qsub -q timer latedays.sh
```

These nodes will be used for performance measurement during grading.

Resources

There is a lot of information online about MPI. Some resources we have found useful include:

- General MPI Tutorial
- Longer MPI Tutorial from Lawrence Livermore National Laboratories
- Official documentation on OpenMPI v1.6, the version that runs on the latedays machines

2 Application

Dr. Roland Dent, Director of the world-famous GraphRats project was quite excited to find that million-rat simulations are possible using a well-optimized simulator running on a multicore processor. But, he dreams of more. “There are over seven billion rats in the world. Shouldn’t we be able to simulate billions of rats?” You have convinced him that such large simulations would require much more computing power, beyond what shared-memory systems can provide. It might be possible on a modern supercomputer, with ten thousand or more nodes that communicate by message passing.

As a feasibility study, you propose implementing an MPI version of the GraphRats simulator runnnng on a single, multicore machine, but using only message passing to communicate and coordinate among the machine’s cores. Your idea is to partition the graph into separate *regions*, mapping each region onto a separate process. Each process will keep track of the rats within its assigned region, computing the new states of all of these rats. It will communicate with processes holding adjacent regions, both to share the node states along the boundaries, and to pass along rats as they move from one region to another.

In order to make the approach work, you suggest the following restrictions on the graphs and initial conditions:

- Graphs must have some locality. You propose allowing only tiled and grid graphs.
- Rats must have a more even initial distribution. You propose allowing only the uniform and diagonal distributions.

Model Parameters

As before, each graph consists of N nodes with a set of edges directed and symmetric edges E indicating which nodes are adjacent. All of the graphs we use are based on a grid of of $k \times k$ nodes. Some of the graphs have additional edges beyond those comprising the grid connections, but there is an underlying *tile limit* t such that the graph can be partitioned into blocks, each with t rows and t columns. Arbitrary edges may exist within a tile, but the only edges between tiles are the regular grid connections at their boundaries. The graph file format has been extended to include the tile limit in the file header. The starter code includes the field `tile_max` as part of the struct `graph_t`. You can also see the granularity provided by the tiles by running and visualizing the simulations (try running `make demo7`).

All other aspects of the graph (rats, reward functions, update modes) are the same as in Assignment 3. Indeed, you can base your sequential version on the sequential version you wrote for Assignment 3 with only minor changes.

The simulator has similar options as before. The only difference is that rather than specifying the number of OMP threads, you specify the number of MPI processes. Its usage is as follows:

```
linux> ./crun -h
Usage: ./crun -g GFILE -r RFILE [-n STEPS] [-s SEED] [-u (r|b|s)] [-q] [-i INT]
   -h        Print this message
   -g GFILE  Graph file
   -r RFILE  Initial rat position file
   -n STEPS  Number of simulation steps
   -s SEED   Initial RNG seed
   -u UPDT   Update mode:
             s: Synchronous.  Compute all new states and then update all
             r: Rat order.    Compute update each rat state in sequence
             b: Batch.        Repeatedly compute states for small batches
                              of rats and then update
   -q        Operate in quiet mode.  Do not generate simulation results
   -i INT    Display update interval
```

As before, you can use the Python program `grun.py` to visualize the simulation results.

Regression Testing

The provided program `regress.py` has similar options before, except that you specify a number of MPI processes rather than OMP threads. Its usage is as follows:

```
linux> ./regress.py -h
Usage: ./regress.py [-h] [-c] [-p PROCS]
    -h       Print this message
    -c       Clear expected result cache
    -p PROCS Specify number of MPI processes
       If > 1, will run crun-mpi.  Else will run crun
    -a       Run ALL tests, including for big graphs
```

Performance Evaluation (80 points)

The provided program `benchmark.py` runs simulations of four different combinations of graph and initial rat position. It aggregates the performance by computing the geometric mean of the four MRPS measurements. By default, the program runs simulations with one and twelve processes in both batch and synchronous modes. The full set of options is as follows:

```
linux> ./benchmark.py [-h] [-a (x|o|n)] [-s SCALE] [-u ULIST] [-f OUTF][-c] [-p PROCS]
    -h             Print this message
    -a (x|o|n)     Set processor affinity: x=None, o=Old flags, n=New flags
    -s SCALE       Reduce number of steps in each benchmark by specified factor
    -u ULIST       Specify update modes(s):
       r: rat order
       s: synchronous
       b: batch
    -f OUTF        Create output file recording measurements
         If file name contains field of form XX..X,
         will replace with ID having that many digits
    -c             Compare simulator output to recorded result
    -p PROCS       Specify upper limit on number of MPI processes
       If > 1, will run crun-mpi.  Else will run crun
```

Here's a brief description of the options:

`-a (x|o|n)` Set processor affinity. This can be important to maximize performance. The options are: do nothing (`x`), the default; use directives suitable for the older installation of MPI found on the Latedays machines (`o`); or use directives suitable for the newer installation of MPI found on the GHC machines (`n`).

`-s` S Reduce the number of steps for each benchmark simulation by a factor of S. For example, with $S = 10$, a simulation that normally would run for 1000 steps will only run for 100. This will let you test your program without requiring a full length run of the benchmark. Generally, the measured performance (in MRPS) will be lower than you would get by running the full simulation.

`-s` $m_1 : m_2 : \cdots$ Run the simulator in the modes specified as a colon-separated list. For example the default is `b:s`, running in batch and synchronous modes.

`-p` P Run with a maximum of P MPI processes. Benchmarks that call for more will instead be run with just P processes. Single-process benchmarks run the program `crun-seq`. Multi-process benchmarks run the program `crun-mpi`.

`-f` *FILE* Generate an output report in file *FILE*, in addition to printing it on standard output. As a special feature, if the file name contains a string of the form `XXX`···`XX`, then the generated file will be named by replacing these characters by randomly generated digits. This provides an easy way to do multiple runs of the simulation to compensate for statistical variations in the program performance.

`-c` Check correctness of the benchmarks. The normal benchmarking does not check that your simulation produces the correct results. With this option enabled, it will compare the state at the beginning and the end of the simulation with ones produced by the reference simulator and stored as a file in the `capture` subdirectory. This feature only works under the following conditions:

- Your program does not write any additional information on standard output.
- You use a scaling factor of 1 (the default), 10, or 100.
- You only run in batch and/or synchronous mode.

For each of the two modes (batch and synchronous), your performance score will be a function of three values:

- The mean speed (in MRPS) of the single-process performance
- The mean speed (in MRPS) of the twelve-process performance
- The speedup that the latter number provides over the former

As a special case, if your single-process performance exceeds the target performance, then the target performance will be used as the denominator in the speedup calculation. This means that you don't need to artificially slow down a fast, single-process version in order to maximize the speedup measurement.

The following table provides performance targets for MRPS and for speedup. These thresholds are set at around 90% of what the reference solution achieves.

Mode	1-process MRPS	12-process MRPS	Speedup
batch	25	87.5	3.5
synchronous	35	112	3.2
Points	17	17	6

You will get full credit for each of these if your performance meets the target and partial credit if it is at least within a factor of 0.5×. As shown in the table, the MRPS measurements each count for 17 points and the speedup measurements each count for 6 points, yielding a maximum possible performance score of 80 points. The `benchmark.py` program will compute this score for you. Your total score will be rounded up to the next integer.

3 Some Advice

Important Requirements

The following are some aspects of the assignment that you should keep in mind:

- You may only use MPI library routines for communicating and coordinating between the MPI processes. You cannot use any form of shared-memory parallelism. The idea is to develop a program that could ultimately be deployed on a large, message-passing system.
- You must use spatial partitioning in your solution. We want you to get experience with that style of application.
- Although performance will be measured with just 1 or 12 processes, your program should be able to execute with any number of processes, even to the point where some of the processes will do nothing. For example, some of the graphs in the regression set have only 4 nodes, and you should be able to pass regression testing with 12 processes.
- You can assume that only uniform and diagonal initial rat states will be measured. You can optimize your performance for just the benchmark graphs and initial rat states provided. That might influence how you partition the graphs into regions. You need not do anything more exotic, in terms of partitioning, than is required to meet the benchmark targets.
- You are free to add other header and code files and to modify the make file. You can switch over to C++ (or Fortran) if you like.
- You can use any kind of code, including calls to standard libraries, as long as it is *platform independent*. You *may not* use any constructs that make particular assumptions about the machine instruction set, such as embedded assembly or calls to an intrinsics library. (The exception to this being the code in `cycletimer.c`.)
- You may not include code generated by other parallel-programming frameworks, such as ISPC, OpenMP, PThreads, etc..

Useful Parts of MPI

The MPI standard is large and complex. You only need to use a core subset of its features. Ideas that are especially useful for this assignment include:

- Using synchronous and asynchronous send and receive constructs for point-to-point communications. Asynchronous communication is preferred, because it allows the processes to operate in a more loosely coupled manner. When exchanging data with adjacent regions, it works well to have a process first initiate all of its send and receive operations and then wait for them to complete.
- Using the probe operation to determine the size of an incoming message. This is useful when sending variable length buffers of rats between processes.
- Using broadcast to send copies of the graph and initial rat states from process 0 (the master) to the others at the beginning of the simulation.

How to Optimize the Program

- You will find that the types of optimizations you did for Assignment 3 will be useful in this assignment, although some of your design choices and tuning parameters will vary, due to the different nature of the benchmark graphs and the different programming model.
- To make things simpler, consider storing a complete representation of the graph and the rat information for each process. It will use more space than is necessary, but it will allow you to have a univeral numbering scheme for nodes, edges, and rats. It also will not harm the performance of your program—cache behavior depends on how much memory actually gets used rather than on how much has been allocated.
- You will find that you need to represent different forms of sets for this assignment, e.g., the set of all rats within a particular slice. There are several common ways to do this:
  - As a *bit vector*, where bit position i is set to 1 if i is in the set and to 0 if it is not. Although it is possible to pack multiple bits into a word, a simple approach is to allocate an array of type `unsigned char` and just use one bit per byte as the flag.
  - As a *list*, consisting of all of the members of the set. It might be worthwhile to sort this list to make stepping through the elements more cache friendly, but you must consider the cost of performing this sort.
  - As a hash table. These can be problematic for very large sets due to poor cache behavior and unpredictable branches.

  It is also possible to have hybrids, maintaining multiple representations together. There are tradeoffs in which operations can be performed quickly with the different representations, including the cache behavior. You will want to experiment with different implementations.

4 Your Report (20 points)

Your report should provide a concise, but complete description of the thought process that went into designing your program and how it evolved over time based on your experiments.

Your report should include the following items:

1. A detailed discussion of the design and rationale behind your approach to improving the sequential performance. This should include reporting on any experiments you conducted to evaluate possible design decisions.

2. A detailed discussion of the design and rationale behind your approach to parallelizing the algorithm. Specifically try to address the following questions:

   - What approaches did you take to parallelize the different parts of the program?
   - How did you maximize the decoupling of processes to avoid waiting for messages from each other.
   - How successful were you in getting speedups from different parts of the program? (These should be backed by experimental measurements.)
   - How did the performance scale as you went from 1 to 12 processes? What were the relative speedups of different parts of your program?
   - How did the graph structure and the initial rat positions affect your ability to exploit parallelism? What steps did you take to work around any performance limitations?
   - Were there any techniques that you tried but found ineffective?

5 Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting your entire directory tree.

1. Your code

   (a) **If you are working with a partner, form a group on Autolab.** Do this before submitting your assignment. One submission per group is sufficient.

   (b) Make sure all of your code is compilable and runnable.

      i. We should be able to simply run `make` in the `code` subdirectory and have everything compile without warnings.

      ii. We should be able to replace your versions of all of the Python code with the original versions and then perform regression testing and benchmarking.

   (c) Remove all nonessential files, especially output images, from your directories.

   (d) Run the command "`make handin.tar`." This will run "`make clean`" and then create an archive of your entire directory tree.

   (e) Submit the file `handin.tar` to Autolab.

2. Your report: Upload your report as file `report.pdf` to Gradescope, one submission per team. After submitting, you will be able to add your teammate using the *add group members* button on the top right of your submission.

A Set Partitioning

Suppose you want to distribute a set with N elements among P processes, including the case where N is not divisible by P. The usual way to do this is to assign $\lceil N/P \rceil$ elements to the first $P-1$ processes and the remaining ones to the final process.

This approach does not give a very uniform result. For example, with $N = 21$ and $P = 5$, we would have $\lceil 21/5 \rceil = 5$. The first 4 processes would get 5 elements, but the final process would get only 1. For the degenerate case where $N < P - 1$, this scheme fails altogether.

A better approach is to have all processes assigned at least $\lfloor N/P \rfloor$ elements, but with the first $N \bmod P$ processes assigned an additional one.

The following C code illustrates how to compute the starting position for the set of elements assigned to process i:

```c
int start_position(int N, int P, int i) {
    int base =  N / P;
    int extra = N % P;
    if (i < extra)
        return i * (base+1);
    else
        return i * base + extra;
}
```

For the example of $N = 21$ and $P = 5$, the function returns values 0, 5, 9, 13, and 17 for values of i ranging from 0 to 4. For the degenerate case of $N = 3$ and $P = 5$, it returns values 0, 1, 2, 3, 3. The latter two values indicate that no elements are assigned to processes 3 and 4. In general, the number of elements assigned to process i can be computed as `start_position(i+1) - start_position(i)`.