15-418/618, Spring 2019
Assignment 3
GraphRats!

Assigned:	Fri., Feb. 15
Due:	Wed., Mar. 6, 11:59 pm
Last day to handin:	Sat., Mar. 9

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-s19.git
```

Assignment Objectives

This assignment will give you a chance to optimize a program to run fast on a shared-memory multiprocessor. You will do so by modifying a sequential implementation of the program to make use of the OpenMP programming framework. Along the way, you will gain experience with the following:

- Making use of different sources of parallelism during different phases of a program.

- Identifying bottlenecks that limit the ability to exploit parallelism in a program and then finding ways to eliminate them.
- Making tradeoffs between deciding what values need to be recomputed and computing just those, versus simply recomputing everything.
- Finding ways to reduce memory contention by having different threads operate on separate copies of a data structure, and then merging the results together.
- Recognizing sources of imbalanced load among the threads and finding ways to mitigate these.

Although the particular application is highly contrived, the structure of its computations is similar to that 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. His long-term goal is to set up and evaluate a rat colony containing over one million rats. While he is gathering the resources to build the colony and breeding the rats to populate it, he would like to use computer simulations to further explore his theoretical model.

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

(a). Node counts

```
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   | 1 |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   | 1 |   | 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   | 5 |   | 9 | 5 |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   | 1 | 1 | 5 | 9 | 12|
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   | 8 | 5 | 7 | 15| 10| 9 |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   | 9 | 14| 14| 17| 15| 8 |
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   | 1 |   | 1 | 13| 13| 10| 13| 20| 14|
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   | 1 | 3 | 12| 16| 12| 19| 18| 18|
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   | 1 | 11| 15| 12| 20| 17| 16| 16|
+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   | 3 | 10| 10| 13| 18| 19| 11| 19|
+---+---+---+---+---+---+---+---+---+---+---+---+
```

(b). Heat map of node load factors

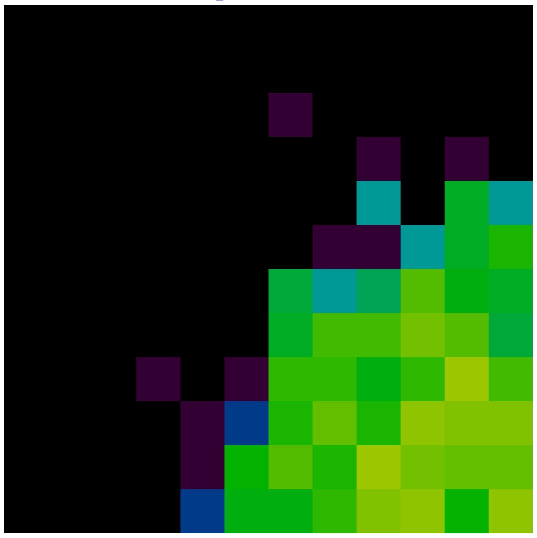

Figure 1: Representing the state of the simulation

Last year, Dr. Dent came to CMU as a visiting scholar. He recruited students from 15-418/618 to write simulators and was very pleased with the insights the simulations provided. He would now like to run simulations on a more sophisticated, and computationally demanding, theoretical model. Your job is to support Dr. Dent's efforts by creating version 2.0 of a high-performance GraphRats simulator. Fortunately, for you, he has available a well-optimized sequential implementation, and so you can focus your efforts on maximizing its parallel performance.

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, $(u, v) \in E$ 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. That is, node u has an edge to its neighbors to the left, right, above, and below. The graphs also have edges connecting nodes to ones beyond 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.

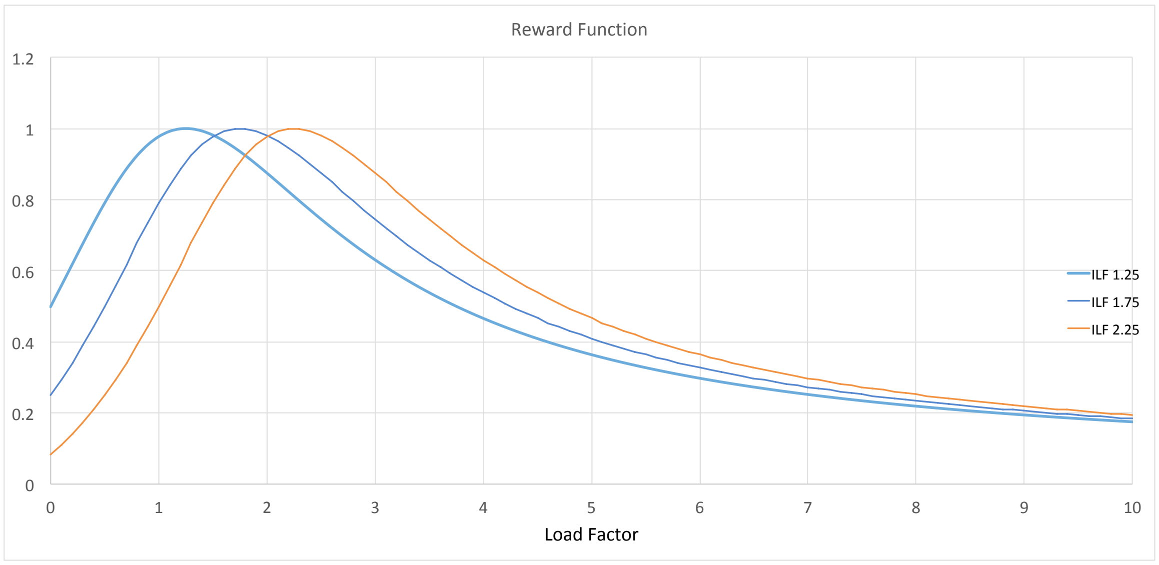

Figure 2: Reward as a function of actual and ideal load factors. The ideal load factors range between 1.25 and 2.25.

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 12×12 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. Black indicates a count of 0. As this figure demonstrates, the text format is only suitable for small graphs, but it can be useful for debugging.

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

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

where l^* is the *ideal load factor* (ILF) and α is a coefficient determined experimentally. Based on his observations of rat colonies, Dr. Dent conjectures that $\alpha = 0.40$.

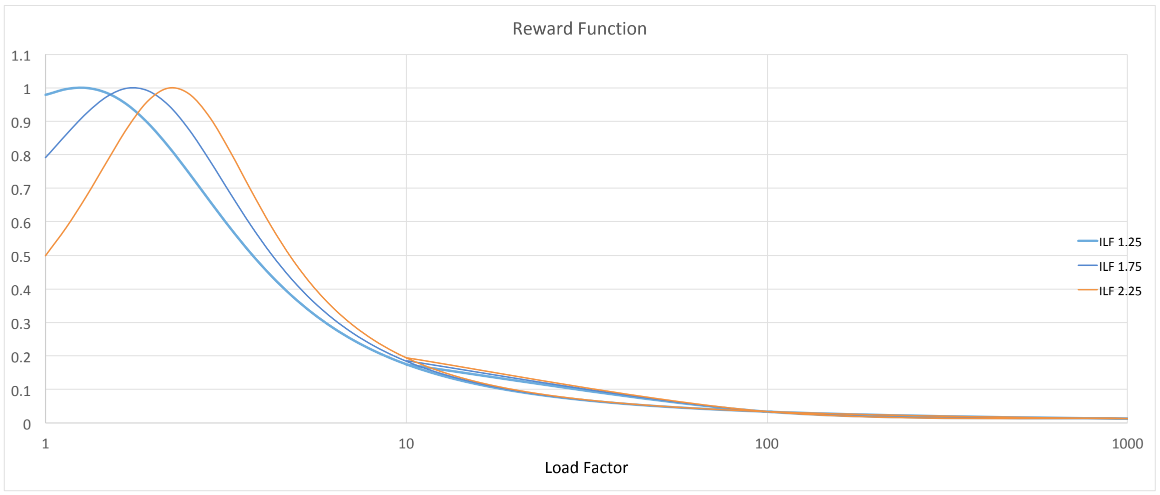

Figure 3: Reward functions over wide range of load factors. The reward value drops off slowly.

He also conjectures that the ideal load factor l^* ranges between 1.25 and 2.25, as explained below. This range indicates that rats like to cluster into groups above the average density. The values of the reward function are illustrated in Figures 2 and 3 for ILF values of 1.25, 1.75, and 2.25. In Figure 2, we see the reward is maximized when the load factor equals the ideal. Lower densities are less preferable, down to rewards ranging between 0.083 and 0.500 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), and less dependent on the ideal load factor, with $Reward(10)$ ranging between 0.175 and 0.194, $Reward(100)$ ranging between 0.03388 and 0.03406, and $Reward(100)$ ranging between 0.013202 and 0.013206.

In his earlier model, Dr. Dent used a constant value for the ideal load factor l^* across all nodes. In recent observations of rats, however, he has noted that they are more willing to crowd together in one location if the nearby locations have a relatively higher population. He would like to test out the impact of this preference by formulating an *adaptive* ideal load factor. That is, for counts c_l (the local count) and c_r (the remote count), define the *imbalance factor* $\beta(c_l, c_r)$ as:

$$\beta(c_l, c_r) = \begin{cases} -1.0 & (c_r = 0 \text{ and } c_l > 0) \text{ or } c_l/c_r \geq 10.0 \\ +1.0 & (c_l = 0 \text{ and } c_r > 0) \text{ or } c_r/c_l \geq 10.0 \\ \log_{10} c_r/c_l & \text{else} \end{cases} \qquad (2)$$

For node u, let $\hat{\beta}(u)$ to be the average value of $\beta(p(u), p(v))$ for all adjacent nodes v. Then the ideal load factor for u is computed as:

$$l^*(u) = 1.75 + 0.5 \cdot \hat{\beta}(u) \qquad (3)$$

We can see that the ILF will have its minimum value of 1.25 when $\hat{\beta}(u) = -1$ (the neighboring cells are much less crowded) and its maximum value of 2.25 when $\hat{\beta}(u) = +1$ (the neighboring cells are much

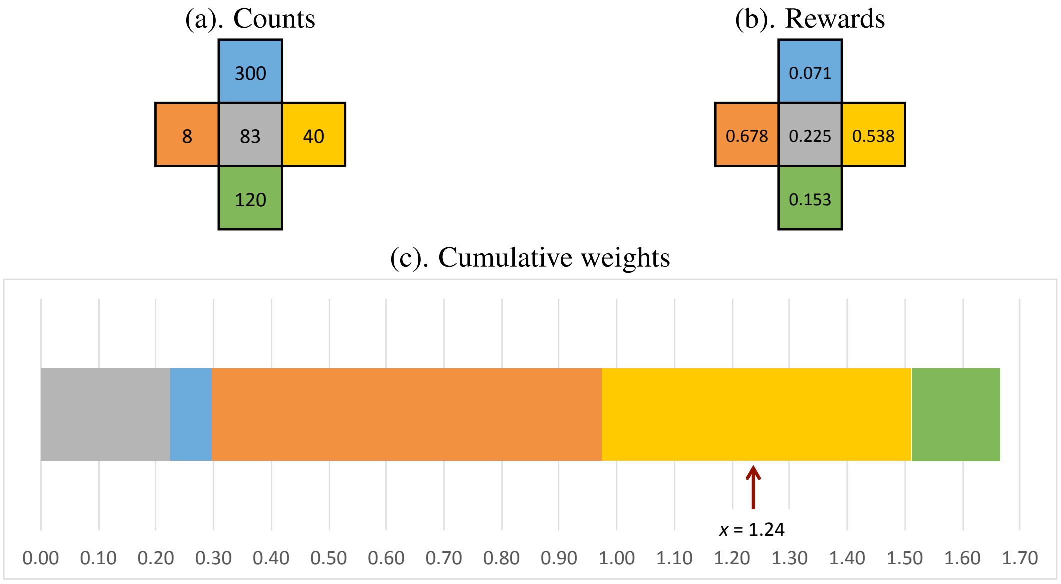

Figure 4: Next move computation. Each rat moves randomly, weighted by the reward values for the potential destinations.

more crowded.) Observe that this value will change over time, as the counts at u and its neighboring nodes change.

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 of subranges summing to a total value s. (In the example of Figure 4, $s = 1.665$.) 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 $x = 1.24$, causing the rat to move to the square on the right.

Each time a rat moves, the count at the previous node decreases, while the count at the new node increases. This, in turn, will causes changes in the reward and ILF values at these nodes, as well as the ILF values of nodes adjacent to them, and this will 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.

(a). Synchronous mode (b). Rat-order mode (c). Batch mode

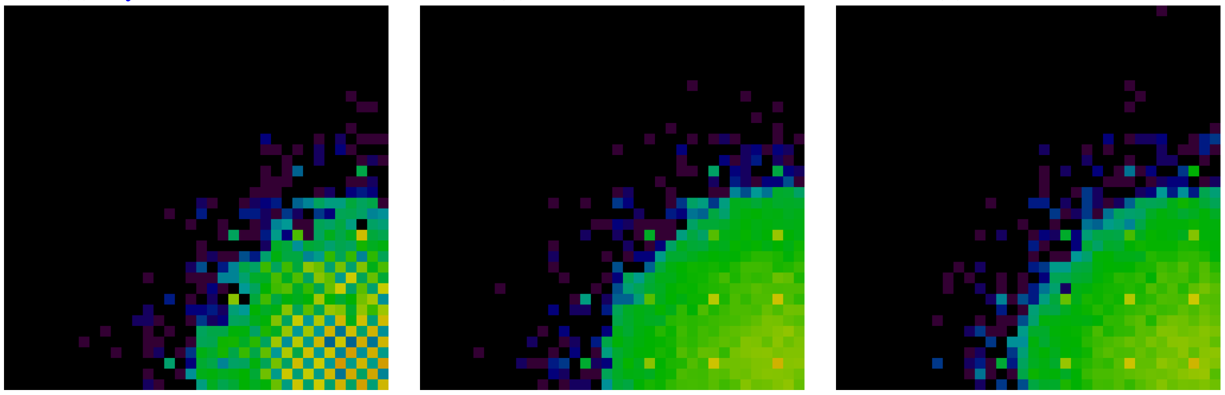

Figure 5: Effect of different simulation modes.

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 $\lceil R/B \rceil - 1$
    For $r$ from $b \cdot B$ to $\min[(b+1) \cdot B, R] - 1$
        Compute destination node for rat $r$
    For $r$ from $b \cdot B$ to $\min[(b+1) \cdot B, R] - 1$
        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 12,960 rats in the lower, right-hand corner of a 36×36 graph and running for 50 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 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

Figure 6 illustrates the graphs that will be used in benchmarking your simulator. All of them are based on grids consisting of a $k \times k$ array of nodes, augmented with a set of *hub* nodes (shown in red). Each hub connects to every node in a *region* (shown with black outlines). A hub provides a high degree of connectivity among the nodes within a region, whereas the only connections from one region to another are via the grid

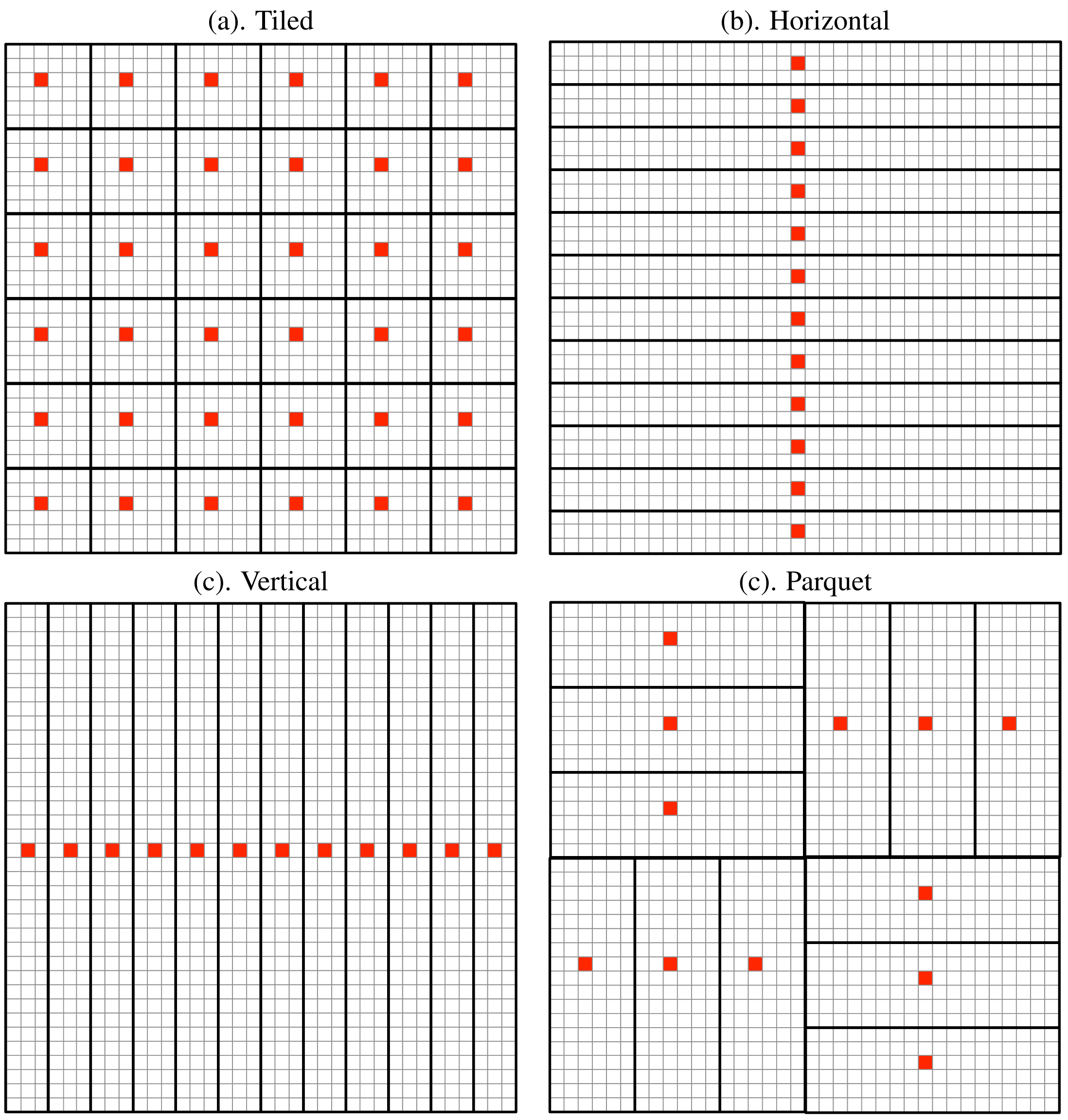

Figure 6: Graph types, for $k = 36$. Red nodes denote hubs, having edges to every node in the region enclosed in its box.

Graph	Edges	Hubs	Max. Degree
Tiled	193,320	36	899
Horizontal	193,560	12	2,699
Vertical	193,560	12	2,699
Parquet	193,560	12	2,699

Figure 7: Graph statistics for $k = 180$ (32,400 nodes)

edges. Figure 7 provide the statistics for 180×180 versions of these graphs. These graphs all have 32,400 nodes and either 193,320 or 193,560 edges. One consequence of the hub structure is that the nodes have varying *degrees*, with most nodes having at most five connections (four grid edges plus one connection to a hub), while the hubs have degrees of either 899 or 2,699, depending on the graph type.

For benchmarking purposes, your simulator will be run on these 180×180 graphs with average load factors of 32, for a total of 1,036,800 rats.

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-seq` and a parallel version `crun-omp`.) The Python version is very slow, but it serves as the standard definition of correct behavior. The C versions compile (compile with `make all`) and run correctly. The sequential version has already been carefully optimized. It computes and stores the various weights once for each batch, and it uses binary search to determine where each rat should move. Your job is to make the C version run faster through parallel execution, using OpenMP pragmas and functions, along with whatever changes you want to make to the data structures and simulation code.

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-XXX -h
Usage: ./crun-XXX  -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
```

where *XXX* is either "`seq`" or "`omp`."

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-`*TKKK*`x`*KKK*`.gph`, where *T* indicates the graph type: `t` for a tiled graph, `h` for a horizontal graph, `v` for a vertical graph, and `p` for a parquet graph. The number *KKK* indicates the value of k.

Rat file names have the form `r-`*KKK*`x`*KKK*`-`*PLL*`.rats`, where *KKK* indicates the value of k, *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 average 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 *nanoseconds per move* (NPM). If a simulation of R rats running for S steps requires T seconds, then NPM is computed as $10^9 \cdot T/(R \cdot S)$.

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-seq -g data/g-t036x036.gph -r data/r-036x036-r10.rats -n 50 -u b | ./grun.py -d -v h
```

We can see that the simulation was run on a 36×36 grid graph, with 12,960 rats initially in the lower, right-hand corner. The simulation ran for 50 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
```

Name	Graph Type	Rat Positions
A	Tiled	Uniform
B	Horizontal	Uniform
C	Vertical	Uniform
D	Parquet	Uniform
E	Vertical	Lower Right
F	Parquet	Diagonal

Figure 8: Graph and initial rat position combinations used for performance benchmarking.

```
    -c       Clear expected result cache
    -t THD   Specify number of OMP threads
             If > 1, will run crun-omp.  Else will run crun-seq
```

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. When the option `-t 1` is given, it will test program `crun-seq`. For higher thread counts, it will run `crun-omp` with that number of threads. The default is 12. The benchmarking program also checks that the node counts generated by your simulator match those of the reference implemetantion. This will help you test larger cases, for which the Python simulator would be much too slow.

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 (90 points)

The provided program `benchmark.py` runs simulations of six different combination of graphs and initial rat positions, as documented in Figure 8. All simulations are run in batch mode. By default, the program runs with 12 threads for 100 simulation steps. It also runs the provided reference version of the simulator, named `crun-soln` on each benchmark. It captures the final output of the two simulators in files and then checks that they are identical.

The full set of options is as follows

```
linux> ./benchmark.py -h
./benchmark.py [-h][-Q] [-k K] [-b BENCHLIST] [-n NSTEP] [-u UPDATE]
  [-t THREADLIMIT] [-r RUNS] [-i ID] [-f OUTFILE] [-c]
    -h            Print this message
    -Q            Quick mode: Don't compare with reference solution
    -k            Specify graph dimension
    -b BENCHLIST  Specify which benchmark(s) to perform as substring of 'ABCDEF'
    -n NSTEP      Specify number of steps to run simulations
    -u UPDATE     Specify update mode
       r: rat order
       s: synchronous
       b: batch
    -t THREADLIMIT Specify number of OMP threads
       If > 1, will run crun-omp.  Else will run crun-seq.
    -r RUNS       Set number of times each benchmark is run
    -i ID         Specify unique ID for distinguishing check files
    -f OUTFILE    Create output file recording measurements
         If file name contains field of form XX..X, will replace with
         ID having that many digits
```

Here's a brief description of the options:

`-Q` Run in "quick" mode. Do not compare the functionality or performance to that of the reference solution, and do not generate a score.

`-k` k Specify the value of k. Possible values are: 12, 36, 60, and 180 (the default).

`-b` *BSTRING* Specify which of the benchmarks to run as a string consisting of characters in the range `A`–`F`. For example, the string "`DF`" would run the two parquet-graph benchmarks.

`-n` S Run the simulations for S steps. Running for small values of S provides a useful way to test your program without requiring a full length run.

`-u (r|b|s)` Run the simulator in the specified update mode.

`-t` T Run with T threads. When $T = 1$, the program `crun-seq` will be run. Otherwise `crun-omp` will be tested.

`-r` r Run each benchmark r times and take minimum of the execution times. The default is 3.

`-i` *`ID`* Specify a unique identifier for use in naming the check files (capturing the outputs of the two simulators.) This is used to allow multiple benchmarking runs to take place simultaneously without causing name collisions.

`-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 simulator to compensate for statistical variations in the program performance.

Suppose your program runs in time T and the reference version runs in time T_r. You will get 15 points if the output matches that of the reference version, and the runtimes satisfy $T_r/T \leq 0.9$. You will get partial credit as long as the outputs match and $T_r/T < 0.5$, i.e., your program is less than 2× slower than ours. The ratio T_r/T is included in the output report for each benchmark. 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.)

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.) Instead, copy your source code over to a subdirectory you have set up in your Latedays directory and recompile it.

The program `submitjob.py` is used to generate and submit command files to the job queue. It is invoked as follows:

```
linux> ./submitjob.py -h
Usage: %s -h -J -s NAME -a ARGS -r ROOT -d DIGITS
  -h         Print this message
  -J         Don't submit job (just generate command file)
  -s NAME    Specify command file name
  -a ARGS    Arguments for benchmark.py (can be quoted string)
  -r ROOT    Specify root name of benchmark output file
  -d DIGITS  Specify number of randomly generated digits in command and
             benchmark output file names
```

Here's a brief description of the options:

`-J` Generate the command file, but do not submit it.

`-s` *NAME* Specify the name of the command file. The default will name is of the form `latedays-`*DDDD*`.sh`, where *DDDD* is a sequence of random digits.

`-a` *ARGS* Provide arguments(s) for `benchmark.py`. Typically, *ARGS* is a quoted string. For example, specifying `-a '-t 6'` will cause the benchmarking to be done with 6 threads rather than 12.

`-r` *ROOT* Specify the prefix of the summary output file name. The output file is generally of the form *ROOT*-*DDDD*`.out`, where *DDDD* is a sequence of random digits.

`-d` d Specify the number of random digits to include in the command and output file names.

The inclusion of random digits in the file names provides a way to avoid naming collisions. For example, you can invoke `submitjob.py` n times with the same arguments, submitting n jobs, each with distinct file names.

After a successful submission, the program will echo the Id of your job. For example, if your job was given the number 337 by the job queue system, the job would have the Id `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. If the command file was `latedays-1234.sh`, then the generated files will be `latedays-1234.sh.o337` and `latedays-1234.sh.e337` (substituting your job number for 337, of course.) In addition, a summary of the results will be written to the output file.

Looking at the command file, you will see that the maximum wall clock time for the job is limited to 30 minutes. This means that the job scheduler will cut your program off after 30 minutes without generating any output.

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

- There are many possible sources of parallelism in the simulator, and you will want to exploit different forms at different stages. For example, sometimes you can run in parallel over the nodes, and other times in parallel over the rats.
- When you get a parallel version working, you may notice a big performance difference when running on a vertical versus a horizontal graph, even though one is just a rotated version of the other. Think about how the nodes are numbered (according to a row-major order) and what sort of load imbalance that can create.
- 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 to devise problem-specific ways of avoiding load imbalance.
- Having multiple threads attempt to update a set of global values will require costly synchronization. Instead, it can be better to have each thread generate data separately, and then use the threads to collaboratively aggregate the separate copies.

Important Requirements and Tips

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

- Use caution when evaluating performance by running on other machines. For example, the GHC machines are good ones to use for code development. For benchmarking, you will want to set the number of threads to either 6 or 12, to get a partitioning of nodes similar to what will happen when you run the benchmarks on the Latedays machines. However, the GHC machines have a higher performance memory system than the Latedays machines, which can potentially mask memory performance bottlenecks that will arise on a Latedays machine. Plan to do many runs on the Latedays machines.
- 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. The only code you cannot modify is in the file `rutil.c`. You must use the provided version of the function `imbalance`, which computes the imbalance factor β.
- 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. You can assume that the graphs of interest will consist of grids combined with a small number of high-degree hub nodes.
- The provided simulator has some features that you should maintain in your version:
  - It does not use any global variables. It encapsulates the graph structure and the simulation state into structs
  - It does not assume any limits on the number of nodes, rats, or any other parameters. All data structures are allocated dynamically.
  - It allocates all of the data structures it requires at the beginning of the simulation, and never requires dynamic allocation after that point.
- 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.
- 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. It should include 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 does the performance scale as you go from 1 (running `crun-seq`) 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, as well as the file `rutil.c` with the original versions and then perform regression testing and benchmarking.
   (c) Remove all nonessential files, especially output images from your directory.
   (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) Please upload your report in PDF format to Gradescope, with 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 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 9 shows simulations of the same graph—a 36×36 tiled graph 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 greater than 1.0 causes the rats to gather into clumps. As the simulation continues, these clumps become larger and more distinct. You can see the hubs tend to get higher counts than other nodes. Their high connectivity causes many rats to go to them, even though they are crowded.

Figure 10 shows simulations of three different 36×36 graphs, in each case with the rats distributed uniformly along the diagonal. The simulation of the tiled graph (a) is the same as in Figure 9(b). For the other graphs, the high connectivity of the hub nodes causes them to become highlighted. These hubs also cause the rats to quickly distribute among their regions. The hubs of the parquet graph (c) become highlighted and cause the rats to distribute among their regions, but there is less spreading from the regions in the upper left and lower right quadrants to those in the upper right and lower left quadrants.

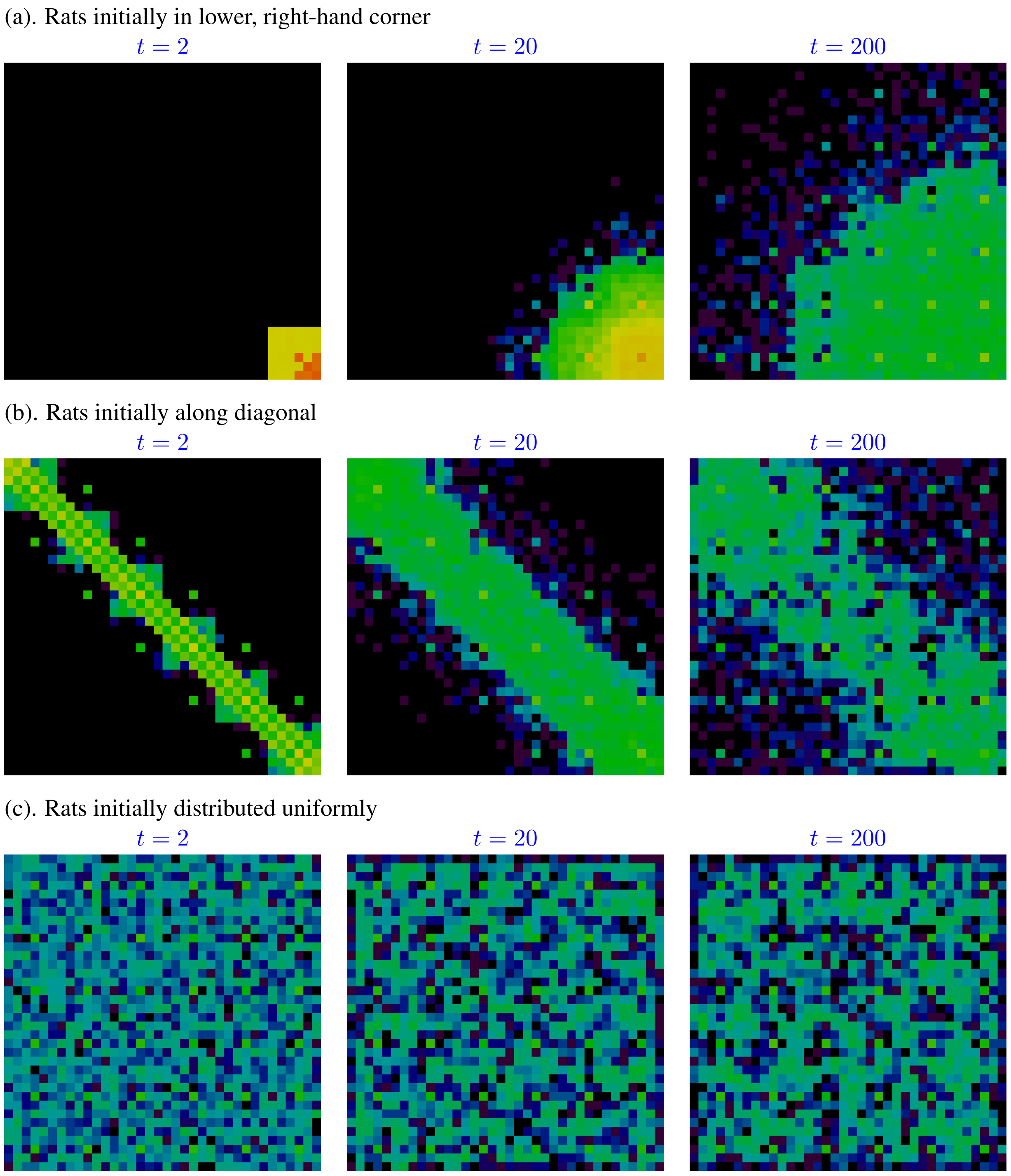

Figure 9: Simulations of a 36×36 tiled graph with different initial conditions ($\lambda = 10$)

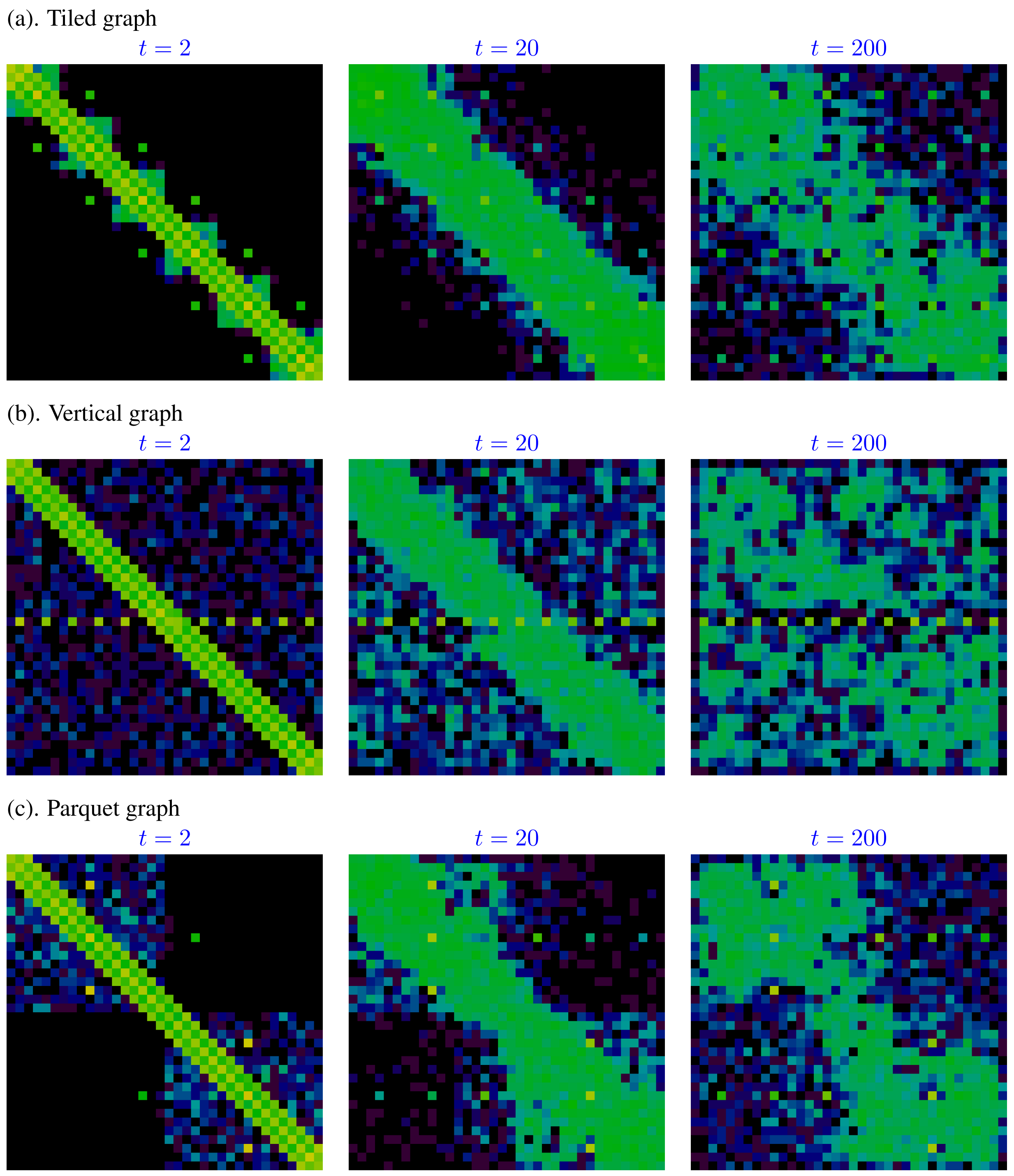

Figure 10: Simulations of different 36×36 graphs. Rats initially distributed along the diagonal ($\lambda = 10$)