15-418/618, Spring 2020
Assignment 4
GraphRats: MPI Edition

Assigned:	Wed., March 4
Due:	Sun., Mar. 29, 11:00 pm
Last day to handin:	Wed., Apr. 1

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-s20.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 library 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, such that they collectively solve a computational problem.

Mapping a problem across multiple processors often requires *partitioning* the data into parts that can be worked on independently. This assignment requires you to write a partitioner. You will find that doing so requires making trade-offs among competing objectives including: generality, performance of the partitioner, degree of load balance, and the resulting amount of communication required among processors.

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.

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 billions of 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

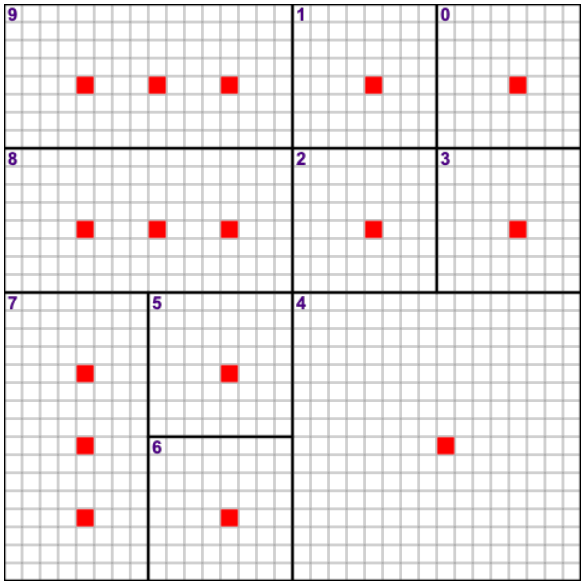

Figure 1: 32×32 Hilbert graph `hlbrtZ`. Regions are numbered so that consecutive regions in the ordering are adjacent in the graph.

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 running 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 P *zones*, mapping each zone onto a separate process (and relying on the OS to map each process onto a separate core.) Each process will keep track of the rats within its assigned zone, computing the new states of all of these rats. It will communicate with processes holding nodes adjacent to ones in its zone, both to share the node states and weights along the boundaries, and to pass along rats as they move from one zone to another. As before, the graphs consist of a number of regions, such that the only connections from one region to another are via the grid edges. As long as each region is fully contained within a single zone, the amount of communication between zones will be manageable.

Dr. Dent watched an online version of the 15-418 lecture of Feb. 7, 2020 and became fascinated by the properties of space-filling curves based on the Hilbert curve construction. He devised a method to generate fractal-graph representations of rat colonies, with the regions numbered in such a way that the consecutive regions in the numbering are adjacent in the graph. He calls these *Hilbert graphs*, an example of which is shown in Figure 1. "Rats are highly intelligent and will appreciate the elegance of this construction!" he exclaimed.

He devised a new set of benchmark graphs, documented in Appendix A. These have between 61 and 256 regions, with widely varying region sizes. One challenge for you will be to devise a partitioning scheme so that you can divide up each graph into P zones in a way that achieves a good combination of load balance and minimizes the required communication between zones.

Model Parameters

As before, each graph consists of N nodes with a set of directed and symmetric edges indicating which nodes are adjacent. All of the graphs are based on a grid of of $w \times h$ nodes. Some of the nodes are designated as "hubs," with edges to all other nodes in their respective regions. The graph file format has been extended to include the region data. The code for loading the graphs into memory calls a *partitioner*, which you must design, to partition the regions (and therefore the graph) into a specified number of zones.

Other aspects of the program (rats, reward functions) are the same as in Assignment 3. The code has been simplified to support only batch update mode. The starter code can be compiled to generate two different simulators: `crun-seq`, suitable for sequential execution, and `crun-mpi`, providing the starting framework for an MPI-based parallel simulator. Compiler-directives based on the compile-time constant `MPI` designate the differences between the two.

The provided MPI-based simulator does the following, when invoked to run with P processes:

1. The master process (Process 0), reads a copy of the graph file and runs your partitioner to assign each graph region to one of P zones.
2. It broadcasts the complete graph data structure to the other $P - 1$ processes.
3. Each process creates a set of data structures to represent its assigned zone.
4. The master process reads a copy of the rat file indicating the starting nodes.
5. The master process runs the entire simulation in sequential mode.

You must modify and extend this code to have the master distribute the rats after step 4 above. You should then replace step 5 by one that has the processes simulate their respective zones, exchanging rat and node information with each other. Periodically (only at the end of the simulation for the benchmark runs), the processes will be directed to provide their node counts to the master process, so that it can supply this information as the program output.

The simulator has a similar set of options as those in Assignment 3:

```
linux> ./crun-seq -h
Usage: ./crun-seq -g GFILE -r RFILE [-n STEPS] [-s SEED] [-q] [-i INT] [-I] [-z ZONE]
   -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
   -q        Operate in quiet mode.  Do not generate simulation results
   -i INT    Display update interval
   -I        Instrument simulation activities
   -z ZONE   Test partitioning into ZONE zones without running simulation
```

The `-z` option provides a way for you to test and evaluate your partitioner. It provides detailed information about the structure of the zones and then exits without performing any simulation.

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

To run a program under MPI, you use the program `mpirun`. A typical invocation is:

```
linux> mpirun -np 6 ./crun-mpi -g data/g-160x160-hlbrtA.gph -r data/r-160x160-r40.rats -n 5 -q
```

This has the simulator run with $P = 6$.

3 Test Programs and Performance Evaluation

The provided program `regress.py` has similar options as in Assignment 3, 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 P     Specify number of MPI processes
       If > 1, will run crun-mpi.  Else will run crun-seq
```

By default, it runs `crun-mpi` with 8 processes.

The provided program `benchmark.py` has the following options:

```
linux> ./benchmark.py -h
Usage: ./benchmark.py [-h] [-Q] [-I]
    [-b BENCHLIST] [-n NSTEP] [-p P1:P2:..:Pk] [-r RUNS] [-i ID] [-f OUTFILE]
    -h             Print this message
    -Q             Quick mode: Don't compare with reference solution
    -I             Include instrumentation results from each run
    -b BENCHLIST   Specify which benchmark(s) to perform as substring of 'ABCDEF'
    -n NSTEP       Specify number of steps to run simulations
    -p P1:P2:..:PK Specify number of MPI processes as a colon-separated list
       If > 1, will run crun-mpi.  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
```

The intention is that you run this program on either a GHC or a Latedays machine. On these machines, it will automatically invoke `mpirun` with a set of arguments that specify the use of *processor affinity*, a specific way to map processes onto cores. By default, it will run with 8 processes on a GHC machine and 12 on Latedays. By default, the program will run each simulation three times and take the minimum of their execution times. This helps make the timings more reliable. You can change this with the command-line option '`-r`.'

The guidelines for using the Latedays machines are the same as for Assignment 3. The provided program `submitjob.py` is used to generate and submit the control files to the job queue.

The performance will be evaluated on eight graphs, as is documented in Appendix A. Six of these are provided to you for testing and evaluation. The additional two, referred to as the *mystery graphs* will not be available to you. You must write your program to be sufficiently general to run these correctly and with good performance. Some general characteristics of these graphs are presented in Appendix A.

All benchmarking will be done using a random initial rat placement. Each run can be benchmarked against the provided program: either `crun-soln-ghc` (GHC) or `crun-soln-latedays` (Latedays). The actual grading will be done on Latedays. Performance points are computed as they were in Assignment 3, except that each benchmark will count up to 8 points, rounded up to the next 0.5 points, for a maximum total of 64 points.

4 Partitioning the Graph

You should modify the code in file `partition.c` to implement a better partitioner than the one provided. Figure 2 documents the API for the partitioner. The function `assign_zones` is provided a list of regions, the number of regions, and the target number of zones. Each region includes information about the number of nodes and edges[1] in the region, which can be used to assign a *weight* to each region.

There are a number of strategies for partitioning a graph, some of which are illustrated in Figure 3 for the 10-region graph of Figure 1. Let us consider the case where there are M regions, with the weight of each region i, denoted w_i, equal to the number of nodes in the region. In this example, the weights range between 64 and 256. The weight of each zone j, denoted W_j is then the sum of the weights of the regions assigned to the zone. One useful objective is to try to minimize the standard deviation of the zone weights, so that the weights of the zones are close to each other. Another is to try to minimize the amount of communication that will be required between the zones, including both 1) how many zones are adjacent to each zone and 2) how many edges there are between each pair of zones.

Figure 3 shows the result of applying three different partitioning strategies with $P = 3$ zones. Below each image is a list of zone weights of the form $W = (W_0, W_1, W_2)$. Figures 11–13 in Appendix B show the results of applying these three strategies with $P = 12$ zones to `hlbrtE`, one of the 160×160 graphs in the grading benchmark set (See Appendix A.)

Round Robin: Assign each region i to zone $i \bmod P$. This strategy is implemented in the provided code and in this case yields a very imbalanced partitioning (Figure3(a)) with between 256 and 448 nodes in the zones, and with a standard deviation of 79.8. Note also that this strategy does not make any consideration for the amount of communication. For `hlbrtE`, this strategy yields a standard deviation of 492.0.

Interval: This strategy is based on a *linear* partitioning of the region numbers. That is, suppose the regions are numbered from 0 to $M - 1$. An *interval partitioning* is a set of values $K_0, K_1, \ldots K_{P-1}$ such that $\sum_{j=0}^{P-1} K_j = M$. An optimal interval partitioning is one with minimum standard deviation. Given an optimal interval partitioning, zone 0 is then assigned the first K_0 regions, zone 1 then next K_1 regions, etc. Because of the region numbering scheme of our Hilbert graphs, each zone then consists of a contiguous set of nodes and these will tend to be grouped in a way that keeps the amount of communication required among the zones reasonably low. This strategy yields a slightly more balanced partitioning (Figure 3(b)) with between 256 and 384 nodes in each zone, and with a standard deviation of 60.3. For `hlbrtE`, this strategy yields a standard deviation of 339.9.

[1] The number of edges in a region is computed the sum of the number of neighbors for each node in the region.

(a) Region data structure

```
/* Representation of a region.  Used by partitioner */
typedef struct {
    int id;
    int x;  // Left X
    int y;  // Upper Y
    int w;  // Width
    int h;  // Height
    int node_count;  // Number of nodes
    int edge_count;  // Number of (directed edges)
    int zone_id;     // Zone assigned by partitioner
} region_t;
```

(b) Partitioner prototype

```
void assign_zones(region_t *region_list, int nregion, int nzone);
```

Figure 2: Partitioner API. When run with parameter `nzone` equal to P, The function `assign_zones` assigns a value between 0 and $P - 1$ to the `zone_id` field in each region.

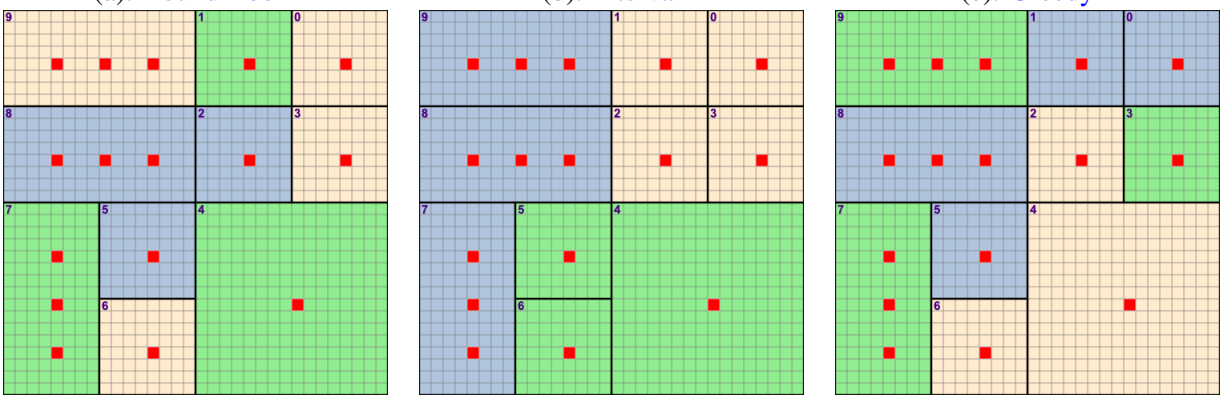

Figure 3: Partitioning of the 32×32 graph from Figure 1 into three zones according to different strategies.

Greedy: There are many strategies that can be classified as *greedy*. Ours uses the following simple approach: first, order all regions from highest weight to lowest. Then, at each step, assign the next region to the zone having the lowest cumulative weight. This strategy yields an optimally balanced partitioning for this example (Figure 3(c)), with between 320 and 384 nodes per zone and with a standard deviation of 30.2. However, it makes no consideration for the amount of communication. For `hlbrtE`, this strategy yields a standard deviation of 11.8.

You are free to use any of these strategies or to devise your own. You should also consider what value to use as the weight of a region: it could be based on the number of nodes, the number of edges, or some combination of the two. Your partitioner may attempt to minimize communication, or it may ignore this. The only requirements are:

- It should be completely general, running for any size graph, any number of regions, and any number of zones.
- The time required to run it will be counted as part of the start-up time for the simulator, and so you should avoid strategies that require a lot of computation.
- You may choose different strategies depending on the general characteristics of the graphs, but you should not "overfit" your selection to the six graphs provided. (Remember that you want good performance on the two mystery graphs.)

You will find some useful code in the file `rutil.c`, with API declared in `rutil.h`. In addition to some general statistical functions, there is an efficient interval partitioner, provided as the function `find_partition`. This function uses dynamic programming to find an optimum interval partitioning in time $O(M^2 \cdot P)$ and space $O(M \cdot P)$. Even for our largest benchmarks ($M = 256$, $P = 12$), this function runs in less than one second.

5 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.
- Although performance will be measured with just 12 processes, your program should be able to run on P processes for any value of P.
- 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 are to restrict your performance improvement techniques to ones that enhance or make better use of parallelism. Any optimizations that would improve the sequential simulator performance are *not* allowed. In particular:
  - You may not modify the function `neighbor_ilf` in `sim.c`. You also cannot use other methods to compute the ideal load factor (ILF) for a node.
  - You may not modify any of the simulation-related functions in `rutil.c` or use different versions of these functions. (You may add to or modify the statistics and optimization functions in this file.)
- 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..
- Although your simulator will only be tested on graphs of at most 40,000 nodes and 300,000 edges (see Appendix A), you should write your code to scale up to graphs of arbitrary size, arbitrary rat counts, and an arbitrary number of processes.

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 zones, it works well to have a process first initiate all of its send operations, then perform the receives, and then wait for the sends 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 initial rat positions from Process 0 (the master) to the others at the beginning of the simulation.

What is Provided

- You will find that the modifications you made to the starter code for Assignment 3 are not very useful here. You'd do better to work from the new starter code.

- The provided code stores a complete representation of the graph for each process. This uses more space than is necessary, but it allows you to have a universal 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.
- The provided code has each process construct data structures representing its assigned zone, stored as fields in the `graph_t` structure (declared in file `crun.h`). All lists of nodes are in sorted order.
  - Array `local_node_list` is a list of the nodes in the zone. Its length is given by the field `local_node_count`.
  - Array `export_node_list` is an array of P lists, where list `j` consists of the nodes in this zone that have edges to nodes in zone `j`. Its length is given by the field `export_node_count[j]`.
  - Array `import_node_list` is an array of P lists, where list `j` consists of the nodes in zone `j` that have edges to nodes in this zone. Its length is given by the field `import_node_count[j]`. Given the symmetry of the graph, you can assume that the contents of `export_node_list[j]` for process `i` is identical to those of `import_node_list[i]` for process `j`.

What You Need to Do

- You will find comments in some of the `.h` and `.c` files with the header "`TODO`." These indicate some of the key places you will need to add or modify the existing code.
- You will need to modify the partitioning code in the file `partition.c` to improve on the provided partitioner. This is discussed more fully in Section 4.
- You will need to allocate space to store information about the rats in each zone, as well as the buffers you use for communication via MPI. Generally, it is best to allocate these at the beginning of the program. Some you can allocate according to the maximum required size. Others you may want to allocate smaller amounts and then grow dynamically (via `realloc`) as needed.
- You will need to understand the processing steps in the function `do_batch` (file `sim.c`) and adapt them for use on a single zone. This will require several rounds of exchanging data with adjacent zones: rats, node counts, and node weights.
- When sending a rat to a new zone, you must also send along its associated seed for random number generation.
- You will need to implement the capability to have every process send its copy of the node counts to Process 0, and for Process 0 to collect these counts from other processes. These should be implemented as functions `send_node_state` and `gather_node_state`, respectively (file `simutil.c`.)
- In order to use the instrumentation features, you will need to include the `START_ACTIVITY` and `FINISH_ACTIVITY` in your code with the appropriate designation of the activity. Use `ACTIVITY_COMM` for communication between processes to pass zone data and `ACTIVITY_GLOBAL_COMM` for special communication with process 0.

How to Optimize the Program

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 zone. 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.
- As a combination of the two: use a bit vector to quickly determine membership and use a list to be able to enumerate all elements in the set.

Other Tips

- The data files include several small graphs: two of size 4×4, and the other of size 12×12, along with associated rat position files. These can be useful when doing detailed debugging.
- You can do direct comparisons of two versions of your code by renaming one of the executables to be either `crun-soln-ghc` or `crun-soln-latedays` (depending on which class of machine you're using). Be sure to keep the original copy of this program, of course. You will find that doing side-by-side comparisons within a single run is the best way to compensate for the high run-to-run variability on the Latedays machines.

Using Instrumentation to Optimize Performance

When you run the simulations with the `-I` option, the program will report how much time is required for each part of the computation for each process. You will find this useful for identifying the critical performance points in your program, as well as any variations among the processes.

As examples, Figure 4 shows the instrumentation results for two simulations of the graph `hlbrtF` using the interval (a) and greedy (b) partitioning schemes. Looking at them individually and comparing them to each other is very instructive. In both cases, times are reported in milliseconds.

With interval partitioning, the `compute_weights` portion dominates the overall time and also has a high standard deviation. Indeed, we see a high standard deviation in both the number of nodes and the number of edges, as well. In combination, these indicate and explain a high degree of imbalance among the processes. The `local_comm` (communication between processes) also has a high standard deviation. One might think it was caused by actual transmission delays, but note how it tends to vary in the opposite direction as `compute_weights`. That indicates that much of this time is the result of lightly loaded processes waiting for heavily loaded ones to complete. Again, the core problem is a load imbalance. The total elapsed time for process 0 is around 2.66 seconds, and this determines the overall execution time for the benchmark.

With greedy partitioning, we see that all of the standard deviations are much lower—the load is well balanced. The total time is around 2.20 seconds, significantly less than with interval partitioning.

These measurements show that having an even load balance is more important than minimizing communication for this benchmark, but that may not be the case across all benchmarks.

(a) Interval partitioning

Zone	0	1	2	3	4	5	6	7	8	9	10	11	Max	Mean	StdDev	
Nodes	1600	3200	3200	1600	1600	2400	2400	1600	2200	1800	2000	2000	3200	2133.3	555.8	
Edges	14264	22340	22300	17448	15842	19924	19914	15664	17572	16400	19406	16896	22340	18164.2	2494.7	
	106	9	3	3	9	4	3	3	4	4	4	3	106	12.9	28.1	unknown
	249	266	260	260	293	261	261	261	261	261	261	260	293	262.8	9.8	startup
	867	1486	1475	983	1052	1233	1232	911	1103	981	1136	1034	1486	1124.4	192.2	compute_weights
	54	88	90	62	86	74	75	56	71	62	75	63	90	71.3	11.7	compute_sums
	340	568	580	352	424	470	476	354	473	409	459	407	580	442.7	74.7	find_moves
	1040	195	202	952	748	569	562	1025	699	893	676	843	1040	700.3	271.5	local_comm
	0	0	0	0	0	0	0	0	0	0	0	0	0	0.0	0.0	global_comm
Elapsed	2660	2613	2613	2613	2613	2613	2613	2613	2613	2613	2613	2613	2660	2616.9	13.0	

(b) Greedy partitioning

Zone	0	1	2	3	4	5	6	7	8	9	10	11	Max	Mean	StdDev	
Nodes	2200	2400	2400	2400	2400	2000	1900	1900	2000	2000	2000	2000	2400	2133.3	201.4	
Edges	17672	17464	17454	17454	17454	18622	18716	18706	18612	18612	18602	18602	18716	18164.2	565.5	
	142	5	5	5	5	6	5	6	6	5	5	5	142	16.7	37.8	unknown
	251	262	262	262	262	263	263	263	262	262	262	262	263	261.3	3.1	startup
	1124	1138	1137	1136	1139	1113	1092	1098	1116	1109	1108	1111	1139	1118.4	15.5	compute_weights
	73	73	75	76	76	75	73	76	79	75	78	80	80	75.8	2.2	compute_sums
	471	485	480	485	486	461	438	449	474	470	470	475	486	470.3	14.1	find_moves
	134	187	192	187	182	233	279	259	213	228	227	217	279	211.5	36.8	local_comm
	0	0	0	0	0	0	0	0	0	0	0	0	0	0.0	0.0	global_comm
Elapsed	2199	2153	2153	2153	2153	2153	2153	2153	2153	2153	2153	2153	2199	2156.8	12.7	

Figure 4: Instrumentation Examples. Both are for simulations of graph `hlbrtF` but with different partitioning schemes

6 Your Report (32 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. You should document both the design and performance of your partitioning code, as well as the design and performance of the simulator.

Your report should include a detailed discussion of the design and rationale behind your approach to parallelizing the algorithm. Specifically, try to address the following questions:

1. What sequence of computations and communications is performed for each batch?
2. How did you maximize the decoupling of processes to avoid waiting for messages from each other.
3. How successful were you in getting speedup in your program? (This should be backed by experimental measurements.)
4. How does the performance scale as you went from 1 to 12 processes?
5. How important is having an even load balance vs. minimizing communication for the different benchmarks?
6. Were there any techniques that you tried but found ineffective?

7 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 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 Benchmark Graphs

The following table provides the statistics for graphs that will be used in benchmarking your simulator. Graphs `hlbrtA`–`hlbrtF`, for which you are provided copies, all have width $w = 160$ and height $h = 160$, for a total of 25,600 nodes. Graphs `hlbrtG` and `hlbrtH` are the mystery graphs that will be used as benchmarks but for which you do not have access. Here we show upper limits on their size parameters. The columns labeled "Min. Region" and "Max. Region" indicate the smallest and largest number of nodes in a region, respectively.

Graph	Nodes	Edges	Regions	Hubs	Min. Region	Max. Region
`hlbrtA`	25,600	150,400	256	256	100	100
`hlbrtB`	25,600	193,834	100	202	100	800
`hlbrtC`	25,600	203,606	150	308	25	800
`hlbrtD`	25,600	193,572	80	156	50	1,600
`hlbrtE`	25,600	199,188	74	138	25	1,600
`hlbrtF`	25,600	192,370	61	121	100	1,600
`hlbrtG`	$\leq$ 40,000	$\leq$ 300,000	$\leq$ 250	$\leq$ 400	$\geq$ 100	$\leq$ 2,000
`hlbrtH`	$\leq$ 40,000	$\leq$ 300,000	$\leq$ 250	$\leq$ 400	$\geq$ 25	$\leq$ 2,000

The following figures show the structure of the first six graphs.

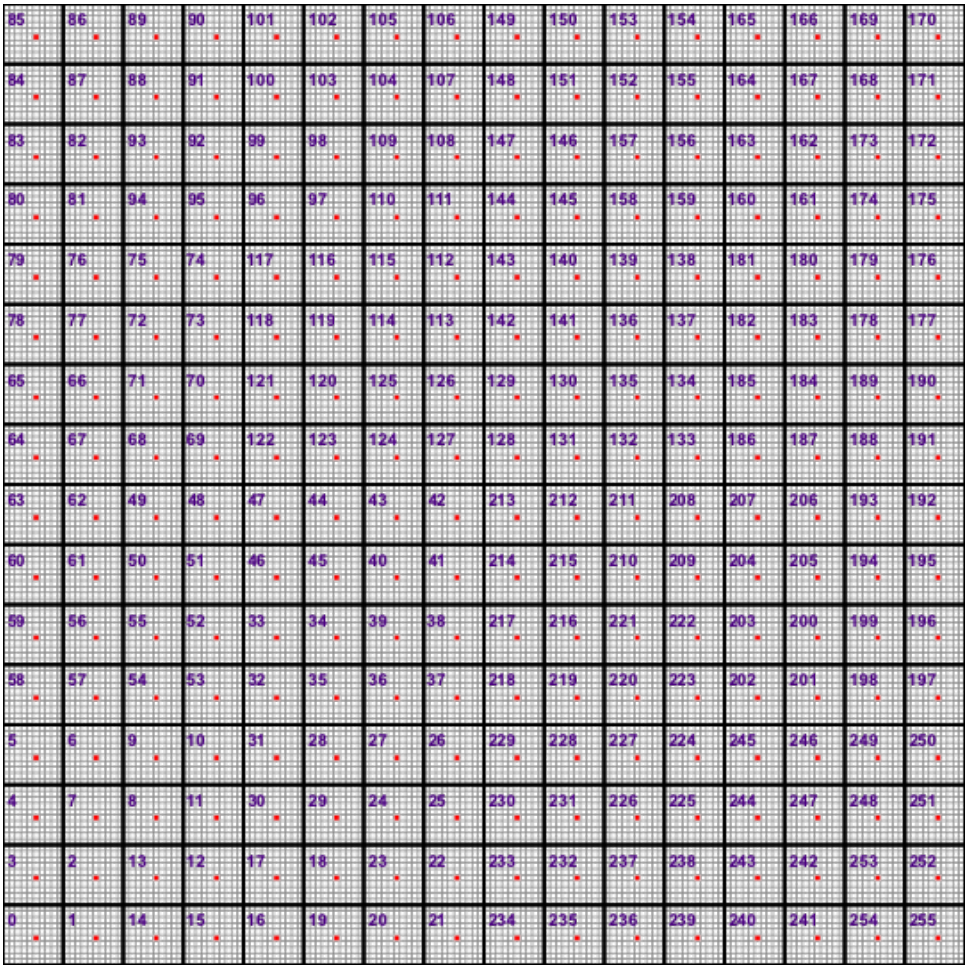

Figure 5: Graph `hlbrtA`

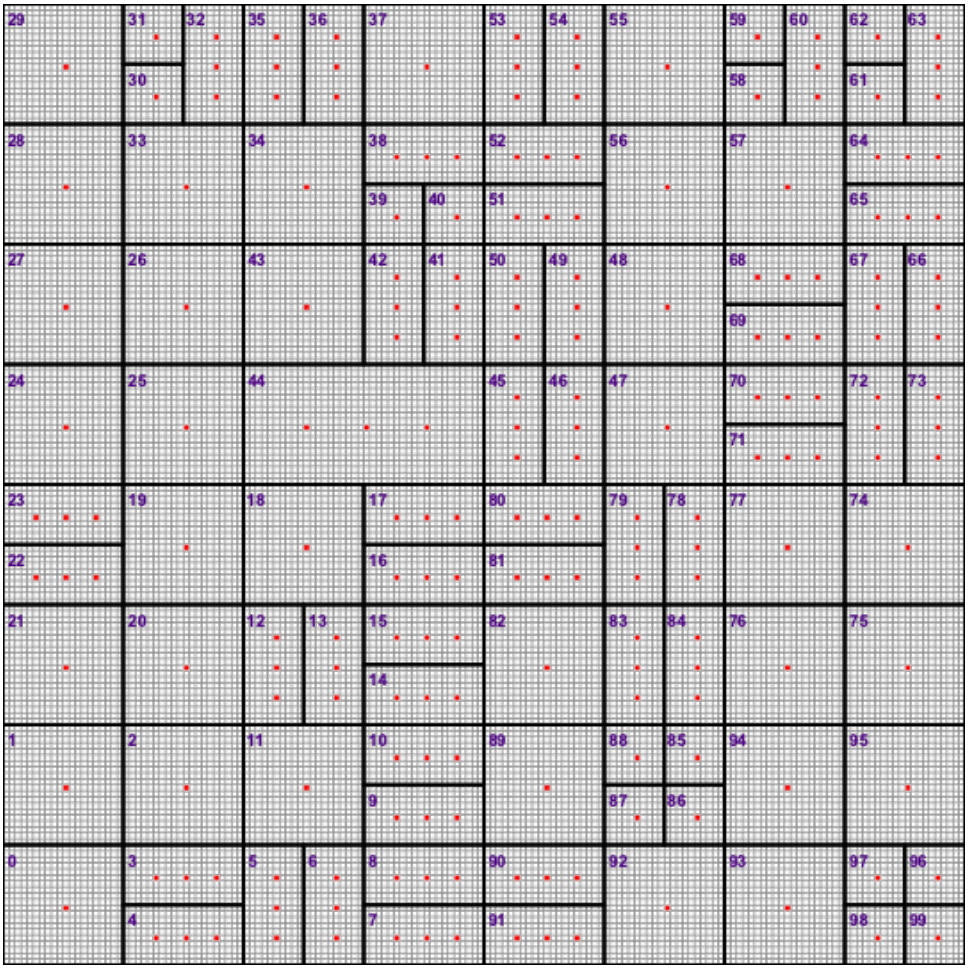

Figure 6: Graph `hlbrtB`

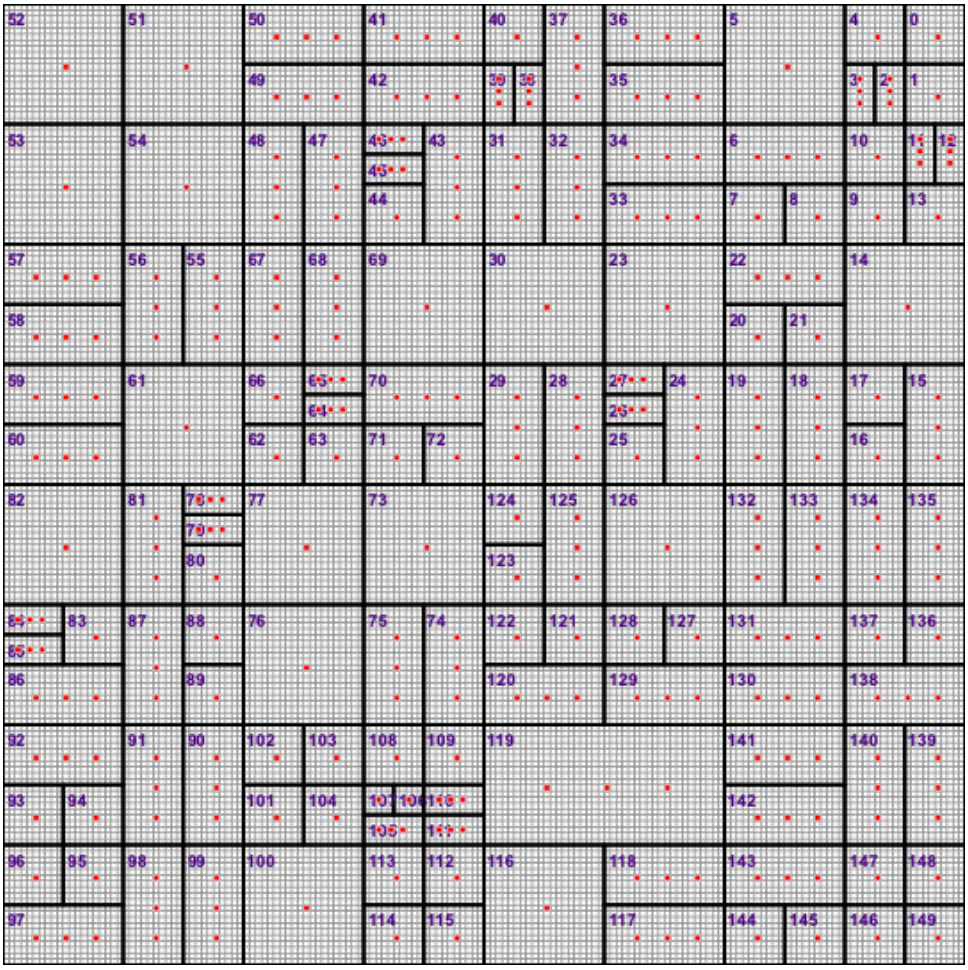

Figure 7: Graph `hlbrtC`

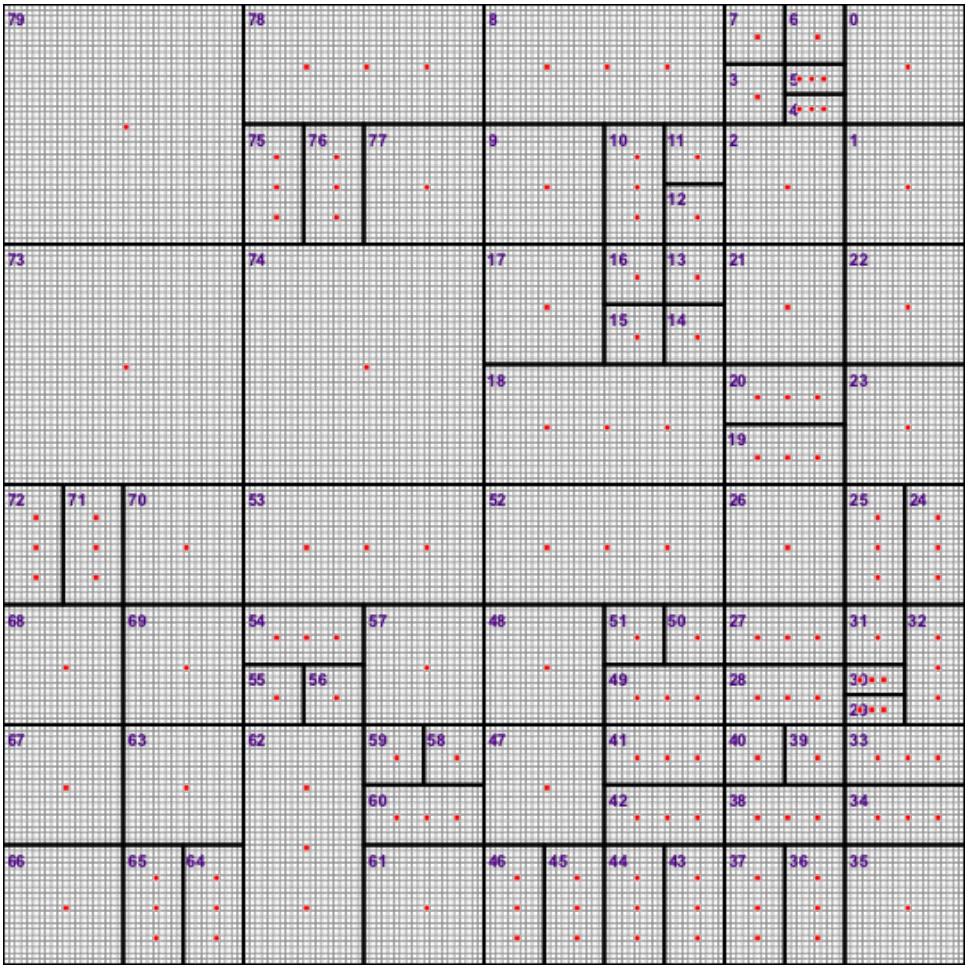

Figure 8: Graph `hlbrtD`

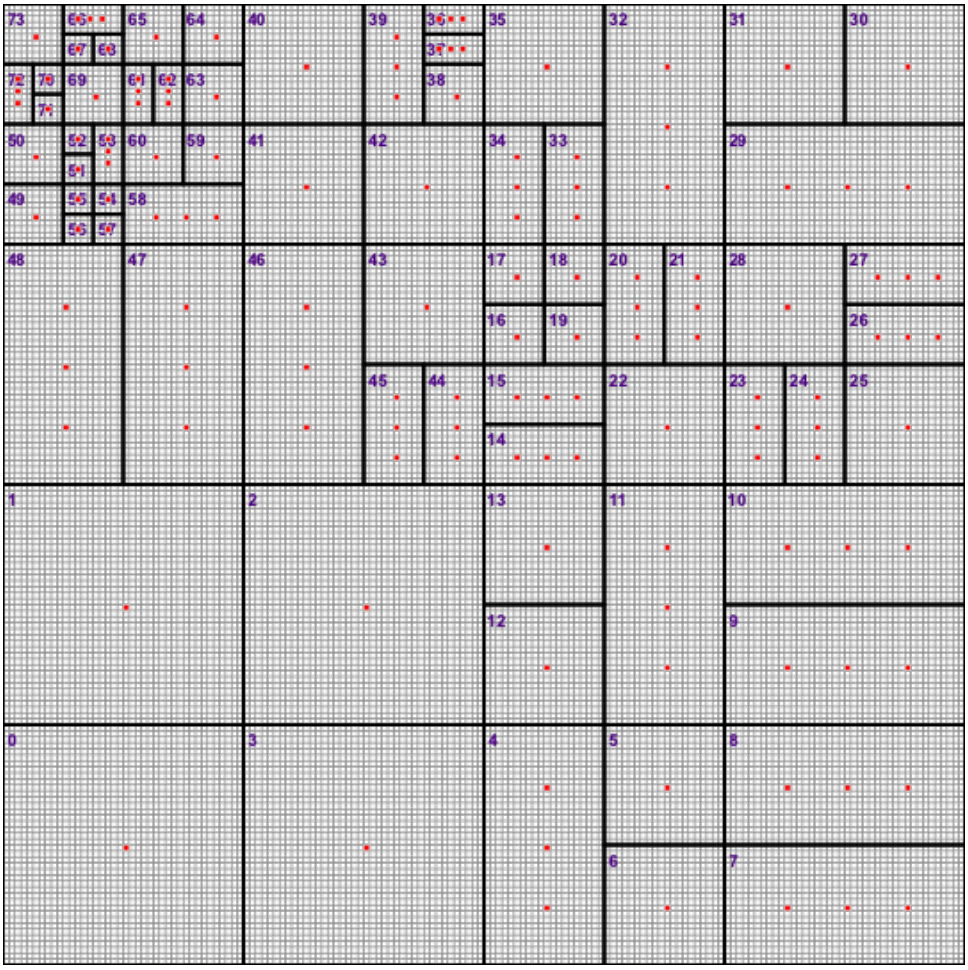

Figure 9: Graph `hlbrtE`

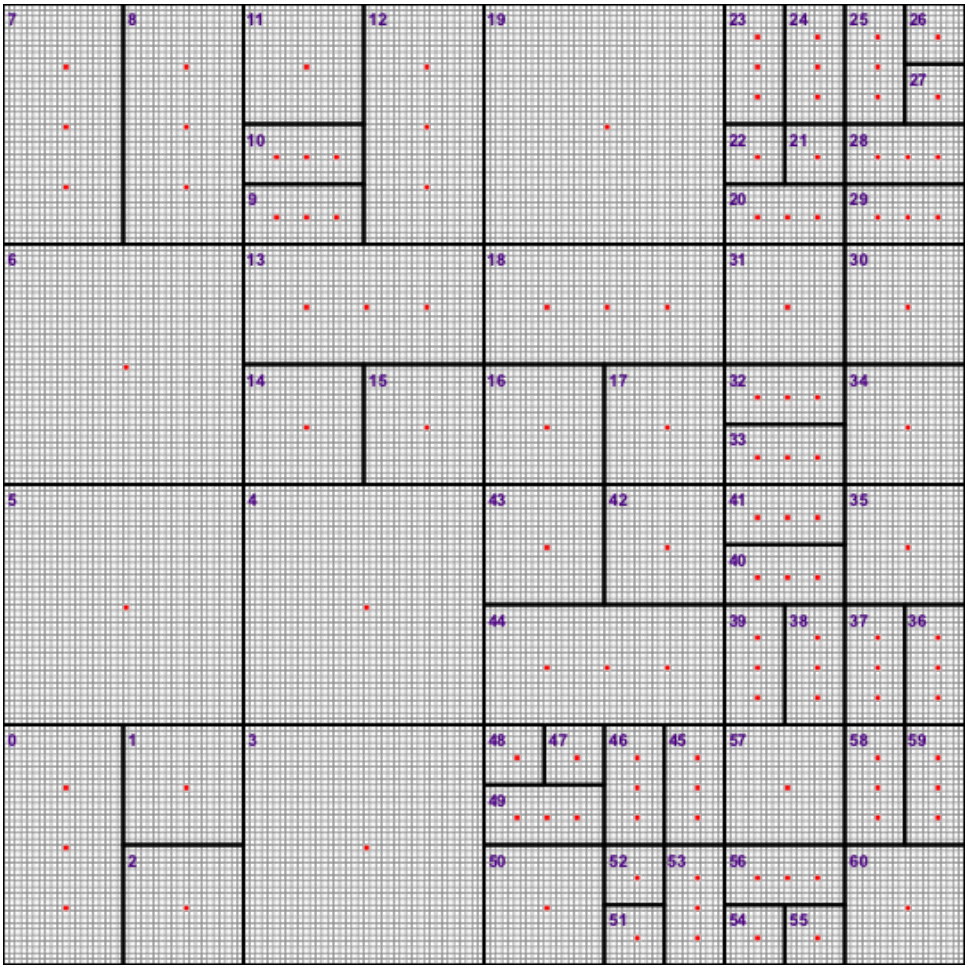

Figure 10: Graph `hlbrtF`

B Graph Partitioning

The following figures illustrate three different partitionings of graph `hlbrtE` into 12 zones, with the cost function based on the number of nodes in each zone.

Method	Min.	Mean	Max.	Std. Dev
Round Robin	1375	2133.3	3200	492.0
Interval	1600	2133.3	2400	339.9
Greedy	2125	2133.3	2150	11.8

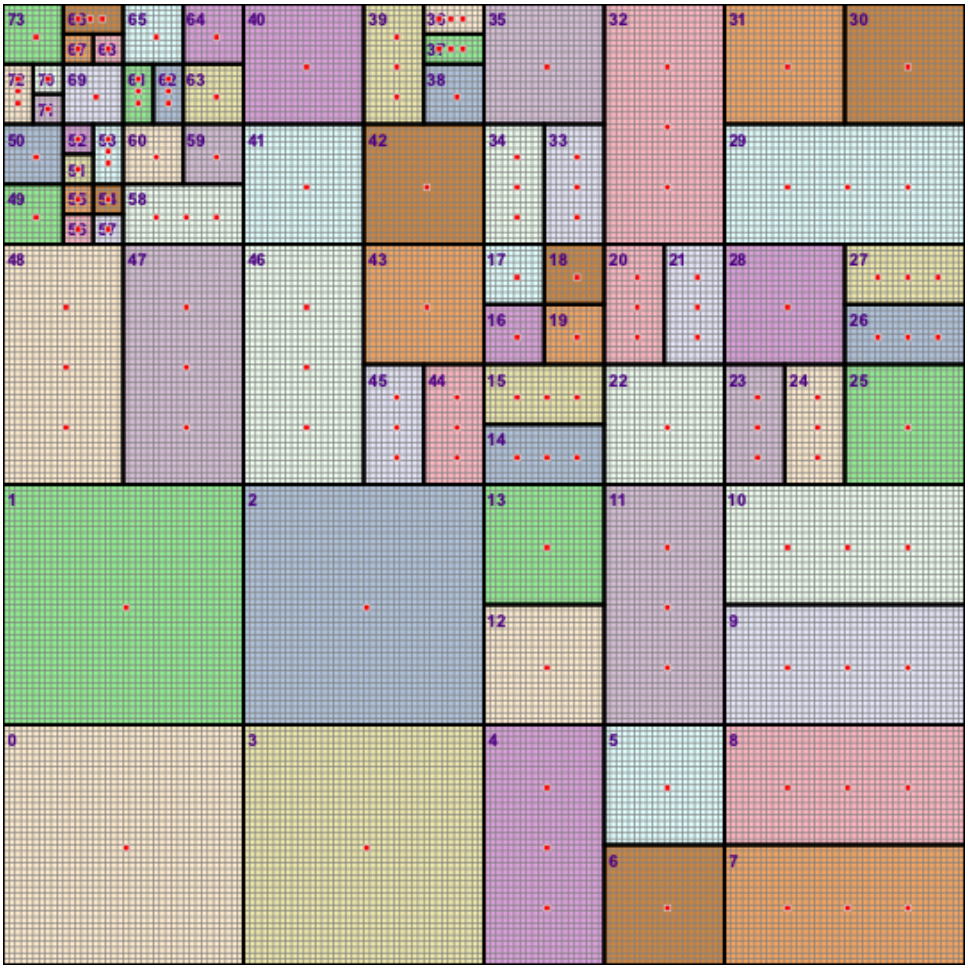

Figure 11: Partitioning of graph `hlbrtE` into 12 zones using the round-robin strategy.

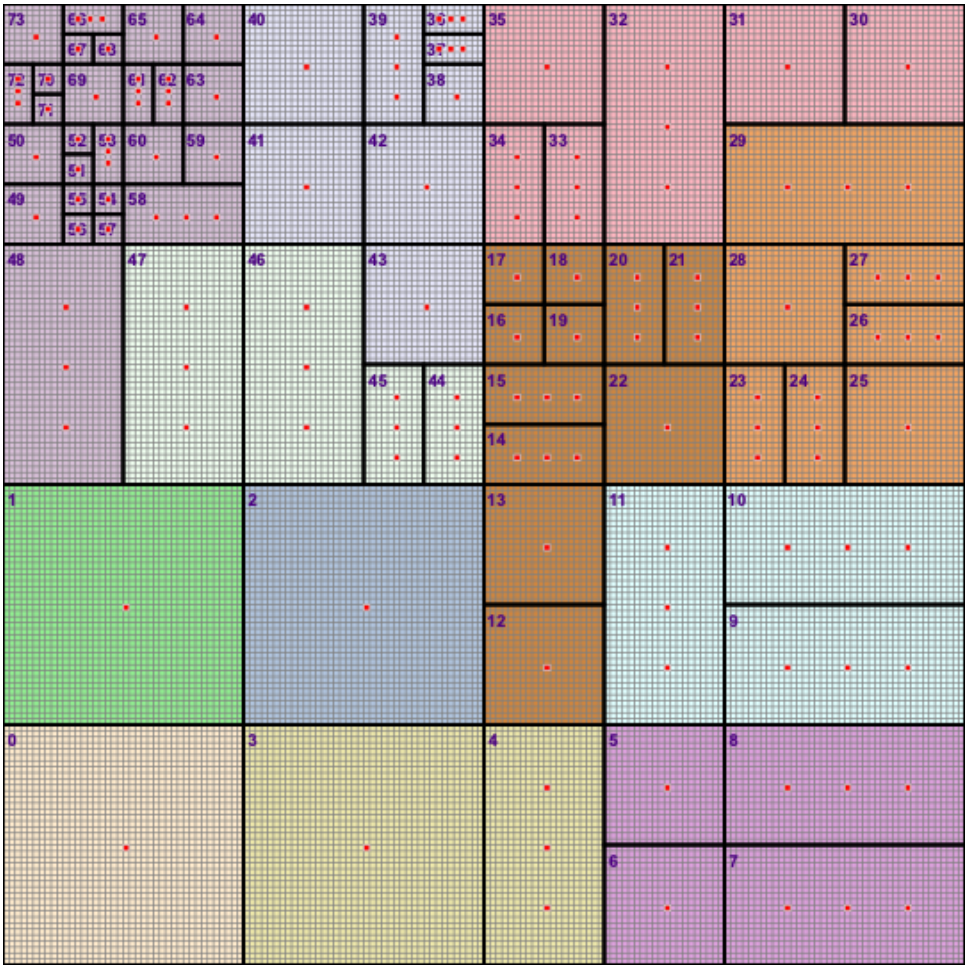

Figure 12: Partitioning of graph `hlbrtE` into 12 zones using the interval strategy.

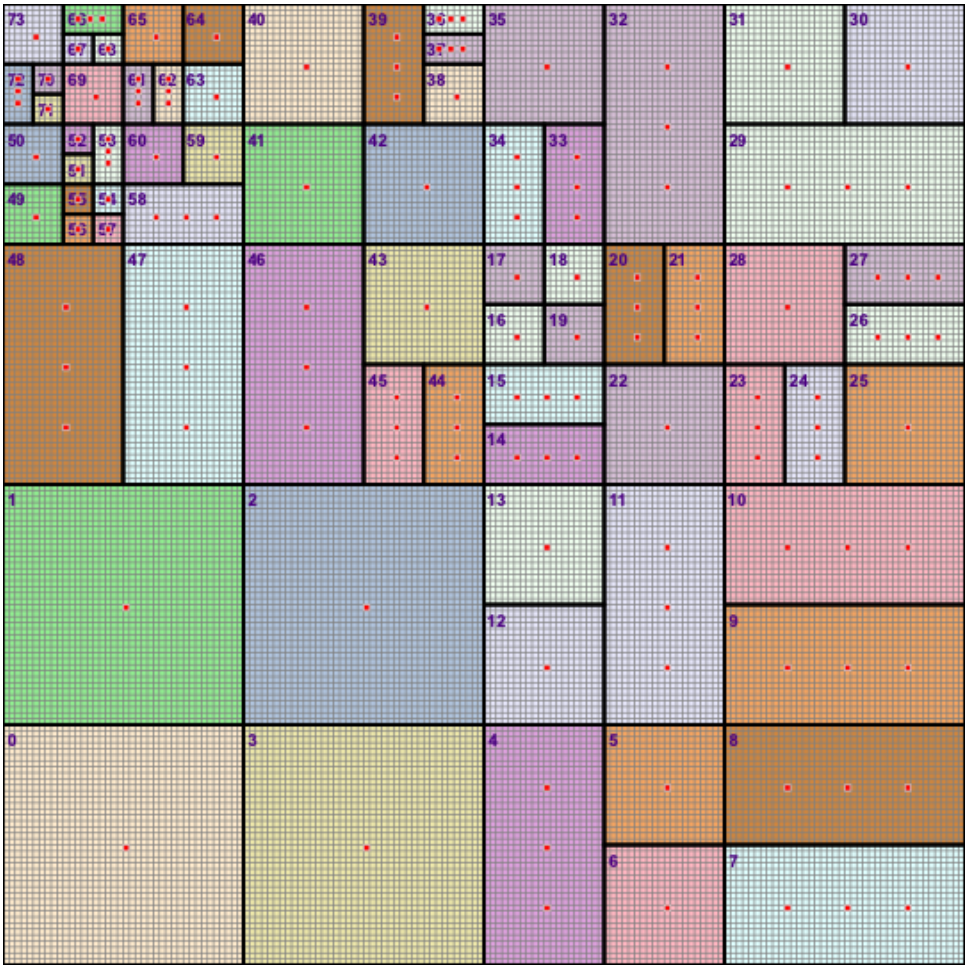

Figure 13: Partitioning of graph `hlbrtE` into 12 zones using a greedy strategy.