Programming Assignment #3

CS–735/835

due 26 Mar 2017 11:59 PM

1 Introduction

This assignment focuses on the design of thread-safe classes using the principles studied in chapter 4, the thread-safe data structures and synchronizers of chapter 5, the use of thread pool executors and the performance gained from parallelism.

The application to implement is a parallel simulator for networks-on-chips. Networks-on-chip are networks with a regular topology that route messages among several cores on a chip, using simple rules. Networks consist of cores, routers and wires.

In this assignment, routers are laid out on a 4-regular grid (each router has 4 neighbors) with a torus topology (the right edge of the grid connects back to the left edge; the bottom of the grid connects back to the top). Wires are unidirectional. Horizontal wires transport data from left to right; vertical wires transport data from top to bottom. Thus, each router has 2 incoming wires (North and West) and two outgoing wires (East and South). A router maintains 4 ports (one for each wire) that contain at most one (incoming or outgoing) message each (no queues). Fig. 1 shows a 4×4 network and the trajectory followed by a message from A to B.

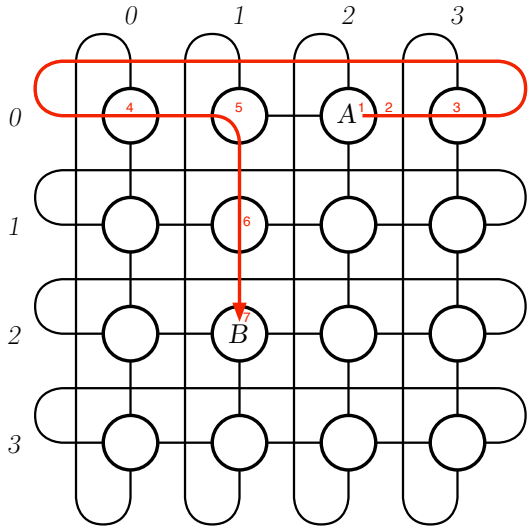

Figure 1: A 4×4 network.

Each router is associated with a core that acts both as a source and as a destination (it sends and receives messages). The core generates new messages at various times and stores them in an outgoing queue, which is unbounded. The router consumes messages from this queue and injects them into the network. Routers also receive messages from other routers and either deliver them if their core is the destination, or continue to route them to an appropriate router using a routing algorithm described below.

Messages are represented as instances of a class `Message`. A message consists of a source and a destination (row and column) and two timestamps (the times at which the message was sent and received). Note that a message is first timestamped when it is created by a core and enters its queue (i.e., when the core needs to send it), not when it enters a router (since a message might spend time in the core queue). Messages can be marked as "tracked": Simulators display information about tracked messages as they traverse the network but move other messages silently. Marking some messages as "tracked" can help with debugging.

Networks-on-chips are mostly synchronous: All the routers route together; all the wires transfer data together, etc. However, cores can be working on a different clock (they are usually faster). To keep things simple, this assignment assumes that one round of routing, wire transfers and core activities takes exactly one time step. However, to simulate the fact that cores can be faster, they can generate more that one message in a single step. To keep track of the time steps, all the components of a network share a single clock. Each component of a network (router, core, wire) is implemented as a type with a principal public method:

- `Router.route()` implements the routing algorithm below.
- `Wire.transfer()` transfers data from router to router.
- `Core.process()` simulates the core computation, which generate new messages.

The routing algorithm consist of the following steps, in order:

1. If the North port contains a message intended for the core, it is delivered.
2. If the West port contains a message intended for the core, it is delivered.
3. If the North port contains a message *and* the South port is free, the message is moved from North to South.
4. If the West port contains a message *and* the East port is free *and* the message is intended for a core in a different column, the message is moved from West to East.
5. If the West port contains a message *and* the South port is free *and* the message is intended for a core in the same column, the message is moved from West to South.
6. The final steps examines the message at the front of the core's outgoing queue, if any:
   - 6a. If the message is intended for the core itself, it is delivered and the routing step is complete.
   - 6b. If the message is intended for a core in the same column *and* the South port if free, the message is moved from the core queue to the South port and the routing step is complete.
   - 6c. If the message is intended for a core in a different column *and* the East port if free, the message is moved from the core queue to the East port and the routing step is complete.

Note that this routing algorithm is deterministic: It specifies how to resolve all conflicts (several messages trying to go through the same port). Specifically, new messages yield to turning messages, which yield to non-turning messages. Note also that a router only consumes at most one message from the core queue (i.e., if step 6a is fired, steps 6b and 6c are skipped). Finally, note that a router can deliver a message to the core that originated it (step 6a) without using any port.

A simulation step proceeds as follows:

- the network clock ticks;
- *then*, `route()` is called on *all* the routers;
- *then*, `transfer` is called on *all* the wires;
- *and*, `process` is called on *all* the cores.

```
List<Core> cores = network.allCores();
List<Router> routers = network.allRouters();
List<Wire> wires = network.allWires();

do {
  clock.step();
  for (Router router : routers)
    router.route();
  for (Wire wire : wires)
    wire.transfer();
  for (Core core : cores)
    core.process();
} while (network.isActive());
```

Listing 1: Sequential simulation.

A sequential (single-thread) simulator could perform such steps with the code from Lis. 1. Code for a complete single-threaded simulator can be found in class `SeqSimulator`, the implementation of which is given.

In this assignment, we want to implement parallel simulators. In a parallel simulator, it is important that routers are processed before wires and cores to ensure that all simulators produce the same results deterministically. Furthermore, is *essential* that *all* the routers have been processed before the wires are processed so there is no race condition at the level of a port: A thread could write into a port while another thread reads from it, which would produce unpredictable behavior. Similarly, is it *essential* that *all* cores and wires have been processed before the next step of routing is started. Wires and cores can be processed concurrently.

2 `NaiveExecSimulator`

A first strategy is to rely on a thread pool with the desired number of workers and to submit routing, wire transfer and core processing tasks to this pool. Class `NaiveExecSimulator` implements such a strategy. If N is the size of the network and k is the number of workers, the simulation submits N routing tasks to the k workers; then, *after those tasks are completed*, it submits $2N$ wire transfer tasks and N core processing tasks to the k workers; then, *after those tasks are completed*, the clocks is stepped and the process continues with more routing tasks, unless the simulation is complete.

Completion is decided using method `Network.isActive`, which is quite costly (it needs to potentially examine all the routers and all the cores) and does not attempt to build a consistent snapshot (and is therefore unreliable while threads are actively running the network). `NaiveExecSimulator` uses a "master thread" that:

- increments the global clock at the beginning of each simulation step;
- starts all the routing tasks;
- waits for all the routing tasks to complete before starting the wire and core tasks;
- waits for the wire and core tasks to complete before checking for termination using method `Network.isActive`;
- begins the next simulation step if needed.

3 `ExecSimulator`

One issue with the `NaiveExecSimulator` implementation is the number (and granularity) of tasks: The simulation of a 100×100 network needs to submit 40000 tasks with each simulation step, each one fairly small.

Class `ExecSimulator` implements a potentially better simulator in which a parameter n controls the number of tasks created in each stage of the simulation. In the first stage, n tasks route at most $\lceil N/n \rceil$ routers each. In the second stage, n tasks process at most $\lceil N/n \rceil$ cores each and $2n$ tasks process at most $\lceil N/n \rceil$ wires each. The core and wire tasks are run concurrently. When n is the size N of the network, this implements the strategy used in `NaiveExecSimulator`. Presumably, smaller n values can result in better granularity and improved performance.

4 Implementation

In terms of concurrency, although the network is being used by multiple threads, the staged nature of the simulation ensures that a core does not have its `process` method called by different threads at the same time, a router does not have its `route` method called by different threads at the same time, a wire does not have its `transfer` method called by different threads at the same time, a thread does not deliver a message to a core while another thread is processing the core, etc. Therefore, it is possible to reduce synchronization is some places (e.g., use volatile variables, or even non-volatile variables if the JMM is well understood). Keep in mind, however, that the assignment is graded on correcteness and experimental study of performance, not on the performance itself. Be safe before you try to be efficient.

Independently from concurrency issues and thread-safety considerations, the implementation of the `Network` class can be a bit tricky. The purpose of the class is to create all the core, router and wire objects and to connect them together. This can be done as the network is constructed. One difficulty, however, is that core objects need to have access to the network's clock (to timestamp messages) and so do router objects (to print information about tracked messages), but the network's clock is not known at construction time (it is set later by a simulator based on the simulator's needs). Therefore, core and router objects need to keep a reference back to the network object to access the clock. An easy way to achieve this is to use member classes within the `Network` class to implement cores and routers (the class that implements wires can be a regular class or a static inner class).

5 Report

The report should discuss at least the following questions:

- Discuss the thread safety of the main objects used in a simulation. In particular, what methods of `Core`, `Router` `Wire` and `Clock` are thread-safe and can be called by *any* thread at *any* time? When a method is *not* thread-safe but ends up being called by multiple threads, what mechanisms are used to ensure correct behavior?
- Create a scenario with two messages in which a turning message (from West to South) is blocked by a message that moves vertically (from North to South). Simulate it (using a sequential or a parallel simulator) with both messages marked, and describe step by step the output from the simulator.
- Compare and contrast the performance of `SeqSimulator`, `NaiveExecSimulator` and `ExecSimulator` when used on a parallel machine (at least 6 cores without hyperthreading). Note that several variables control these experiments:
  - network size (the number of cores);
  - simulation size (the number of messages sent and received);
  - number of worker threads (in the case of parallel simulators);
  - granularity (in the case of `ExecSimulator`).

Of particular interest is the following question: For a given scenario (network size and simulation size), what is the optimal number of workers and granularity to be used by `ExecSimulator`?

Notes:

- Public classes to be implemented are described at: `http://www.cs.unh.edu/~cs735/Programming/`. Public interfaces cannot be modified (i.e., you cannot add or remove a public or protected method, constructor, field or class, nor modify the checked exceptions thrown by a public or protected method or constructor). You can add private and package-private classes, methods, fields, etc.
- Here is a small example of a single message going from A to B through the 4×4 network of Fig. 1:

```
% cat /tmp/times1
1 (0,2) (2,1) 1*
% sbt --warn 'runMain cs735_835.noc.ExecSimulator 4 4 file:/tmp/times1 2 2'
at 1, new msg 1 generated by (0, 2) intended for (2, 1)
at 2, (0, 2) starts msg 1 towards East
at 3, (0, 3) moves msg 1 from West to East
at 4, (0, 0) moves msg 1 from West to East
at 5, (0, 1) moves msg 1 from West to South
at 6, (1, 1) moves msg 1 from North to South
at 7, msg 1 is delivered to (2, 1)
simulation completed: 0.01 seconds
msg 1 sent by (0, 2) at 1, delivered to (2, 1) at 7
```

- Here is another example on a 2×2 network using two (tracked) messages. Note that both messages are delivered at the same time to core $(1, 1)$. However, the routing algorithm guarantees that message 1 is delivered before message 2:

```
% cat /tmp/times2
1 (0,0) (1,1) 1*
2 (1,0) (1,1) 2*
% sbt --warn 'runMain cs735_835.noc.ExecSimulator 2 2 file:/tmp/times2 2 2'
at 1, new msg 1 generated by (0, 0) intended for (1, 1)
at 2, (0, 0) starts msg 1 towards East
at 2, new msg 2 generated by (1, 0) intended for (1, 1)
at 3, (1, 0) starts msg 2 towards East
at 3, (0, 1) moves msg 1 from West to South
at 4, msg 1 is delivered to (1, 1)
at 4, msg 2 is delivered to (1, 1)
simulation completed: 0.00 seconds
msg 1 sent by (0, 0) at 1, delivered to (1, 1) at 4
msg 2 sent by (1, 0) at 2, delivered to (1, 1) at 4
```

- Here is a larger example of 10000 messages going through a 100×100 network:

```
% sbt --warn 'runMain cs735_835.noc.ExecSimulator 100 100
  http://cs.unh.edu/~cs735/Programming/10000-random-messages.in 2 2' | tail -5
simulation completed: 2.65 seconds
msg 9997 sent by (98, 86) at 5036, delivered to (91, 34) at 5178
msg 9998 sent by (71, 36) at 5036, delivered to (33, 61) at 5124
msg 9999 sent by (35, 74) at 5036, delivered to (57, 59) at 5144
msg 10000 sent by (34, 43) at 5037, delivered to (38, 22) at 5121
```

Since simulations are deterministic, the output produced by any simulator should be identical to `http://cs.unh.edu/~cs735/Programming/10000-random-messages.out`.