15-418/618, Spring 2018
Assignment 3
GraphRats!

Assigned:	Wed., Feb. 14
Due:	Wed., Mar. 7, 11:59 pm
Last day to handin:	Sat., Mar. 10

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 3 starter code from the course Github using:

```
linux> git clone https://github.com/cmu15418/asst3-s18.git
```

Assignment Objectives

This assignment will give you a chance to improve the performance of a program that implements a simple form of *agent modeling*. You will need to improve its sequential performance and then use OpenMP to further improve its performance through multithreaded parallelism. You will need to study the program to determine ways to improve its efficiency by avoiding redundant computation. Then you will need to find

ways to extract parallelism in ways that avoid sequential bottlenecks, memory contention, workload imbalance, and costly communication. You will need to instrument your code to determine where time is being spent and to evaluate your intermediate optimizations. Although the particular application is highly contrived, the structure of its computations are similar to those found in graph and sparse-matrix applications. The skills you develop will be transferable to other problem domains and to other computing platforms.

Machines

The OpenMP standard is supported by a variety of compilers, including GCC, on a variety of platforms. Programs can be written in C, C++, and Fortran. For this assignment, you will be working in C. 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, a collection of 17 Xeon processors, each having 12 cores. Jobs are submitted to this cluster through a batch queue, and so you can get fairly reliable timings.

Resources

There are many documents describing OpenMP, including those linked from the OpenMP home page at `http://www.openmp.org`. Like many standards, it started with a small core of simple and powerful concepts but has grown over the years to contain many quirks and features. You only need to use a small subset of its capabilities. A good starting point is the document at http://www.cs.cmu.edu/~418/doc/openmp.pdf.

2 Application

The renowned theoretical social scientist Dr. Roland ("Ro") Dent of the Reinøya Academy for the Technology Transfer of Universal Science, located in Reinøya Norway[1] has devised a mathematical model of how the geographic distribution of animals derives from their own social preferences. His thesis is that animals do not like being alone, but they also don't like being overcrowded, and they will migrate from one region to another in a predictable manner to achieve these preferences. He has performed small-scale studies using colonies of 1000 rats to formulate this model, and he is ready to test his theories at a much larger scale. Before doing so, he proposes using computer simulations to further explore his theoretical model.

In a major public relations coup, Carnegie Mellon University recruited Dr. Dent to the university, providing resources to establish the GraphRats Project. A major attraction to Dr. Dent was the availability of students, like you, who can develop software capable of performing large-scale simulations. Your job is to support Dr. Dent's efforts by creating a high-performance GraphRats simulator.

Model Parameters

Dr. Dent formulated the GraphRats model by discretizing space into unit squares, forming the nodes of a graph. The graph has N nodes. The edges E of the graph indicate which nodes are adjacent. That is,

[1] Often referred to by its Latin Name "RATTUS Norvegicus"

(a). Node counts

													1		1	1			
															1	9	9	6	7
											1		1	8	23	27	22	24	25
												3	9	11	22	24	23	37	25
												4	12	23	25	30	32	37	35	38
											3	4	20	31	33	31	29	40	37	33
											10	21	29	28	33	34	42	39	41	34
								1		4	19	23	26	33	41	46	44	49	38	52
									2	8	26	28	33	37	45	45	47	47	43	57
				1				1	3	16	25	29	35	27	37	54	53	61	69	51
								1	9	22	35	35	41	41	49	55	63	59	63	61
									15	18	31	37	31	55	51	58	66	70	70	64
									17	21	23	34	40	37	45	53	60	64	66	56

(b). Heat map of node load factors

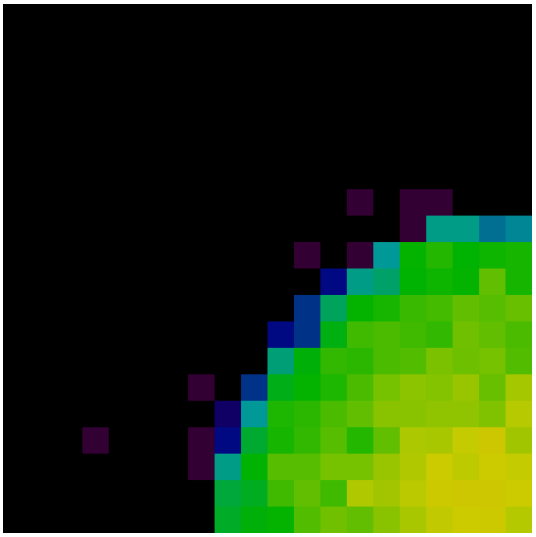


Figure 1: Representing the state of the simulation

$(u, v) \in E$ when square u is adjacent to square v. The edges are directed and symmetric: $(u, v) \in E$ if and only if $(v, u) \in E$. 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.

Rats are numbered from 0 to $R-1$. At any given time, each rat is assigned to a node, indicating its location. Rats move among the nodes, with each move following an edge.

We define the *average density* as the ratio $\lambda = R/N$, indicating the average number of rats per node. For a node u, we define its *count* $p(u)$ to be the number of rats at that node, and its *load factor* $l(u) = p(u)/\lambda$ to be the ratio between its count and the average.

Simulation State

At any time, the state of the model is represented by the positions of the rats. Based on these positions, the state of each node u can be expressed in terms of its count $p(u)$ or its load factor $l(u)$. Figure 1 shows two ways the GraphRats simulator can represent the model state for a 20×20 grid graph. In the *text* format (a), the count of each node in the grid is given as a decimal value. Blank grid positions indicate nodes u with $p(u) = 0$. In the *heat-map* format (b), the load factors are represented by colors, with longer wavelength colors (red, orange) indicating high load factors, and shorter wavelength colors (blue, violet) indicating low load factors. As this figure demonstrates, the text format is only suitable for small graphs, but it can be useful for debugging.

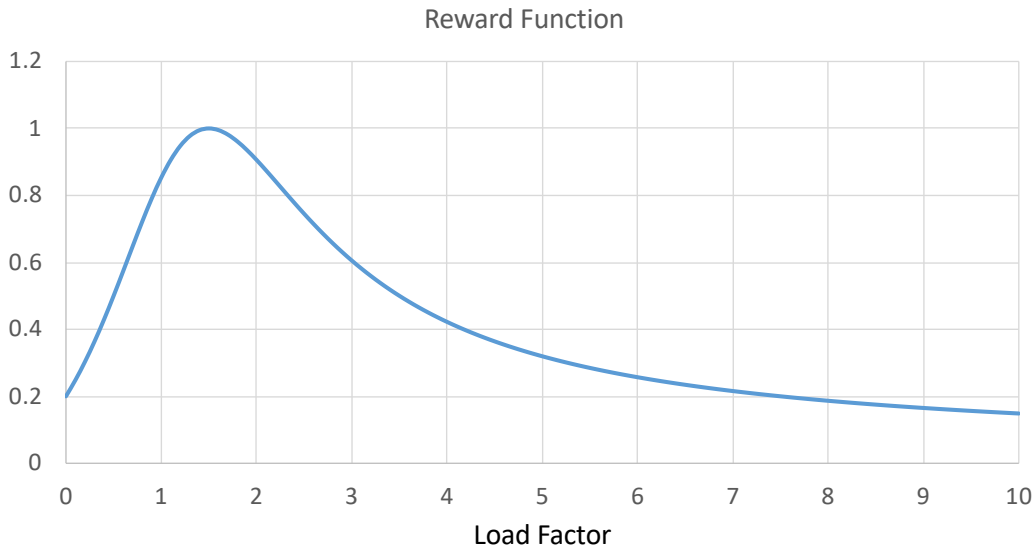


Figure 2: Reward as a function of load factor. The ideal load is 1.5.

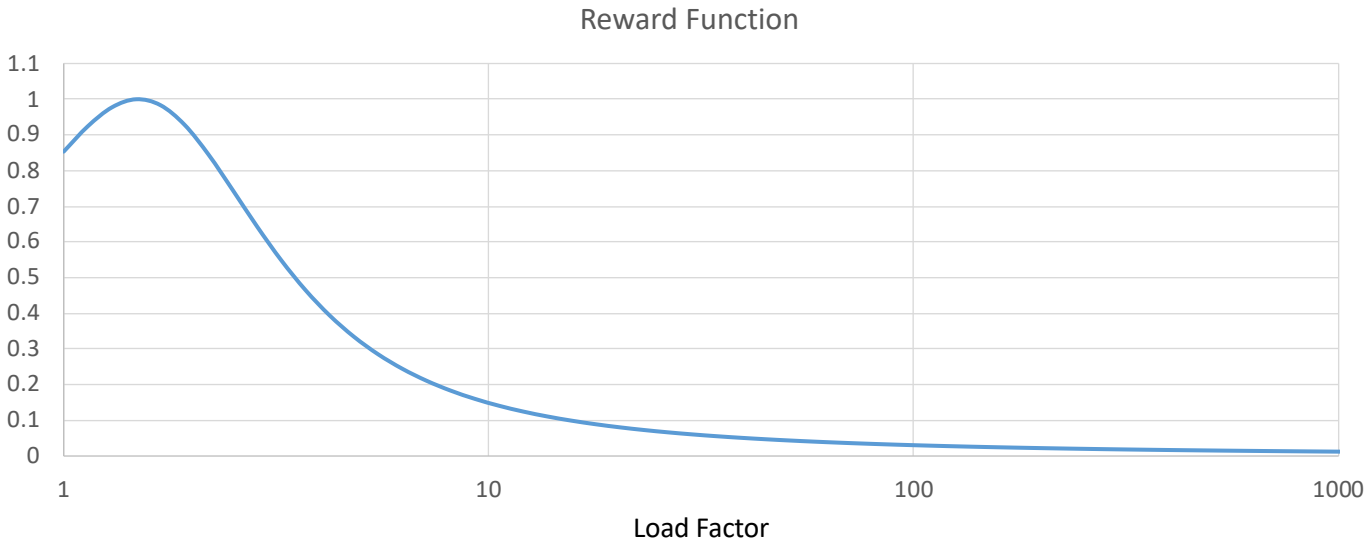


Figure 3: Reward function over wide range of load factors. The reward value drops off slowly.

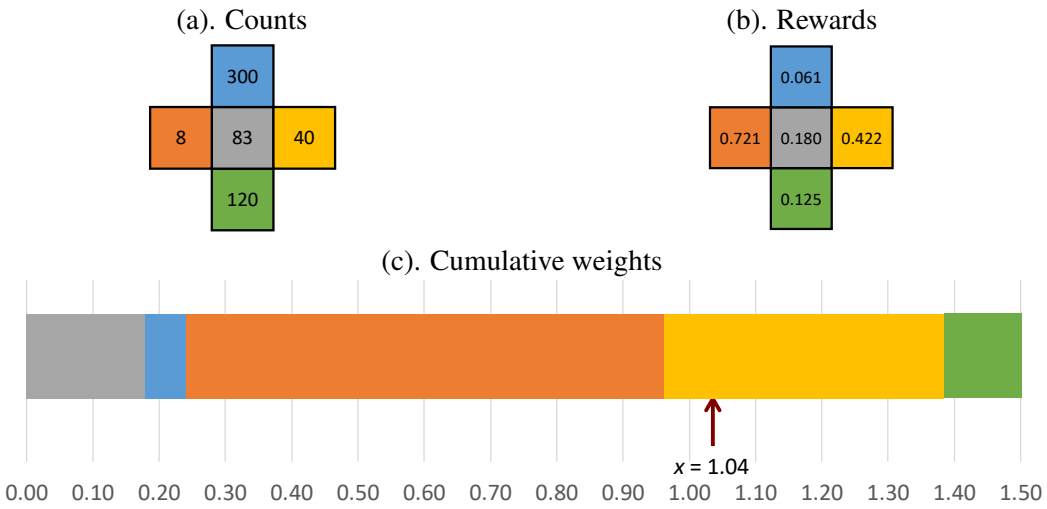


Figure 4: Next move computation. Each rat moves randomly, weighted by the reward values for the potential destinations.

Movement

Rats prefer to group together, but they don't like being too crowded, and they migrate in order to achieve these preferences. This migration is represented in the model by having the rats move from one node to another in order to maximize a *reward function* expressed in terms of the load factor l by the equation

$$Reward(l) \quad = \quad \frac{1}{1 + \left(\log_2 \left[1 + \alpha(l - l^*) \right] \right)^2} \tag{1}$$

where l^* is the *ideal load factor* and α is a coefficient determined experimentally. Based on his observations of rat colonies, Dr. Dent conjectures that the ideal load factor is $l^* = 1.5$ (i.e., rats like to cluster into groups just above the average density), and that $\alpha = 0.25$. This yields the reward function illustrated in Figures 2 and 3. In Figure 2, we see that rats prefer to be in groups with load factor 1.5, yielding a reward of 1.0. Lower densities are less preferable, down to a reward of 0.20 as the load factor approaches 0.0. Similarly, more crowded conditions are less desirable, with the reward dropping as the load factor increases. As Figure 3 illustrates, this drop is very gradual (note the log scale), having values $Reward(10) \approx 0.149$, $Reward(100) \approx 0.030$, and $Reward(1000) \approx 0.0123$.

Each time a rat moves, it does so randomly, but with the random choice weighted by the rewards at the possible destinations. More precisely, a rat at node u considers the count at its current node $p(u)$, as well as those at each adjacent node $p(v)$, such that $(u, v) \in E$. This is illustrated in Figure 4(a), for a node having only grid connections. Each of the possible destinations (including possibly staying at the current node) has an associated reward value, based on computing the reward function for its load factor. These values are shown in Figure 4(b) for the case of $\lambda = 10$. If we arrange these values along a line (starting with the node and then following a row-major ordering of the neighbors), as is shown in Figure 4(c), they form a sequence

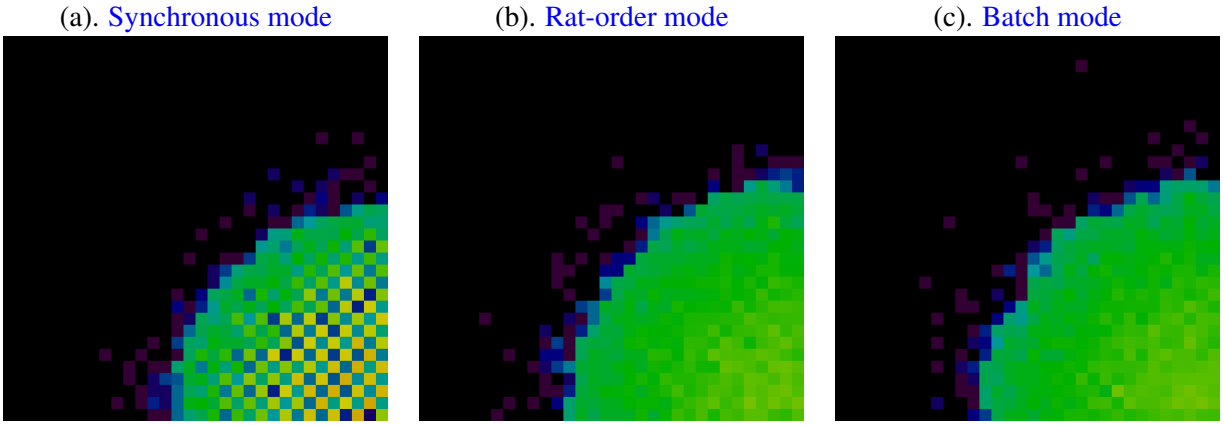


Figure 5: Effect of different simulation modes.

of subranges summing to a total value s. (In the example of Figure 4, $s = 1.509$.) Choosing a random value $x \in [0, s)$ selects the destination node v, for which x lies within the subrange associated with v. The figure illustrates the case where $x = 1.04$, causing the rat to move to the square on the right.

Each time a rat moves, it changes the reward values at its source and destination nodes, and these affect the choices made by other rats. We consider three different rules for computing and updating the rat positions:

Synchronous: All new positions are computed, and then all rats are moved.

Rat-order: For each rat, a new position is computed and then the rat is moved.

Batch: For batch size B, the new positions for one set of B rats is computed, and then these rats are moved. This process is repeated for all batches.

More precisely, we can characterize all three of these rules as variants of a batch mode, where the batch size equals R for synchronous mode, 1 for rat-order mode, and some intermediate value B for batch mode. The computation can therefore be summarized by the following pseudo-code:

```
For b from 0 to ⌈R/B⌉ − 1
    For r from b · B to min[(b + 1) · B, R] − 1
        Compute destination node for rat r
    For r from b · B to min[(b + 1) · B, R] − 1
        Move rat r to its destination
```

Figure 5 illustrates how the different modes lead to different simulation behavior. These examples all started by initializing the simulation with 10,240 rats in the lower, right-hand corner of a 32×32 grid graph and running for 100 steps. In the synchronous mode simulation (a), we can see a checkered pattern. This is an artifact of the mode—on each step, all of the rats base their next choice on the previous distribution of

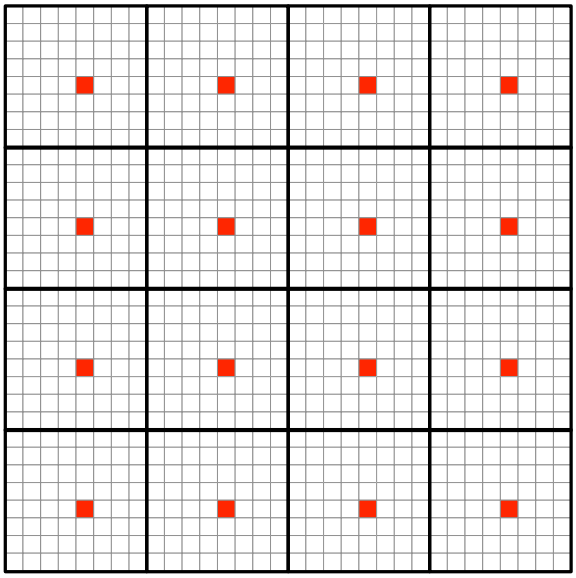


Figure 6: 32×32 tiled graph. Red nodes denote hubs, having edges to every node in the 8×8 region.

counts. Many will move to the same, low-count nodes, leaving their previous nodes almost empty. When you watch the simulation running, you will see oscillatory behavior, yielding the checkering pattern. Rat-order mode (b) yields a much smoother spreading of the rats. Unfortunately, rat-order mode does not lend well to parallel computation, and so batch-order mode (c) is presented as a compromise. By setting the batch size B to 2% of R, the smooth spreading occurs, but in a way that is amenable to parallel execution.

Graphs

All of the graphs used in the simulation are based on grids consisting of a $k \times k$ array of nodes. The simplest version is the *grid* graph, having only the grid connections as edges. That is, each node has an edge to the nodes above, to the left, to the right, and below.

Figure 6 illustrates the extension of the grid to a *tiled* graph. That is, square blocks of the graph (shown with outlined boxes) are referred to as *regions*. Each region contains a designated *hub* node, having edges to all other nodes in the region. One consequence of this structure is that the nodes have varying *degrees*. With the regions consisting of $d \times d$ nodes, most nodes have at most five edge connections, but the hub nodes have degree $d^2 - 1$.

Figure 7 shows the even more exotic *fractal* graph.[2] This graph is also partitioned into regions, some being square and some being rectangular, and having different sizes. Each of the rectangular regions contains four hub nodes. We can see, therefore, that the graph is highly irregular. The four hub nodes in the largest region have connections to half of the nodes in the graph. You will find that running simulations with this graph

[2]You may ask: "What kind of rat environment would be represented as a fractal graph?" Dr. Dent hypothesizes that it is representative of real-world urban environments.

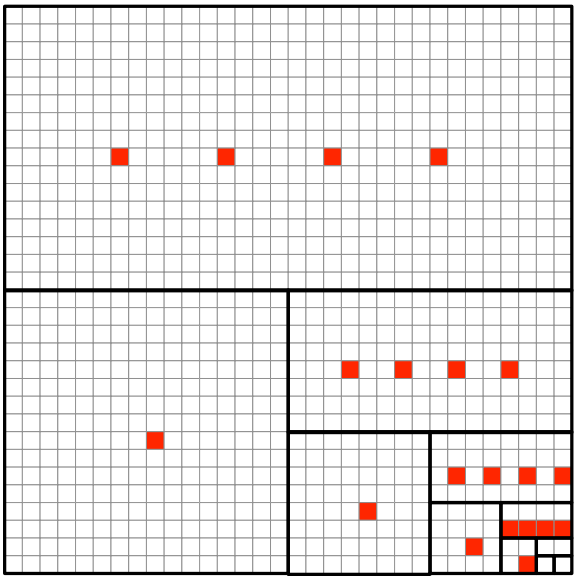


Figure 7: 32×32 fractal graph. Red nodes denote hubs, having edges to every node in its region, with the regions varying in size. Some regions have four hubs, while others just one.

presents challenges for efficiency and for workload balance.

For benchmarking purposes, your simulator will be run on graphs with grids of 160×160 =25,600 nodes, and with load factors of 40, for a total of 1,024,000 rats. The basic grid graph has 101,760 edges; the tiled graph has 152,850 edges,, and the fractal graph has 254,940 edges. The tiled graph has a maximum node degree of 99, while the fractal graph has a maximum node degree of 12,799.

3 Your Task

The starter code, in the `code` subdirectory, contains two fully functional GraphRats simulators, one written in Python (`grun.py`) and one in C (compiling to a sequential version `crun` and a parallel version `crun-omp`.) The C versions compile (compile with `make all`) and run correctly, but they are very slow. The Python version is even slower. Your job is to make both C versions run faster, first through sequential optimizations and then by using OpenMP pragmas and library functions to express parallelism.

In doing your optimizations, you must preserve the *exact functionality* of the provided code. Despite the seeming randomness of how rat moves are generated, the program is completely deterministic. Given the following parameters: 1) the graph, 2) the initial rat positions, 3) a global seed, 4) the update mode, and 5) the number of simulation steps, the program will produce the exact same sequence of rat moves—and therefore the exact same node counts—every time it is run. This determinism is maintained in a way that won't cause any sequential bottlenecks by associating a separate pseudo-random number seed with each rat.

Running the Simulator

The simulator can be run with many options. 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] [-t THD]
   -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
   -t THD    Set number of threads
```

You must provide a graph file and a file describing the initial rat positions. These are included in the subdirectory `code/data`. Graph file names have the form:

g–*TNNN*.gph

where *T* indicates the graph type: `u` ("uniform") for a simple grid, `t` for a tiled graph, and `f` for a fractal graph. The number *NNN* indicates the number of nodes in the graph. Rat file names have the form:

r–*NNN*–*PLL*.rats

where *NNN* indicates the number of nodes, *P* describes the initial positions: `u` for distributed uniformly, `d` for distributed along the diagonal, and `r` for all being in the lower, right-hand corner. The number *LL* indicates the load factor.

Additional optional arguments allow you to specify the number of simulation steps, the global seed, the update mode (random, batch, or synchronous), and the number of threads (only meaningful when running `crun-omp`.)

By default, the simulator prints the node counts on every step. Be careful to do this only for small graphs and small simulation runs! The optional flag `-q` instructs the simulator to operate in "quiet" mode, where it does not print any simulation results. Setting the display update interval to I with the `-i` flag causes the simulator to only print the node counts once every I time steps.

The simulator measures its elapsed time and expresses the performance in terms of "mega-rats per second" (MRPS). If a simulation of R rats running for S steps requires T seconds, then MRPS is computed as $10^{-6} \cdot R \cdot S/T$.

Visualizing the Simulation Results

The Python program `grun.py` can act as a simulator, accepting the same arguments as `crun`. In addition, it can serve as a *visualizer* for other simulators, generating both the text and heat-map displays of the

simulation state. This is done by piping the output of the simulator into `grun.py` operating in *driven mode*.

As an example, the visualization shown in Figure 5(c) was generated with the command:

```
linux> ./crun -g data/g-u1024.gph -r data/r-1024-r10.rats -n 100 -u b | ./grun.py -d -v h
```

We can see that the simulation was run on a 32×32 grid graph, with 10,240 rats initially in the lower, right-hand corner. The simulation ran for 100 steps in batch mode. These results were piped into `grun.py` to provide a heat-map visualization. (You must have an X window connection to view the heat maps.) The command line option `-v a` (for "ASCII") will generate a text representation of the node counts. (If you try to do this for a graph that won't fit in your terminal window, the program will fail silently.)

Regression Testing

The provided program `regress.py` provides a convenient way for you to check the functionality of your program. It compares the output of the C versions of the simulator with the Python version for a number of (mostly small) configurations. Its usage is as follows:

```
linux> ./regress.py -h
Usage: ./regress.py [-h] [-c] [-t THD]
    -h       Print this message
    -c       Clear expected result cache
    -t THD   Specify number of OMP threads
       If > 1, will run crun-omp.  Else will run crun
    -a       Run ALL tests, including for big graphs
```

The program maintains a cache holding the simulation results generated by the Python simulator. You can force the simulator to regenerate the cached values with the `-c` flag. By default, or when the option `-t 1` is given, it will test program `crun`. For higher thread counts, it will run `crun-omp` with that number of threads. Normally, regression testing is performed only on a collection of graphs that are smaller than will be used for benchmarking. Specifying the `-a` option will also cause testing of the larger graphs for a small number of simulation steps. It will take a while for the Python simulator to complete these simulations, but once they are added to the cache, the only performance limitation will be due to your simulation code.

Doing frequent regression testing will help you avoid optimizing incorrect code. Realize also, that the tests performed are not comprehensive. We reserve the right to evaluate your program for correctness on other graphs!

Performance Evaluation (80 points)

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

```
linux> ./benchmark.py -h
Usage: ./benchmark.py [-h] [-s SCALE] [-u UPDATELIST] [-t THREADLIMIT]
```

```
[-f OUTFILE] [-c]
(All lists given as colon-separated text.)
    -h            Print this message
    -s SCALE      Reduce number of steps in each benchmark by specified factor
    -u UPDATELIST Specify update modes(s):
       r: rat order
       s: synchronous
       b: batch
    -t THREADLIMIT Specify upper limit on number of OMP threads
       If > 1, will run crun-omp.  Else will run crun
    -f OUTFILE    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
```

Here's a brief description of the options:

`-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.

`-t` T Only run the benchmarks that use up to T threads. Single-threaded benchmarks run the program `crun-seq`. Multi-threaded benchmarks run the program `crun-omp`.

`-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`$\cdots$`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.

You can run `benchmark.py` on any machine, but the grading standards will be based on the performance running on a Latedays processor (see the next section about how to use these processors.)

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

- The mean speed (in MRPS) of the single-core performance

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

The idea here is that we really want your twelve-core simulator to run fast, but this is best achieved by speeding up a fast, single-core version. Although you could improve the apparent speedup by detuning the single-core performance, that will not lead to an optimal overall score. As a special case, if your single-core 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-threaded version in order to maximize the speedup measurement.

The following table provides performance targets for MRPS and for speedup.

Mode	1-thread MRPS	12-thread MRPS	Speedup
batch	20	70	3.5
synchronous	32	256	8.0
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.

Running on the Latedays Cluster

The Latedays cluster contains 18 machines (1 head node plus 17 worker nodes). Each machine features:

- Two, six-core Xeon e5-2620 v3 processors (2.4 GHz, 15MB L3 cache, hyper-threading, AVX2 instruction support).
- 16 GB RAM (60 GB/sec of BW)

You can login to the Latedays head node `latedays.andrew.cmu.edu` via your Andrew login. You can edit and compile your code on the head node, and then run jobs on the cluster's worker nodes using a batch queue. You have a home directory on Latedays that is not your Andrew home directory. (You have a 2GB quota.) However, your Andrew home directory is mounted as `~/AFS`.

Do not attempt to submit jobs from your AFS directory, since that directory is not mounted when your job runs on the worker nodes of the cluster. (It is only mounted and accessible on the head node.) Please copy your source code over to a subdirectory you have set up in your Latedays directory and recompile it.

To submit a job to the job queue, use the default submission script we've provided you: `latedays.sh`.

```
linux> qsub latedays.sh
```

This code will execute the program `benchmark.py`. After a successful submission, the script will echo the name of your job. For example, if your job was given the number 337 by the job queue system, the job would have the name: `337.latedays.andrew.cmu.edu`.

After you submit a job, you can check the status of the queue via one of the following commands:

```
linux> showq
linux> qstat
```

When your job is complete, log files from standard output and standard error will be placed in your working directory as `latedays.qsub.o337` and `latedays.qsub.e337` (substituting your job number for 337, of course.) In addition, a summary of the results will be written to the file `benchmark.out`.

Looking at `latedays.sh`, you will see that the maximum wall clock time for the job is limited to 20 minutes. This means that the job scheduler will cut your program off after 20 minutes.

Some other things to keep in mind:

- Only use the Latedays head node for tasks that require minimal execution time, e.g., editing files, compiling them, and running short tests.
- The Latedays Python installation does not contain the libraries required by `qrun.py` for visualization.

4 Some Advice

How to Optimize the Program

- Your simulations will run for enough steps that it will prove profitable to spend time at the beginning setting up additional data structures that will improve the subsequent simulation performance. Any such preprocessing must be done as part of your simulation program. You cannot, for example, store additional information in a file and then load it at the beginning of the simulation.
- There are many ways to reduce the amount of computation performed during the simulation. For example, within a single batch, all of the rats at some node will base their next moves on the same sequence of reward values (Figure 4). Can you avoid recomputing this sequence over and over again?
- If you perform the next-move computation illustrated in Figure 4 via linear search, you will achieve very poor performance for the hub nodes, having as many as 12,800 possible destinations. Can you think of a better way?
- There are many sources of parallelism in this program: across nodes, edges, and rats. You will want to exploit different forms of parallelism for the different simulation operations.
- When you use OpenMP static partitioning of `for` loops, it will divide the iterations into blocks of even size. When done across the nodes, this might yield an unbalanced load, due to the nonuniform degrees of the nodes. You could use dynamic parallelism, but it might be better still to implement a customized form of static partitioning.

Important Requirements and Tips

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

- Don't print on `stdout`. Since the simulator is designed to pipe its output into a visualization program, standard output should be reserved for simulation results. If you want to print error messages or other information, use `stderr`. The provided function `outmsg` will prove useful for this.
- 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.
- Although you will only be tuning your code for a small number of graphs and initial rat states, you should design your code to use techniques and algorithms that will generalize across a wide range of graph/state combinations. Think of the benchmark cases as points in a large parameter space. Your program should be designed to achieve reasonable performance across the entire space.
- The provided simulator does not use any global variables. It encapsulates the graph structure and the simulation state into structs, with all arrays allocated dynamically. You would do well to continue that convention to keep your code modular and to avoid having any built-in limits on the data structure sizes.
- The provided `cycletimer.h` and `cycletimer.c` files provide low cost timer routines. You would do well to instrument your code to track the time spent doing different types of operations required, e.g., computing the next moves, updating the rat positions, etc. This will help you identify the most costly operations and to determine how the different operations scale as the number of threads increases.
- Using synchronization constructs, such as atomic operations, can be very costly. You would do better to have the threads first update local copies of data structures, and then work together to aggregate these into a single data structure.
- 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.

5 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 avoid wasting time using synchronization constructs?
   - 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 threads? 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?

6 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
      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
   (a) If this is your first time using Gradescope, go to `https://gradescope.com` and sign up for free with the entry code `9RW38E`. Please register using your Andrew email address as id. If you already have a Gradescope account, the course should appear on your dashboard.
   (b) Please upload your report as file `report.pdf` to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the *add group members* button on the top right of your submission.

A A Gallery of Rat Simulations

We encourage you to explore the behavior of the simulator using different combinations of graph structure and initial state. You will gain a better understanding of the factors that affect simulation performance. For starters, the provided make file will run 10 different demonstrations. Try running

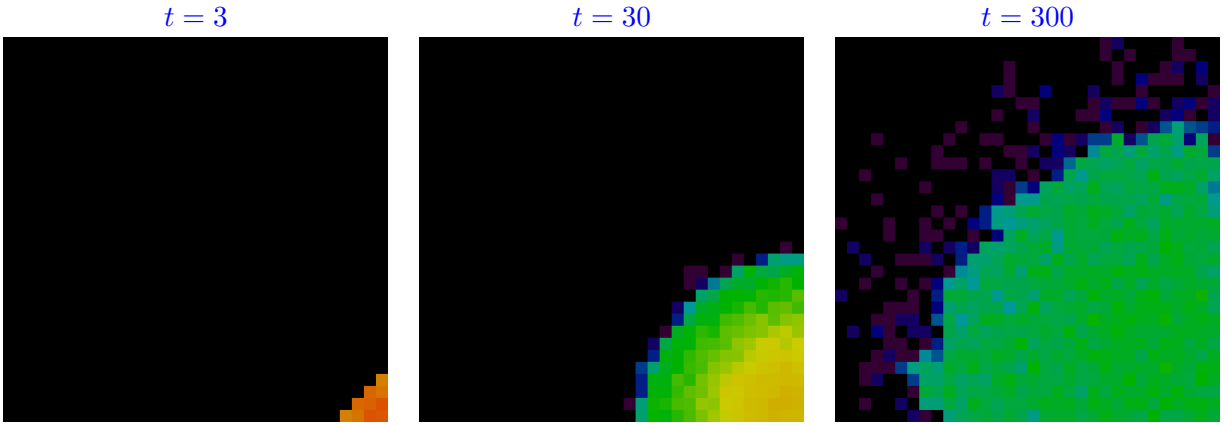
```
linux> make demoX
```

for *X* ranging from 1 to 10.

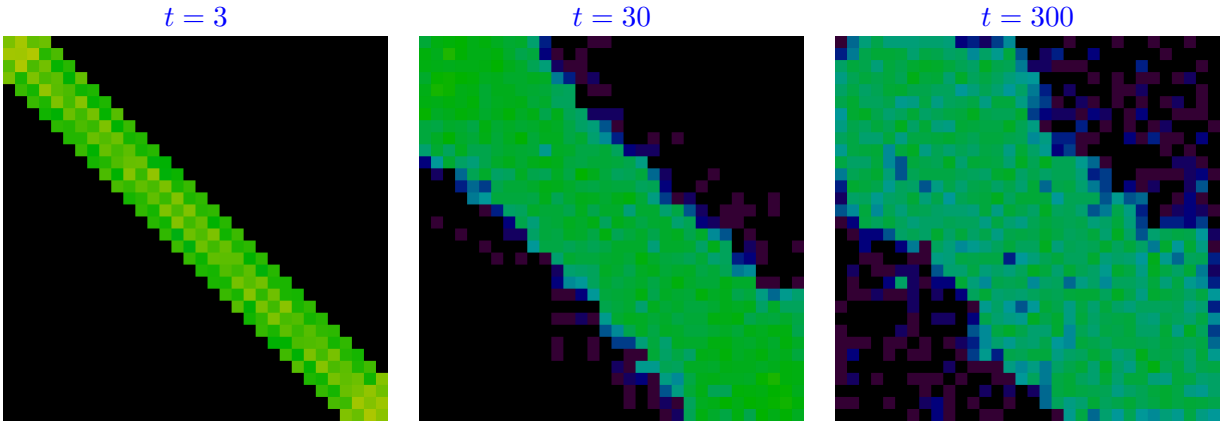
Figure 8 shows simulations of the same graph—a 32×32 grid with different initial conditions. With the rats initially in the lower right hand corner (a), they expand outward to reduce crowding. The progression is slow, however, since nodes with low occupancy also have low reward values. You can see a few nodes that are dark violet. These have only a single, isolated rat. With the rats initially along the diagonal (b), they expand outward, but then they begin to form uneven distributions. This unevenness is even more pronounced when starting with a uniform distributions (c). The fact that the reward value is maximized with a load factor of 1.5 causes the rats to gather into clumps. As the simulation continues, these clumps become larger and more distinct.

Figure 9 shows simulations of three different 32×32 graphs, in each case with the rats distributed uniformly along the diagonal. The simulation of the grid graph (a) is the same as in Figure 8(b). For the fractal graph (b), we see that the hub nodes become highlighted. Their high connectivity causes many rats to go to them, even though they are crowded. These hubs also cause the rats to quickly distribute among their regions. Similarly, the hubs of the tiled graph (c) become highlighted and cause the rats to distribute among their regions, but the smaller size of these regions reduces the rate of spreading.

(a). Rats initially in lower, right-hand corner



(b). Rats initially along diagonal



(c). Rats initially distributed uniformly

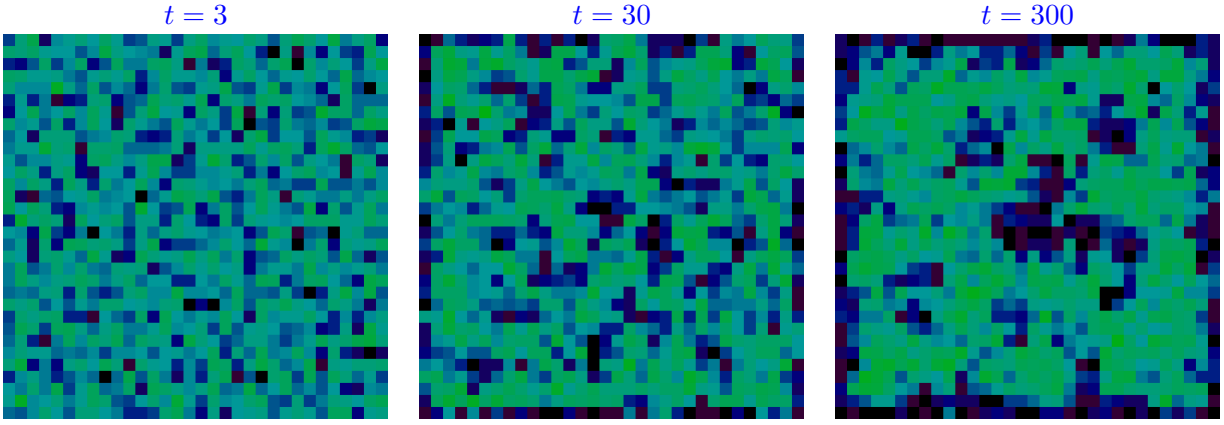
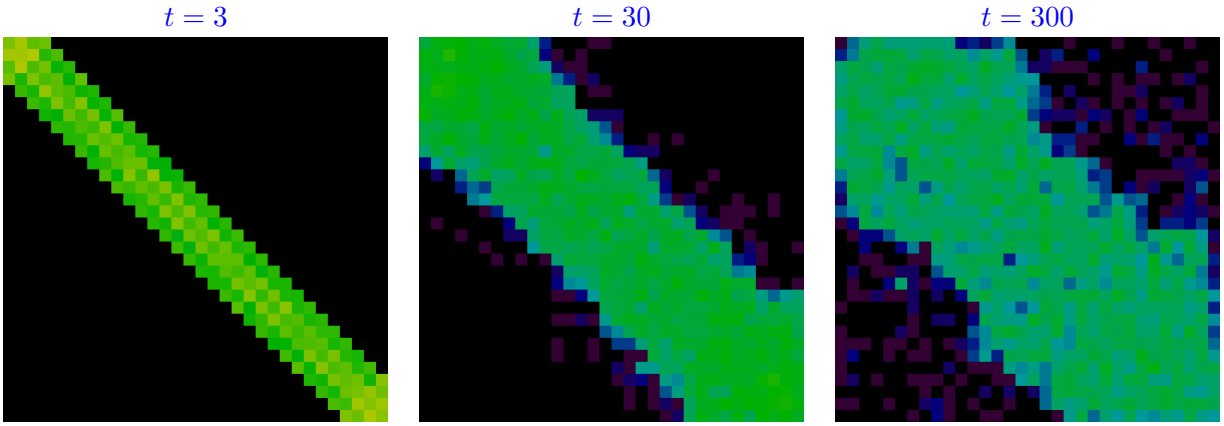
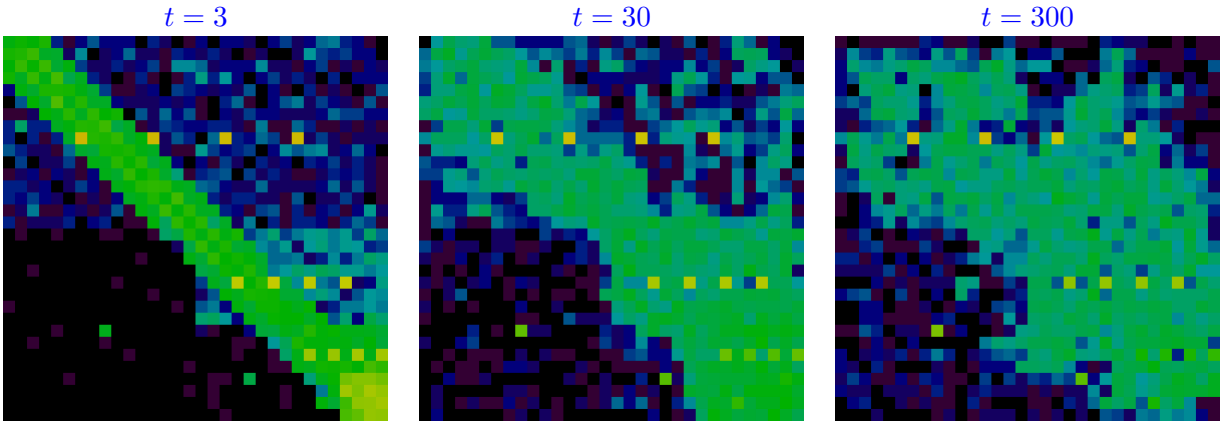


Figure 8: Simulations of a 32×32 grid graph with different initial conditions ($\lambda = 10$)

(a). Grid graph



(b). Fractal graph



(c). Tiled graph

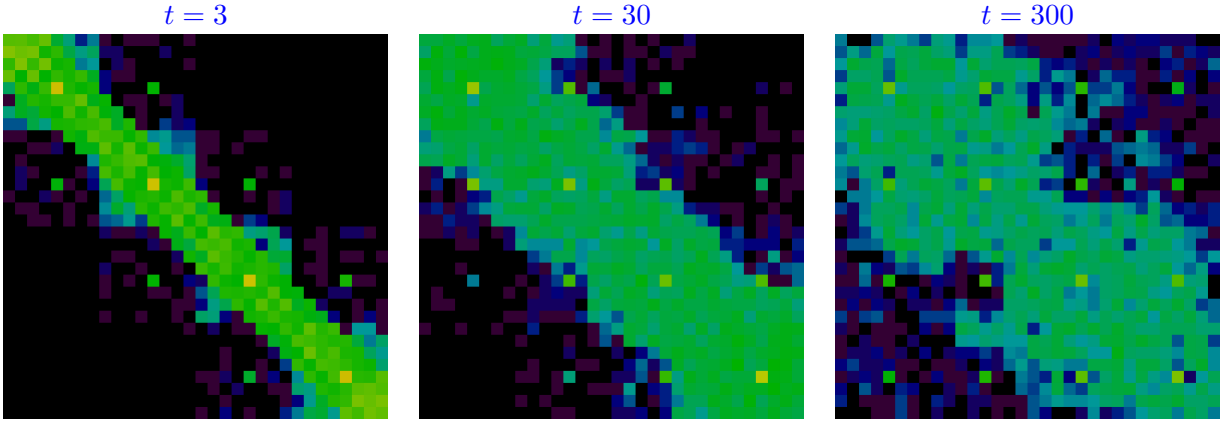


Figure 9: Simulations of different 32×32 graphs. Rats initially distributed along the diagonal ($\lambda = 10$)