Shared Memory Programming

Arvind Krishnamurthy
Fall 2004

Parallel Programming Overview
Basic parallel programming problems:
1. Creating parallelism & managing parallelism
   - Scheduling to guarantee parallelism and load-balance
2. Communication between processors
   - Building shared data structures
3. Synchronization
   - Point-to-point or “pairwise”
   - Global synchronization (barriers)

- Make use of a running example, “Sharks and Fish”

A Model Problem: Sharks and Fish
- Illustration of parallel programming
  - Original version (discrete event only) proposed by Geoffrey Fox
  - Called WATOR
    - Sharks and fish living in a 2D toroidal ocean
  - We can imagine several variations to show different physical phenomenon
- Basic idea: sharks and fish living in an ocean
  - rules for movement
  - breeding, eating, and death
  - forces in the ocean
  - forces between sea creatures

Sharks and Fish as Discrete Event System
- Ocean modeled as a 2D toroidal grid
- Each cell occupied by at most one sea creature

Fish-only: the Game of Life
- A new fish is born if
  - a cell is empty
  - exactly 3 (of 8) neighbors contain fish
- A fish dies (of overcrowding) if
  - cell contains a fish
  - 4 or more neighboring cells are full
- A fish dies (of loneliness) if
  - cell contains a fish
  - less than 2 neighboring cells are full
- Other configurations are stable

- The original Wator problem adds fish-eating sharks

Parallelism in Sharks and Fish
- The activities in this system are discrete events
- The simulation is synchronous
  - use two copies of the grid (old and new)
  - the value of each new grid cell in new depends only on the 9 cells (itself plus neighbors) in old grid
  - Each grid cell update is independent: reordering or parallelism OK
  - Simulation proceeds in timesteps, where (logically) each cell is evaluated at every timestep

- The original Wator problem adds fish-eating sharks

old ocean

new ocean
Parallelism in Sharks and Fish

- **Parallelism** is straightforward
  - Ocean is regular data structure
  - Even decomposition across processors gives load balance
- **Locality** is achieved by using large patches of the ocean
  - Boundary values from neighboring patches are needed for communication (either explicit or implicit as in cache transfers)

  
  Advanced optimization: visit only occupied cells (and neighbors) → load balance is more difficult

Language Notions of Thread Creation

- **cobegin/coend**
  - Statements in block may run in parallel
  - Cobegins are nested
  - Scoped, so you cannot have a missing coend

- **fork/join**
  - `tid = fork(job1, a1);`
  - `job2(a2);`
  - `join tid;`

  Forked function runs in parallel with current thread
  - Join waits for completion (may be in different function)

  - cobegin cleaner, but fork is more general

Programming with Threads

Several Thread Libraries

- PTHREADS is the Posix Standard
  - Solaris threads are very similar
  - Relatively low level
  - Portable but sometimes slow
- P4 (Parmacs) is a widely used portable package
- OpenMP is newer standard
  - Support for scientific programming on shared memory [http://www.openmp.org](http://www.openmp.org)
- User-level vs. kernel level threads
  - User-level threads cannot make use of multi-processors!
  - Kernel-level threads have more overhead
  - Kernel-level threads better integrated with OS actions (page-faults etc.)

Posix Thread Example

```c
#include <pthread.h>
void print_fun( void *message ) {
  printf("%s \n", message);
}
main() {
  pthread_t thread1, thread2;
  char *message1 = "Hello";
  char *message2 = "World";
  pthread_create( &thread1, NULL, (void*)&print_fun, (void*) message1);
  pthread_create(&thread2, NULL, (void*)&print_fun, (void*) message2);
  return(0);
}
```

Compile using gcc –lpthread

Note: There is a race condition in the print statements

Forking Posix Threads

**Signature:**

```c
int pthread_create(pthread_t *thread_id, const pthread_attr_t *thread_attribute,
                  void *(*thread_fun)(void *), void *fun_arg);
```

**Example call:**

```c
ercode = pthread_create(&thread_id; &thread_attribute &thread_fun; &fun_arg);
```

- **thread_id** is the thread id or handle (used to halt, etc.)
- **thread_attribute** various attributes
  - Standard default values obtained by passing a NULL pointer
- **thread_fun** the function to be run (takes and returns void*)
- **fun_arg** an argument can be passed to thread_fun when it starts
- **errorcode** will be set nonzero if the create operation fails

Loop Level Parallelism

Many scientific application have parallelism in loops

- With threads:
  ```c
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      pthread_create (update_cell, ..., ocean);
  ```

  Also needs i & j

- What's wrong with this approach?
SPMD Parallelism with Threads

Creating a fixed number of threads is common:

```c
pthread_t threads[NTHREADS]; /* thread info */
int errCode; /* error code */
int *status; /* return code */
for (int worker=0; worker<NTHREADS; worker++) {
    ids[worker]=worker;
    errcode=pthread_create(&threads[worker],
        NULL, work,
        &ids[worker]));
    if (errcode) { . . . }
}
for (worker=0; worker<NTHREADS; worker++) {
    errcode=pthread_join(threads[worker],
        (void *) &status));
    if (errcode ! ! *status != worker) { . . . }
```

Loop Level Parallelism

- Many scientific applications have parallelism in loops
  - degree may be fixed by data, either
    - start p threads and partition data (SPMD style)
    - start a thread per loop iteration
  - Parallel degree may be fixed, but not work
    - self-scheduling: have each processor grab the next fixed-sized chunk of work
      - want this to be larger than 1 array element
    - guided self-scheduling: decrease chunk size as a remaining work decreases [Polychronopoulos]
  - How to do this:
    - With threads, create a data structure to keep track of chunks

Dynamic Parallelism

- Divide-and-Conquer problems are task-parallel
  - classic example is search (recursive function)
  - arises in numerical algorithms, dense as well as sparse
  - natural style is to create a thread at each divide point
    - too much parallelism at the bottom
    - thread creation time too high
  - Stop splitting at some point to limit overhead
  - Use a "task queue" to schedule
    - have a pool of worker threads
    - place root in a bag (unordered queue)
    - at each divide point, put children
      - isn't this the same as forking them?

Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
  - For Sharks and Fish, natural to share 2 oceans
  - Also need indices i and j, or range of indices to update
  - Often done by creating a large "thread data" struct
    - Passed into all threads as argument

Synchronization in Sharks and Fish

- We use 2 copies of the ocean mesh to avoid synchronization of each element
  - Need to coordinate
    - Every processor must be done updating one grid before using it
    - Also useful to swap old/new to avoid overhead of allocation
    - Need to make sure done with old before making into new
  - Global synchronization of this kind is very common
    - Timesteps, iterations in solvers, etc.

Basic Types of Synchronization: Barrier

Barrier -- global synchronization
- fork multiple copies of the same function "work"
  - SPMD "Single Program Multiple Data"
- Simple use of barriers -- threads hit the same one
  - work_on_my_subgrid();
  - barrier;
  - read_neighboring_values();
  - barrier;
- more complicated -- barriers on branches (or loops)
  - if (tid % 2 == 0) {
    - work1();
    - barrier
  } else { barrier }
- barriers are not provided in many thread libraries
Pairwise Synchronization

- Sharks and Fish example needs only barriers
- Imagine other variations in which pairs of processors would synchronize:
  - World divided into independent "ponds" with creatures rarely moving between them
  - Producer-consumer model of parallelism
  - All processors updating some global information, such as total population count asynchronously
  - Mutual exclusion needed

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks
- threads are working mostly independently
- need to access common data structure
  ```
  lock *l = alloc_and_init(); /* shared */
  acquire(l);
  access data
  release(l);
  ```
- Java and other languages have lexically scoped synchronization
  - similar to cobegin/coend vs. fork and join
- Semaphores are locks plus shared counters and can be used for mutual exclusion
- Locks only affect processors using them:
  - pair-wise synchronization

Pthreads Locks

- Steps: declare a lock and initialize it; make sure it is locked before doing something critical
- Standard locks semantics: only one can thread can have it
  ```
  pthread_mutex_t pond_lock[i];    // declaration
  pthread_mutex_init(&pond_lock[i], NULL);    // initialization
  ```

Locking Issues

- Repeated locking of the same lock
  - Linux has "fast" vs. "recursive" locks
- Reader/writer locks: allow multiple readers to own a lock at any time, but not allow any readers if there is a writer
  ```
  pthread_mutex_trylock is non-blocking ("non-committal")
  ```
- Deadlock issues:
  - Example: T1 locks pond1 followed by pond2, T2 locks pond2 followed by pond1
  - Deadlocks can be analyzed with the "waits-for" graph
  - T1 is waiting for T2 (to release pond2), and T2 is waiting for T1 (to release pond1), and a cycle in this graph implies deadlock
  - Deadlock avoidance: order locks, and each thread obtains the locks it needs in increasing order of relevance ➔ no cycles!

Condition Variables

- Allows for threads to wait for a condition to be satisfied
- Used along with a mutex lock
  ```
  pthread_mutex_lock(&mut);
  while (tasks_left == 0) {
    pthread_cond_wait(&cv, &mut);
  }
  pthread_mutex_unlock(&mut);
  ```

Condition Variable Issues

- Programming discipline: always obtain the lock before signaling or waiting on a condition variable
  - Makes sure that no signals are lost
  ```
  pthread_cond_wait implicitly relinquishes the lock and obtains it back when woken up
  ```
  ```
  pthread_cond_signal wakes up exactly one waiting thread, pthread_cond_broadcast wakes up all waiting threads
  ```
  ```
  - signal could be used without locking
  - A thread could wake up and reset the program level criteria; broadcast does not imply all threads are runnable