Threading

Week 4, Lecture 1
So Far …

• Multicore Architecture
  – While tedious to many a computer scientist, *it is critical to have a basic understanding of what is going on underneath you.*
  – Recall discussion on cache coherency and performance: *Not writing code that conforms to the architectural features can result in performance degradation.*

• Today
  – Exposure to programming models: POSIX threads
  – How concurrency is introduced into actual algorithms
  – Observe correctness and performance problems, their cause, and fix.
Our First Algorithm …

• Starting with a simple algorithm to illustrate a few concepts …
  – How one implements algorithms that use multicore.
  – The correctness considerations that arise.
  – Performance features.
  – How concurrency sometimes forces a new approach to be taken to a problem …
Our First Algorithm …

- Starting with a simple algorithm to illustrate a few concepts …
  - How one implements algorithms that use multicore.
  - The correctness considerations that arise.
  - Performance features.
  - How concurrency sometimes forces a new approach to be taken to a problem …

- Algorithm: Cellular Automata (a.k.a. “Game of Life”)
  - Large # of cells, related via a simple neighborhood layout
  - Simple set of rules that is applied to evolve the state of the collection of cells in time.
Algorithm: Game of Life

• Start with an “infinite” 2D array of cells, each of which is in one of two possible states: live or dead.
• Each cell interacts with its 8 neighbors.
• Update Rules for Cells
  1. Any live cell with < 2 live neighbors dies. (“Underpopulation”)
  2. Any live cell with > 3 live neighbors dies. (“Overcrowding”)
  3. Any live cell with 2 or 3 live neighbors lives unchanged to the next generation.
  4. Any cell with exactly 3 live neighbors cells will be populated with a living cell.
• Apply the above rules simultaneously to every cell.
• Repeat cell update every “clock tick” …
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi) {
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    w->data[MODIDX(j, i+1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i, w->n, w->m)] +
                    w->data[MODIDX(j-1, i, w->n, w->m)];
                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    w->data[MODIDX(j, i+1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i, w->n, w->m)] +
                    w->data[MODIDX(j-1, i, w->n, w->m)];

                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```

Iterate "iters" time steps
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    w->data[MODIDX(j, i+1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i, w->n, w->m)] +
                    w->data[MODIDX(j+1, i, w->n, w->m)] +
                    w->data[MODIDX(j-1, i, w->n, w->m)];

                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                w->data[MODIDX(j, i+1, w->n, w->m)] +
                w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                w->data[MODIDX(j, i-1, w->n, w->m)] +
                w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                w->data[MODIDX(j-1, i-1, w->n, w->m)] +
                w->data[MODIDX(j+1, i, w->n, w->m)] +
                w->data[MODIDX(j-1, i, w->n, w->m)];

                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {

                int sum =
                    w->data[MODIDX(j, i+1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                    w->data[MODIDX(j, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j-1, i-1, w->n, w->m)] +
                    w->data[MODIDX(j+1, i, w->n, w->m)] +
                    w->data[MODIDX(j-1, i, w->n, w->m)];

                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```

Copyright © 2009 by W. Feng. Based on material from M. Sottile.
Algorithm: First Hack

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi) {
    int i, j, iter;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                w->data[MODIDX(j, i+1, w->n, w->m)] +
                w->data[MODIDX(j+1, i+1, w->n, w->m)] +
                w->data[MODIDX(j-1, i+1, w->n, w->m)] +
                w->data[MODIDX(j, i-1, w->n, w->m)] +
                w->data[MODIDX(j+1, i-1, w->n, w->m)] +
                w->data[MODIDX(j-1, i-1, w->n, w->m)] +
                w->data[MODIDX(j+1, i, w->n, w->m)] +
                w->data[MODIDX(j-1, i, w->n, w->m)];

                if (w->data[IDX(j, i, w->n)] == 0 && sum == 3)
                    w->data[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        w->data[IDX(j, i, w->n)] = 0;
                    else
                        w->data[IDX(j, i, w->n)] = w->data[IDX(j, i, w->n)];
                }
            }
        }
    }
}
```
• Hmmm … a sequential algorithm with a bug already?

• The algorithm requires that all cells are updated at the same time. So, …
  – Cannot do an in-place update because the order of iteration over the array can cause the $i^{th}$ iteration for internal cells (i.e., those not at the top or left side) to make their result be a function of partly the $i^{th}$ iteration and partly the $(i-1)^{th}$ iteration.

• Solution?
char *update(word_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;
    char *cur, *prev, *tmp;

    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    prev[MODIDX(j, i+1, w->n, w->m)] + /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }
    }

    tmp = cur;
    cur = prev;
    prev = tmp;
}
Algorithm with Double Buffering

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi) {
    int i, j, iter;
    char *cur, *prev, *tmp;
    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum = prev[MODIDX(j, i+1, w->n, w->m)] + /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }
    }

    tmp = cur;
    cur = prev;
    prev = tmp;
}
```

Get some pointers for two buffers and a temp for swapping.
Algorithm with Double Buffering

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;
    char *cur, *prev, *tmp;

    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    prev[MODIDX(j, i+1, w->n, w->m)] + /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }

        tmp = cur;
        cur = prev;
        prev = tmp;
    }
}
```
```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi) {
    int i, j, iter;
    char *cur, *prev, *tmp;

    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    // Compute sum from previous time step.
                    prev[MODIDX(j, i+1, w->n, w->m)] + /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }
        tmp = cur;
        cur = prev;
        prev = tmp;
    }
}
```

Copyright © 2009 by W. Feng. Based on material from M. Sottile.
Algorithm with Double Buffering

```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi)
{
    int i, j, iter;
    char *cur, *prev, *tmp;

    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                    prev[MODIDX(j, i+1, w->n, w->m)] /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }
    }

    tmp = cur;
    cur = prev;
    prev = tmp;
}
```

Apply automaton rules with exclusive read access to previous time step, exclusive write into current.
```c
char *update(world_t *w, int iters, int m_lo, int m_hi, int n_lo, int n_hi) {
    int i, j, iter;
    char *cur, *prev, *tmp;

    cur = w->worldB;
    prev = w->worldA;

    for (iter=0; iter<iters; iter++) {
        for (i=0; i<m_hi; i++) {
            for (j=0; j<n_hi; j++) {
                int sum =
                prev[MODIDX(j, i+1, w->n, w->m)] + /* ... other 7 terms */;
                if (prev[IDX(j, i, w->n)] == 0 && sum == 3)
                    cur[IDX(j, i, w->n)] = 1;
                else {
                    if (sum < 2 || sum > 3)
                        cur[IDX(j, i, w->n)] = 0;
                    else
                        cur[IDX(j, i, w->n)] = prev[IDX(j, i, w->n)];
                }
            }
        }
    }

    tmp = cur;
    cur = prev;
    prev = tmp;
}
```

Swap pointers.
Sequential to Parallel?

• Easiest Approach
  – Split the array across multiple processing elements such that each PE works on a part of the array.
  – For a sufficiently large array, the result will be the algorithm runs faster. (What is the problem with a small array?)
  – Key Detail: Loop Control
    • Let each thread have a different start and end.
      ```c
      for (i=m_lo; i<m_hi; i++)
      ```
Threads: Partitioning the Problem

\[ n = 0 \ldots n_{hi} \]

\[ m = 0 \ldots m_{hi}/4 \]

\[ m = m_{hi}/4 \ldots m_{hi}/2 \]

\[ m = m_{hi}/2 \ldots (3m_{hi})/4 \]

\[ m = (3m_{hi})/4 \ldots m_{hi} \]
Pthreads

• We will use the following primitives, which is a tiny subset of pthreads.
  – pthread_create
  – pthread_join
  – pthread_exit
  – pthread_barrier
Threading with Cellular Automata (CA)

• Idea
  – Encapsulate the “work loop” inside of a thread.

• Approach
  – For each thread, set-up per-thread bounds such that each thread works on a distinct (disjoint) portion of the array.

• Context
  – POSIX thread (a.k.a. pthreads)
  – Note
    • Analogous functions to those in pthreads exist for other threading models, e.g., Windows threads
Fork-Join Model

• Idea
  – The main thread *forks* control to a set of threads that will execute concurrently.
  – These “worker” threads can then *join* back up with the main thread, possibly repeating the fork-join pattern.

- Cellular Automata Example
  – Only fork once …
**pthread_create**: The “Fork” Step

```c
rc = pthread_create(&threads[i], NULL, worker_thread, (void *)i);
```

- First arg: Thread context for the $i^{th}$ thread.
- Second arg: Optional attributes.
- Third arg: A function that will perform the work. The function must have the signature `void *name (void *arg)`
- Fourth arg: The argument to pass to the function that forms the executable thread body. In C, this is cast to a `void*` and cast back to the right type inside the function.
- Return: Error code.
**pthread_create**: The “Fork” Step

- Creates a new thread inside the program that is identified by the thread context argument (unique per thread).
- Executes code represented by the function passed-in on the third argument.
- The call immediately returns, and upon return, the thread has started, and the thread that called `pthread_create` continues on.
Example Usage: `pthread_create`

```c
pthread_t *threads;

threads = malloc(sizeof(pthread_t)*nthreads);

/* spawn off a set of update threads */
for (i=0; i<nthreads; i++) {
    rc = pthread_create(&threads[i], NULL, worker_thread, (void *)i);
    if (rc) {
        printf("ERROR with pthread_create!\n");
        exit(EXIT_FAILURE);
    }
}
```
The Worker Thread

- Use the same update routine as before *but* restricted to a subset of the elements (or cells) of the array.
- Let the worker routine passed to `pthread_create()` determine the bounds given its thread ID (0 … n-1) and the globally known array dimensions.

```c
void *worker_thread(void *args) {
    int thread_id = (int)args;
    int m_lo, m_hi;
    m_lo = (m/nthreads)*thread_id;
    m_hi = (m/nthreads)*(thread_id+1);
    last = update(iters, m_lo, m_hi, 0, w->n);
    pthread_exit(NULL);
}
```

```c
rc = pthread_create(&threads[i], NULL, worker_thread, (void *)i);
```
**pthread_join: The “Join” Step**

- Only want the master (main) thread to proceed after worker threads have finished.

  ```c
  rc = pthread_join(threads[i], (void **)&status);
  ```

- 1st arg: The thread to join
- 2nd arg: The thread status value
- Return: Error code

```c
/* join the threads */
for (i=0;i<nthreads;i++) {
    rc = pthread_join(threads[i], (void **)&status);
    if (rc) {
        printf("ERROR with pthread_join!\n");
        exit(EXIT_FAILURE);
    }
}
```
pthread_join: The “Join” Step

• Block the caller until the thread that is being joined calls pthread_exit().

• The argument to pthread_exit() is the status pointer that is returned through pthread_join().
  – Not using it here, but like the argument to the worker routine, it is a void * that you can put data into.

```c
void *worker_thread(void *args) {
    int thread_id = (int)args;
    int m_lo, m_hi;

    m_lo = (m/nthreads)*thread_id;
    m_hi = (m/nthreads)*(thread_id+1);

    last = update(iters,m_lo,m_hi,0,w->n);

    pthread_exit(NULL);
}
```
Three Primitives = Basic Parallel Program

- `int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);`

- `int pthread_join(pthread_t thread, void **value_ptr);`

- `void pthread_exit(void *value_ptr);`

- These three pthreads calls can be used for a simple fork-join parallel program.
Correctness?

- Validate correctness against a known working reference.
- How is our program?
  - Lousy ... it only *sometimes* works.

```
wuchun-fengs-computer-2:~ feng$ perl ./compare.pl
fail.
fail.               fail.
fail.               fail.
fail.               fail.
fail.               fail.
fail.               success!
fail.               fail.
fail.               fail.
fail.               fail.
fail.               fail.
```
#!/usr/bin/perl
$m = 200;
$n = 1600;
$threads = 2;
$iters = 2;

for ($i = 0; $i < 100000; $i {
    system("./gol_serial $m $n $iters $i > ../ref");
    system("./gol_parallel1 $m $n $iters $i $threads > ../test");
    $wc = 'diff ../test ../ref | wc -l';
    if ($wc == 0) {
        print "success!\n";
    } else {
        print "fail.\n";
    }
}

It Sometimes Works?!

- This sometimes happens in sequential programming, but usually because you have an actual bug that can cause a crash, e.g.,
  - Microsoft PowerPoint 2004 animation bug
  - Wandering off the end of an array in C into memory into another data structure.
- The Game of Life (gol) parallel program runs without crashing
  - Syntactically correct.
  - Semantically correct … in the single-threaded sense!
  - So, what’s wrong?
Hint: Here’s the Difference

Serial  Parallel  Difference
Synchronization … the Lack Thereof …

• When a processor finishes working on its block, it swaps its local pointers and starts working on the block in the next buffer.

• Assumptions
  – All threads work on the same buffers at any given time.
  – The double buffering scheme assumes that one buffer is read only during each iteration.
What We Assume …
What We Assume ...
What We Assume ...
So, What’s the Problem?

• What the code says is to …
  – Swap the buffers immediately after finishing the block an individual thread is responsible for and start again.
  – The code does NOT state that the thread should NOT start updating the next iteration until all other threads have ALSO swapped their buffers.
What Happened ...
What Happened …

 iteration 1  iteration 1  iteration 2  iteration 2
What Happened …

Iteration 2  Iteration 1  Iteration 3  Iteration 2
Uncoordinated Mess

• Double buffering requires that only one buffer be read each iteration.
  – But we are NOT enforcing this in a global sense – only locally to a thread.
  – Why is this bad?
    • We need a globally consistent state across the threads for the overall algorithm to make sense.
Double Buffering &
Now … Synchronization

• The “tug of war” between correctness and performance.

• Correctness
  – Double buffering to ensure that values read to compute cell value for next time were all from the same iteration.
  – Result: Serialized code to enforce ordering of loop iterations.

• Performance
  – Parallelize but a similar phenomenon can occur due to uncoordinated boundary access between threads where a cell being updated by one thread reads data updated by another.
Soapbox Perspective: Correctness

- There exists a set of constraints (e.g., invariants) that are needed to make an algorithm work.
  - Parallelization: Ensure these invariants still hold.
  - Cellular Automata (a.k.a. Game of Life)
    - Constraint: During an iteration, data is read from the previous timestep buffer and written only in the current timestep buffer.
    - Sequential code reflected this … but need to ensure that the same constraint is enforced globally in parallel version.
The Problematic Code Segment

```c
/** ..snip.. **/
else {
    if (sum < 2 || sum > 3) cur[IDX(j,i,w->n)] = 0;
    else cur[IDX(j,i,w->n)] = prev[IDX(j,i,w->n)];
}
}
}
}

tmp = cur;
cur = prev;
prev = tmp;
} /* iteration loop end */
```

Here we have enforced the local constraint, but not the global one! The thread must ensure that the global constraint holds also.
Solution: Barriers

• Similar to synchronizing by joining …
  – Force threads to stop executing until every thread has reached the barrier point. Then allow threads to proceed.
  – Preferred over joining and re-forking for performance reasons.
    1. Thread teardown and startup is inefficient.
    2. Join and fork could be achieved by keeping the thread around and re-using its thread context when forking again to re-start the computation. (Barriers are simply a cleaner way of doing this.)
    3. A barrier is a form of “collective” operation. Why important?
  • Collectives can be performed efficiently in the run-time system or hardware, so using the abstraction is a better way to take advantage of this than to hand-code it.
Barrier: Basic Idea

Time

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
</table>

Barrier

Barrier

Barrier

Copyright © 2009 by W. Feng. Based on material from M. Sottile.
Barrier: Why?

Arrival time skew!
**pthread Barrier: Synchronize!**

- A barrier is a *structure* provided by pthreads.
  - A *counter* surrounded by synchronization mechanisms.
  - What properties are of the *counter*?

- Programmer Beware!
  - Sometimes a barrier is *not* provided by your pthreads version.
  - How would you implement it? (See comment about properties above.)
**pthread_barrier**: Synchronize!

- Declare a globally visible barrier structure:
  ```c
  pthread_barrier_t barrier;
  ```

- Whenever you want the code to stop and wait until all peers have hit the barrier, insert a `pthread_barrier_wait`:
  ```c
  pthread_barrier_wait(&barrier);
  ```

- Before starting, initialize the barrier.
  - Need to how many threads are required to hit the barrier before it unblocks, i.e.,
    ```c
    pthread_barrier_init(&barrier, nthreads);
    ```

- Upon finishing clean-up the barrier.
  ```c
  pthread_barrier_destroy(&barrier);
  ```
The Fix, Part 1

- add a global barrier that is initialized before threads start.

```c

barrier_t barrier;

/* code ... */

int main(int argc, char **argv) {
    /* code ... */

    /* init barrier */
    barrier_init(&barrier, nthreads);

    /* spawn threads with pthread_create() */
}
```
• Add barrier wait call where we need it to occur, i.e., near the buffer pointer swap.
• Because the swap is local, we can put the barrier before or after the actual swap … however … after is advisable. Why?

```c
if (prev[IDX(j,i,w->n)] == 0 && sum == 3) 
    cur[IDX(j,i,w->n)] = 1;
else {
    if (sum < 2 || sum > 3) cur[IDX(j,i,w->n)] = 0;
    else cur[IDX(j,i,w->n)] = prev[IDX(j,i,w->n)];
}
barrier_wait(&barrier);

tmp = cur;
cur = prev;
prev = tmp;
```
The Result

- Buffer swap to implement double buffering locally in order to enforce the **local invariant**, then
- Barrier to perform a global synchronization and ensure that every processor is working on the same timestep in order to enforce the **global invariant**.
- It works against the reference implementation!

wuchun-fengs-computer-2:~ feng$ perl ./compare.pl
success!
success!
success!
success!
success!
success!
Next Time

- **Performance**
  - What is the penalty of barriers for enforcing correctness?
  - How do we measure performance?
  - Strong versus weak scaling.

- **New Algorithm**
  - Implement as part of a multicore programming assignment.

- **Transition to GPGPU Portion of the Course**