Lecture 5: Parallel machines and models; shared memory programming

David Bindel

8 Feb 2010

Logistics

- Try out the wiki! In particular, try it if you don't have a partner.
  `https://confluence.cornell.edu/display/cs5220s10/`
- TA is Nikos Karampatziakis.
  OH: 4156 Upson, M 3-4, Th 3-4.

Recap from last time

Last time: parallel *hardware* and *programming models*

- Programming model doesn't *have* to "match" hardware
- Common HW:
  - Shared memory (uniform or non-uniform)
  - Distributed memory
  - Hybrid
- Models
  - Shared memory (threaded) – pthreads, OpenMP, Cilk, ...
  - Message passing – MPI
- Today: shared memory programming
  - ... after we finish a couple more parallel environments!

Global address space programming

- Collection of named threads
  - Local and shared data, like shared memory
  - Shared data is partitioned – non-uniform cost
  - Cost is programmer visible (know “affinity” of data)
- Like a hybrid of shared memory and distributed memory
- Examples: UPC, Titanium, Co-Array Fortran

Global address space hardware?

- Some network interfaces allow remote DMA (direct memory access)
- Processors can do one-sided put/get ops to other memories
  - Remote CPU doesn't have to actively participate
- Don't cache remote data locally – skip coherency issues
- Example: Cray T3E, clusters with Quadrics, Myrinet, Infiniband

Data parallel programming model

- Single thread of control
- Parallelism in operations acting on arrays
  - Think MATLAB! (the good and the bad)
- Communication implicit in primitves
- Doesn't fit all problems

SIMD and vector systems

- Single Instruction Multiple Data systems
  - One control unit
  - Lots of little processors: CM2, Maspar
  - Long dead
- Vector machines
  - Multiple parallel functional units
  - Compiler responsible for using units efficiently
  - Example: SSE and company
  - Bigger: GPUs
  - Bigger: Cray X1, Earth simulator

Hybrid programming model

Hardware is hybrid — consider clusters! Program to match hardware?

- Vector ops with SSE / GPU
- Shared memory on nodes (OpenMP)
- MPI between nodes
- Issue: conflicting libraries?!
  - Are MPI calls thread-safe?
  - Only a phase?
- Must be a better way...

Memory model

- Single processor: return last write
  - What about DMA and memory-mapped I/O?
- Simplest generalization: *sequential consistency* – as if
  - Each process runs in program order
  - Instructions from different processes are interleaved
  - Interleaved instructions ran on one processor

Sequential consistency

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

– Lamport, 1979

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Without sequential consistency support, what if

1. Processor 2 caches `flag`?
2. Compiler optimizes away loop?
3. Compiler reorders assignments on P1?

Starts to look restrictive!

Sequential consistency: the good, the bad, the ugly

Program behavior is “intuitive”:

- Nobody sees garbage values
- Time always moves forward

One issue is *cache coherence*:

- Coherence: different copies, same value
- Requires (nontrivial) hardware support

Also an issue for optimizing compiler!

There are cheaper *relaxed* consistency models.

Snoopy bus protocol

Basic idea:

- Broadcast operations on memory bus
- Cache controllers "snoop" on all bus transactions
    - Memory writes induce serial order
    - Act to enforce coherence (invalidate, update, etc)

Problems:

- Bus bandwidth limits scaling
- Contending writes are slow

There are other protocol options (e.g. directory-based).
But usually give up on *full* sequential consistency.

Weakening sequential consistency

Try to reduce to the *true* cost of sharing

- `volatile` tells compiler when to worry about sharing
- Memory fences tell when to force consistency
- Synchronization primitives (lock/unlock) include fences

Sharing

True sharing:

- Frequent writes cause a bottleneck.
- Idea: make independent copies (if possible).
- Example problem: malloc/free data structure.

False sharing:

- Distinct variables on same cache block
- Idea: make processor memory contiguous (if possible)
- Example problem: array of ints, one per processor

Take-home message

- Sequentially consistent shared memory is a useful idea...
  - “Natural” analogue to serial case
  - Architects work hard to support it
- ... but implementation is costly!
  - Makes life hard for optimizing compilers
  - Coherence traffic slows things down
  - Helps to limit sharing

Okay. Let’s switch gears and discuss threaded code.

Reminder: Shared memory programming model

Program consists of *threads* of control.

- Can be created dynamically
- Each has private variables (e.g. local)
- Each has shared variables (e.g. heap)
- Communication through shared variables
- Coordinate by synchronizing on variables
- Examples: pthreads, OpenMP, Cilk, Java threads

Mechanisms for thread birth/death

- Statically allocate threads at start
- Fork/join (pthreads)
- Fork detached threads (pthreads)
- Cobegin/coend (OpenMP?)
  - Like fork/join, but lexically scoped
- Futures (?)
  - `v = future(somefun(x))`
  - Attempts to use `v` wait on evaluation

Mechanisms for synchronization

- Locks/mutexes (enforce mutual exclusion)
- Monitors (like locks with lexical scoping)
- Barriers
- Condition variables (notification)

Concrete code: pthreads

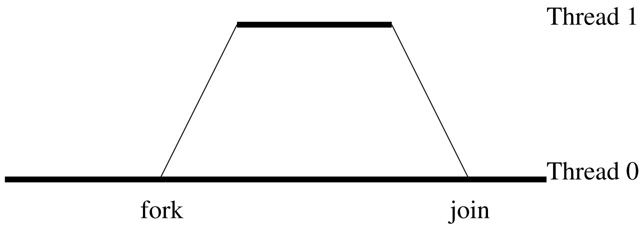
- pthreads = POSIX threads
- Standardized across UNIX family
- Fairly low-level
- Heavy weight?

Wait, what's a thread?

Processes have *state*. Threads share some:

- Instruction pointer (per thread)
- Register file (per thread)
- Call stack (per thread)
- Heap memory (shared)

Thread birth and death



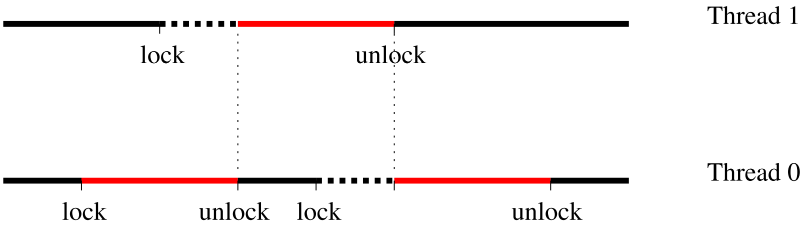
Thread is created by *forking*.
When done, *join* original thread.

Thread birth and death

```
void thread_fun(void* arg);

pthread_t thread_id;
pthread_create(&thread_id, &thread_attr,
               thread_fun, &fun_arg);
...
pthread_join(&thread_id, NULL);
```

Mutex

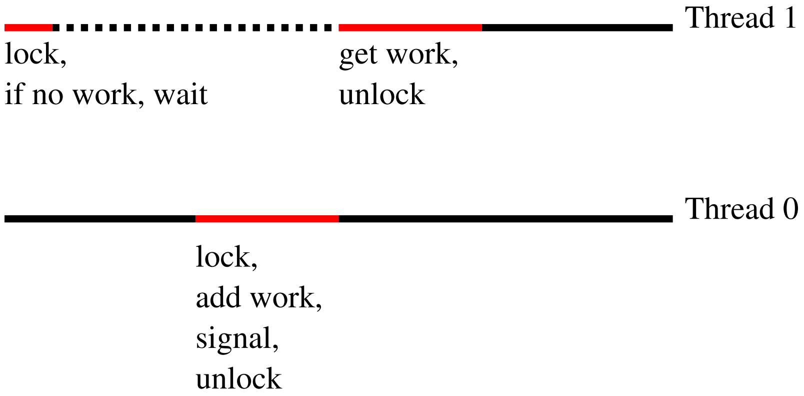


Allow only one process at a time in *critical section* (red).
Synchronize using locks, aka mutexes (*mutual exclusion vars*).

Mutex

```
pthread_mutex_t l;
pthread_mutex_init(&l, NULL);
...
pthread_mutex_lock(&l);
/* Critical section here */
pthread_mutex_unlock(&l);
...
pthread_mutex_destroy(&l);
```

Condition variables



Allow thread to wait until condition holds (e.g. work available).

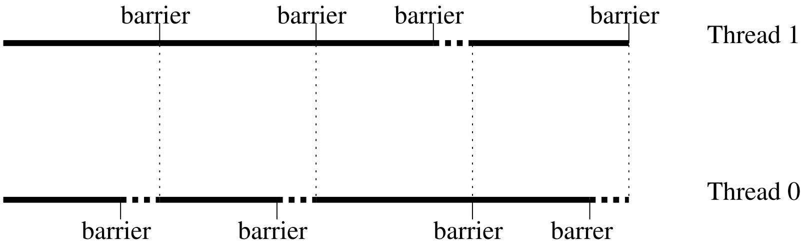
Condition variables

```
        pthread_mutex_t l;
        pthread_cond_t cv;
        pthread_mutex_init(&l)
        pthread_cond_init(&cv, NULL);

/* Thread 0 */            /* Thread 1 */
mutex_lock(&l);           mutex_lock(&l);
add_work();               if (!work_ready)
cond_signal(&cv);             cond_wait(&cv, &l);
mutex_unlock(&l);         get_work();
                          mutex_unlock();

        pthread_cond_destroy(&cv);
        pthread_mutex_destroy(&l);
```

Barriers



Computation phases separated by barriers.
Everyone reaches the barrier, the proceeds.

Barriers

```
pthread_barrier_t b;
pthread_barrier_init(&b, NULL, nthreads);
...
pthread_barrier_wait(&b);
...
```

Synchronization pitfalls

- Incorrect synchronization $\implies$ *deadlock*
  - All threads waiting for what the others have
  - Doesn't always happen! $\implies$ hard to debug
- Too little synchronization $\implies$ data races
  - Again, doesn't always happen!
- Too much synchronization $\implies$ poor performance
  - ... but makes it easier to think through correctness

Deadlock

Thread 0:

```
lock(l1); lock(l2);
Do something
unlock(l2); unlock(l1);
```

Thread 1:

```
lock(l2); lock(l1);
Do something
unlock(l1); unlock(l2);
```

Conditions:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

The problem with pthreads

Portable standard, but...

- Low-level library standard
- Verbose
- Makes it easy to goof on synchronization
- Compiler doesn't help out much

OpenMP is a common alternative (next lecture).

Example: Work queues

- Job composed of different tasks
- Work gang of threads to execute tasks
- Maybe tasks can be added over time?
- Want dynamic load balance

Example: Work queues

Basic data:

- Gang of threads
- Work queue data structure
- Mutex protecting data structure
- Condition to signal work available
- Flag to indicate all done?

Example: Work queues

```
task_t get_task() {
  task_t result;
  pthread_mutex_lock(&task_l);
  if (done_flag) {
    pthread_mutex_unlock(&task_l);
    pthread_exit(NULL);
  }
  if (num_tasks == 0)
    pthread_cond_wait(&task_ready, &task_l);
  ... Remove task from data struct ...
  pthread_mutex_unlock(&task_l);
  return result;
}
```

Example: Work queues

```
void add_task(task_t task) {
  pthread_mutex_lock(&task_l);
  ... Add task to data struct ...
  if (num_tasks++ == 0)
    pthread_cond_signal(&task_ready);
  pthread_mutex_unlock(&task_l);
}
```