Assignment 3: GraphRats
Topics

- Application
- Implementation Issues
- Optimizing for Sequential Performance
- Optimizing for Parallel Performance
- Useful Advice
Basic Idea

- **Transitions**
  - Each rat decides where to move next
    - Don’t like crowds
    - But also don’t like to be alone
  - Weighted random choice

- **Graph**
  - K X K grid

- **Initial State**
  - Start with all R rats in corner
Node Count Representation

$t = 0$

$t = 1.$

320
Simulation Example
Visualizations

Text ("a" for ASCII)

Heat Map ("h")

\[ t = 30 \]

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>14</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>16</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>16</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>
Running it yourself

```
linux> cd some directory
linux> git clone https://github.com/cmu15418/asst3-s18.git
linux> cd asst3-s18/code
Linux> make demoX
   X from 1 to 10
```

**Demos**

- 1: Text visualization, synchronous updates
- 2: Heap-map, synchronous updates
Determining Rat Moves

- **Count number of rats at current and adjacent locations**
  - Adjacency structure represented as graph

- **Compute reward value for each location**
  - Based on load factor $l = \text{count/average count}$
  - $l^*$ Ideal load factor (= 1.5)
  - $\alpha$ Fitting parameter (= 0.5)

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

\[ Reward(l) = \frac{1}{1 + \log_2 (1 + \alpha(l - l^*))^2} \]

- Maximized at load factor 1.5
  - Just above average population
  - Drops for smaller loads (too few) and larger loads (too crowded)
Reward Function (cont.)

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

- Falls off gradually
  - \( Reward(1000) = 0.0123 \)
Selecting Next Move

- Choose random number between 0 and sum of rewards
- Move according to interval hit
Update Models

- **Synchronous**
  - Demo 2
  - Compute next positions for all rats, and then move them
  - Causes oscillations/instabilities

- **Rat-order**
  - Demo 3
  - For each rat, compute its next position and then move it
  - Smooth transitions, but costly

- **Batch**
  - Demo 4
  - For each batch of B rats, compute next moves and then move them
  - \( B = 0.02 \times R \)
  - Smooth enough, with better performance possibilities
What We Provide

- **Python version of simulator**
  - Demo 4
  - Very slow

- **C version of simulator**
  - Faster, but still too slow
  - Demo 5: 8X8 grid, 320 rats
  - Demo 6: 160X160 grid, 1,024,000 rats
    - That’s what we’ll be using for benchmarks!
    - You’ll have to be patient using the starter code

- **Generate visualizations by piping C simulator output into Python simulator**
  - Operating in visualization mode
  - See Makefile for examples
Correctness

- **Simulator is Deterministic**
  - Global random seed
  - Random seeds for each rat
  - Process rats in fixed order

- **You Must Preserve Exact Same Behavior**
  - Python simulator generates same result as C simulator
  - Use `regress.py` to check
  - Don’t rely only on the small cases it currently checks
Graphs: Grid (Demo 6)

- $k \times k$ nodes, each with nearest neighbor connectivity
- Max degree = 4
- $k = 160$: 25,600 nodes 101,760 edges

Highly regular
Low degree
Rats spread slowly
Smaller d X d regions, each with hub node
- Node with edge to every other node in region
- Regions connected only by nearest-neighbor edges at border
- Maximum degree = $d^2 - 1$ (= 99 for benchmark graph)

- $k = 160$, $d = 10$: 25,600 nodes 152,850 edges
Fractal Graph (Demo 8)

- Regions of varying size
  - Some have 4 hub nodes, others 1
  - Maximum degree = $k^2/2-1$ ($= 12,799$ for benchmark graph)
- $k = 160$: 25,600 nodes 254,940 edges

Highly irregular
Widely varying degrees
Rats spread quickly through regions
Hub nodes typically have high rat counts
States (Fractal Graph)

Right Corner (r)
Demo 8

Diagonal (d)
Demo 9

Uniform (u)
Demo 10

\[ t = 0 \]

\[ t = 3 \]
Graph Representation

N node, M edges

neighbor \_\_start \ (length = N+1)

Having pointer to end is useful (why?)

neighbor includes self edges
length = N+M
Sample Code

- From sim.c
- Compute reward value for node

```c
/* Compute weight for node nid */
static inline double compute_weight(state_t *s, int nid)
{
    int count = s->rat_count[nid];
    return mweight(((double) count/s->load_factor);
}
```

- Simulation state stored in state_t struct
- Reward function computed by mweight
Sample Code

- From sim.c
- Compute sum of reward values for node

```c
/* Compute sum of weights in region of nid */
static inline double compute_sum_weight(state_t *s, int nid)
{
    graph_t *g = s->g;
    double sum = 0.0;
    int eid;
    int eid_start = g->neighbor_start[nid];
    int eid_end = g->neighbor_start[nid+1];
    int *neighbor = g->neighbor;
    for (eid = eid_start; eid < eid_end; eid++) {
        sum += compute_weight(s, neighbor[eid]);
    }
    return sum;
}
```
Sample Code

Compute next move for rat

```c
static inline int next_random_move(state_t *s, int r)
{
    int nid = s->rat_position[r];
    random_t *seedp = &s->rat_seed[r];
    double tsum = compute_sum_weight(s, nid);
    graph_t *g = s->g;
    double val = next_random_float(seedp, tsum);
    double psum = 0.0;
    int eid;
    int eid_start = g->neighbor_start[nid];
    int eid_end = g->neighbor_start[nid+1];
    int *neighbor = g->neighbor;
    for (eid = eid_start; eid < eid_end; eid++) {
        psum += compute_weight(s, neighbor[eid]);
        if (val < psum) {
            return neighbor[eid];
        }
    }
}
```
Sequential Efficiency Considerations

Consider move computation for rat at node with degree D
- How many (on average) iterations of loop in `next_random_move`?
- How many calls are made to `mweight`?

Suppose there are X rats in batch at single node with degree D
- How many (on average) iterations of loop in `next_random_move`?
- How many calls are made to `mweight`?
Finding Parallelism

- **Sequential constraints**
  - Must complete time steps sequentially
  - Must complete each batch before starting next

- **Sources of parallelism**
  - Over nodes
    - Computing reward functions
  - Over rats (within a batch)
    - Computing next moves
    - Updating node counts
Performance Targets

- Mega-Rats Per Second (MRPS)
  - R rats running for S steps
  - Requires time T
  - MRPS = \(10^{-6} \times R \times S / T\)

- Runs
  - 5 combinations of graph/initial state
  - Compute geometric mean of MRPS’s

- Target performance

<table>
<thead>
<tr>
<th></th>
<th>1 Thread</th>
<th>12 Threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>32</td>
<td>256</td>
<td>8.0</td>
</tr>
<tr>
<td>Batch</td>
<td>20</td>
<td>70</td>
<td>3.5</td>
</tr>
</tbody>
</table>
Some Suggestions

- **Focus initially on sequential performance**
  - But think of ways that will allow parallelism
  - Simple ideas / data structures generally work best
  - Use timing to guide optimizations

- **Synchronous mode easier to make fast**
  - Both sequentially and parallel

- **Machines**
  - Can develop on any machine
    - GHC machines work well
  - Performance will be evaluated on Latedays machines
    - Batch submission process
    - These have different characteristics from GHC machines when measuring parallel performance
Instrumenting Your Code

How do you know how much time each activity takes?
- Create simple library using cycletimer code
- Bracket steps in your code with library calls
- Use macros so that you can disable code for maximum performance

```c
START_ACTIVITY(ACTIVITY_NEXT);
#pragma omp parallel for schedule(static)
for (ri = 0; ri < local_count; ri++) {
    int rid = ri + local_start;
    s->rat_position[rid] = fast_next_random_move(s, rid);
}
FINISH_ACTIVITY(ACTIVITY_NEXT);
```
Evaluating Your Instrumented Code

1 thread

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup</td>
<td>194 ms</td>
<td>1.0%</td>
</tr>
<tr>
<td>compute_weights</td>
<td>2077 ms</td>
<td>11.1%</td>
</tr>
<tr>
<td>compute_sums</td>
<td>4029 ms</td>
<td>21.6%</td>
</tr>
<tr>
<td>find_moves</td>
<td>11733 ms</td>
<td>62.8%</td>
</tr>
<tr>
<td>set_ops</td>
<td>651 ms</td>
<td>3.5%</td>
</tr>
<tr>
<td>unknown</td>
<td>3 ms</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

+++ 25600 t 40 u 400 b 1 18.70 21.91

12 threads

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>startup</td>
<td>192 ms</td>
<td>3.2%</td>
</tr>
<tr>
<td>compute_weights</td>
<td>426 ms</td>
<td>7.0%</td>
</tr>
<tr>
<td>compute_sums</td>
<td>940 ms</td>
<td>15.5%</td>
</tr>
<tr>
<td>find_moves</td>
<td>3168 ms</td>
<td>52.3%</td>
</tr>
<tr>
<td>set_ops</td>
<td>1325 ms</td>
<td>21.9%</td>
</tr>
<tr>
<td>unknown</td>
<td>2 ms</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

+++ 25600 t 40 u 400 b 12 6.06 67.55 (3.08X)

- Can see which activities account for most time
- Can see which activities limit parallel speedup
Some Logos

GraphLab

GraphChi: Going small with GraphLab

GraphRats