Barriers

[§8.2] Like locks, barriers can be implemented in different ways, depending upon how important efficiency is.

- Performance criteria
  - Latency: time spent from reaching the barrier to leaving it
  - Traffic: number of bytes communicated as a function of number of processors
- In current systems, barriers are typically implemented in software using locks, flags, counters.
  - Adequate for small systems
  - Not scalable for large systems

A thread might have this general organization:

```
..
parallel region
BARRIER
parallel region
BARRIER
..
```

Note that barriers are usually constructed using locks, and thus can use any of the lock implementations in the previous lecture.

A barrier can be implemented like this (first attempt):

```
// shared variables used in barrier & their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// barrier implementation
void barrier () {
   lock(&barLock);
      if (numArrived == 0)  // first thread sets flag
         canGo = 0;
      numArrived++;
```

```
      int myCount = numArrived;
   unlock(&barLock);

   if (myCount < NUM_THREADS) {
      while (canGo == 0) {}; // wait for last thread
   }
   else { // this is the last thread to arrive
      numArrived = 0; // reset for next barrier
      canGo = 1; // release all threads
 }
}
```

What's wrong with this?

Sense-reversal centralized barrier

[§8.2.1] The simplest solution to the correctness problem above just toggles the barrier …

- the first time, the threads wait for **`canGo`** to become 1;
- the next time they wait for it to become 0;
- and then they alternate waiting for it to become 1 and 0 at successive barriers.

Here is the code:

```
// variables used in a barrier and their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// thread-private variable
int valueToAwait = 0;

// barrier implementation
void barrier () {
   valueToAwait = 1 - valueToAwait; // toggle it
   lock(&barLock);
      numArrived++;
      int myCount = numArrived;
   unlock(&barLock);
```

```
    if (myCount < NUM_THREADS) {
       while (canGo != valueToAwait) {}; // await last thread
    }
    else  { // this is the last thread to arrive
       numArrived = 0; // reset for next barrier
       canGo = valueToAwait; // release all threads
    }
}
```

How does the traffic at this barrier scale?

Combining-tree barrier

[§8.2.2] A tree-based strategy can be used to reduce contention, similarly to the way we used partial sums in Lecture 6.

- Threads represent the leaf nodes of a tree.
- The non-leaf nodes are the variables that the threads spin on.
- Each thread spins on the variable of its immediate parent, which constitutes an intermediate barrier.
- Once all threads have arrived at the intermediate barrier, one of these threads goes on and spins on the variable immediately above.
- This is repeated until the root is reached. At this point, the root releases all threads by setting a flag.

How does this improve performance?

But there is an offsetting cost to a combining tree. What is it?

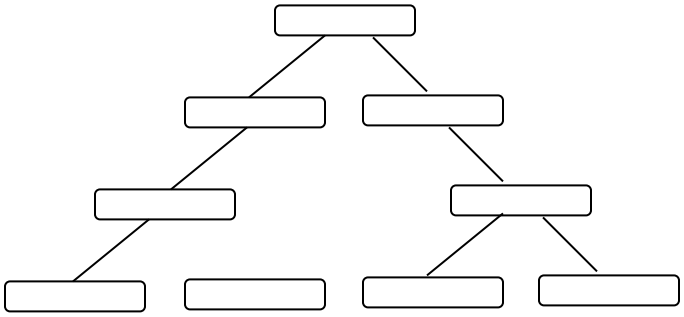
[§8.2.3] In very large supercomputers, however, this technique does not suffice.

The BlueGene/L system has a special *barrier network* for implementing barriers and broadcasting notifications to processors.

The network contains four independent channels.

Each level does a global **and** of the signals from the levels below it.

The signals are combined in hardware and propagate to the top of a combining tree.



The tree can also be used to do a global interrupt when the entire machine or partition must be stopped as soon as possible “for diagnostic purposes.”

In this case, each level does a global **or** of the signals from beneath.

Once the signal propagates to the top of the tree, the resultant notification is broadcast down the tree.

The round-trip latency is only 1.5 μs for a system of 64K nodes.