Lecture 6: Intro to shared memory programming

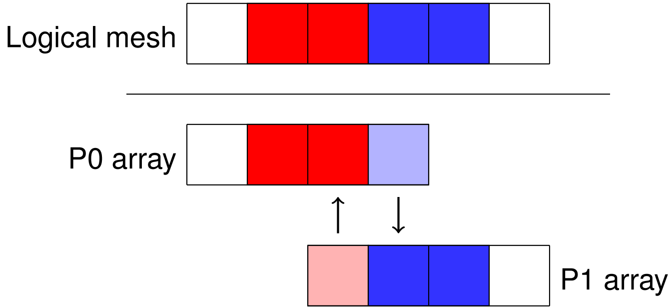
David Bindel

15 Sep 2011

Logistics

- HW 3: Out soon!
- For HW 2:
  - Basic idea should be easy ...
  - ... if you don't get stuck on logistics
  - ... which can easily happen
  - Please don't procrastinate.
- Final project proposal
  - Will post a little CMS entry soon
  - Use Piazza to advertise ideas / find partners
- Final project goals
  - Goal 1: Do some performance analysis/tuning
  - Goal 2: Work on something you care about
  - Goal 3: Get someone else interested! (groups 1–4)
  - Output: Preliminary presentation and final write-up

Reminder: 1D wave



Common message passing pattern

- Logical *global* structure
- *Local* representation per processor
- Local data may have redundancy
  - Example: Data in ghost cells
  - Example: Replicated book-keeping data (`pidx` in our code)

Message passing pain

Common message passing pattern

- Logical *global* structure
- *Local* representation per processor
- Local data may have redundancy
  - Example: Data in ghost cells
  - Example: Replicated book-keeping data (`pidx` in our code)

Big pain point:

- Thinking about many partly-overlapping representations
- Maintaining consistent picture across processes

Wouldn't it be nice to have just one representation?

Shared memory vs message passing

- Implicit communication via memory vs explicit messages
- Still need separate global vs local picture?
  - **No:** One thread-safe data structure may be easier
  - **Yes:** More sharing can hurt performance
    - Synchronization costs cycles even with no contention
    - Contention for locks reduces parallelism
    - Cache coherency can slow even non-contending access
- “Easy” approach: add multi-threading to serial code
- Better performance: design like a message-passing 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

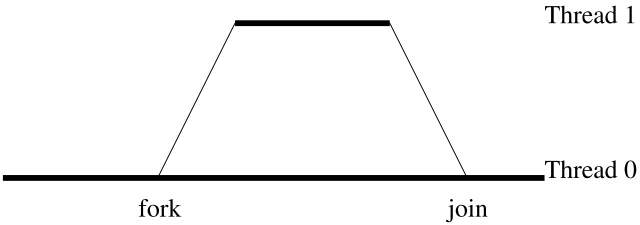
- pthreads = POSIX threads
- Standardized across UNIX family
- Fairly low-level
- Heavy weight?

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



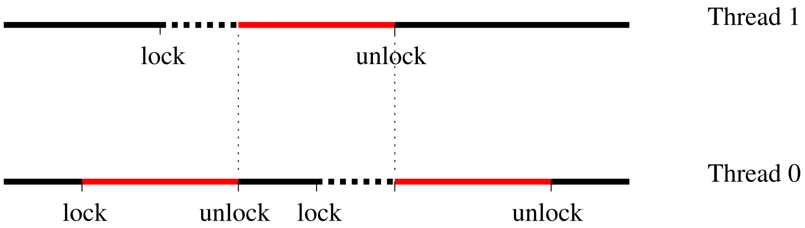
Thread is created by *forking*.
When done, *join* original thread.

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

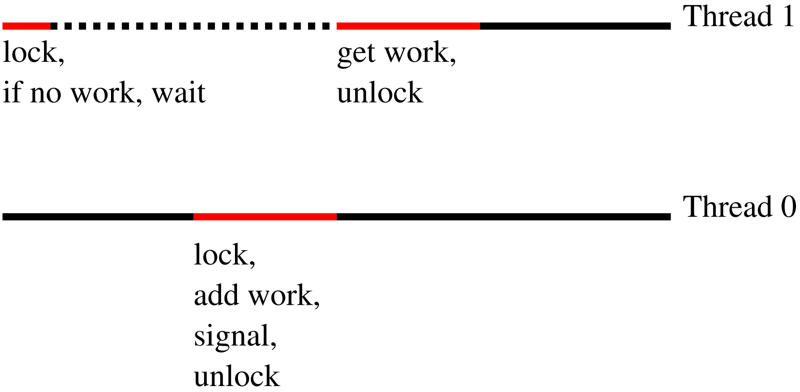


Allow only one process at a time in *critical section* (red).
Synchronize using locks, aka mutexes (*mutual exclusion vars*).

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

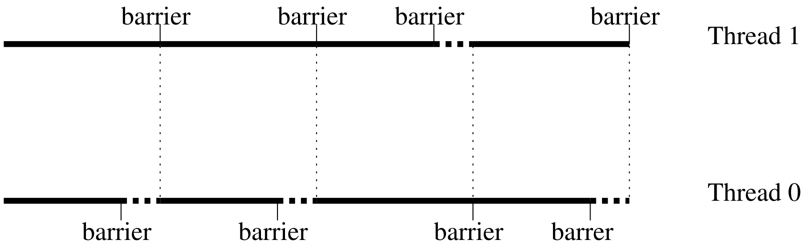
```
pthread_mutex_t l;
pthread_cond_t cv;
pthread_mutex_init(&l)
pthread_cond_init(&cv, NULL);
```

```
/* Thread 0 */
mutex_lock(&l);
add_work();
cond_signal(&cv);
mutex_unlock(&l);
```

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

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

Barriers



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

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);
}
```

Monte Carlo

Basic idea: Express answer a as

$$a = E[f(X)]$$

for some random variable(s) X.

Typical toy example:

$$\pi/4 = E[\chi_{[0,1]}(X^2 + Y^2)] \text{ where } X, Y \sim U(-1, 1).$$

We'll be slightly more interesting...

A toy problem

Given ten points (X_i, Y_i) drawn uniformly in $[0, 1]^2$, what is the expected minimum distance between any pair?

Toy problem: Version 1

Serial version:

```
sum_fX = 0;
for i = 1:ntrials
  x = rand(10,2);
  fX = min distance between points in x;
  sum_fX = sum_fX + fx;
end
result = sum_fX/ntrials;
```

Parallel version: run twice and average results?!
No communication — *embarrassingly parallel*

Need to worry a bit about `rand`...

Error estimators

Central limit theorem: if R is computed result, then

$$R \sim N\left(E[f(X)], \frac{\sigma_{f(X)}}{\sqrt{n}}\right).$$

So:

- Compute sample standard deviation $\sigma_{\hat{f(X)}}$
- Error bars are $\pm\sigma_{\hat{f(X)}}/\sqrt{n}$
- Use error bars to monitor convergence

Toy problem: Version 2

Serial version:

```
sum_fX = 0;
sum_fX2 = 0;
for i = 1:ntrials
  x  = rand(10,2);
  fX = min distance between points in x;
  sum_fX  = sum_fX + fX;
  sum_fX2 = sum_fX + fX*fX;
  result = sum_fX/i;
  errbar = sqrt(sum_fX2-sum_fX*sum_fX/i)/i;
  if (abs(errbar/result) < reltol), break; end
end
result = sum_fX/ntrials;
```

Parallel version: ?

Pondering parallelism

Two major points:

- How should we handle random number generation?
- How should we manage termination criteria?

Some additional points (briefly):

- How quickly can we compute `fX`?
- Can we accelerate convergence (variance reduction)?