Parallel Programming Overview

Finding parallelism and locality in a problem:
- "Sharks and Fish" particle example today
- More on sources of parallelism and locality next week

Basic parallel programming problems:
1. Creating parallelism
   - Loop Scheduling
2. Communication between processors
   - Building shared data structures
3. Synchronization
   - Point-to-point or "pairwise"
   - Global synchronization (barriers)

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 behavior
- 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

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 only depends on the 9 cells (itself and 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

Fish-only: the Game of Life

- An new fish is born if
  - A cell is empty
  - Exactly 3 (of 8) neighbors contain fish
- A fish dies (of overcrowding) if
  - A cell contains a fish
  - 4 or more neighboring cells are full
- A fish dies (of loneliness) if
  - A cell contains a fish
  - Less than 2 neighboring cells are full
- Other configurations are stable
- The original Wator problem adds sharks that eat fish
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
  • although, there isn’t much reuse...
• Advanced optimization: visit only occupied cells (and neighbors) → load balance is more difficult

Particle Systems
• A particle system has
  • a finite number of particles.
  • moving in space according to Newton’s Laws (i.e. \( F = ma \)).
  • time is continuous.
• Examples:
  • stars in space with laws of gravity.
  • electron beam and ion beam semiconductor manufacturing.
  • atoms in a molecule with electrostatic forces.
  • neutrons in a fission reactor.
  • cars on a freeway with Newton’s laws plus model of driver and engine.
• Many simulations combine particle simulation techniques with some discrete event techniques (e.g., Sharks and Fish).

Forces in Particle Systems
• Force on each particle decomposed into near and far:
  \[ \text{force} = \text{external force} + \text{nearby force} + \text{far-field force} \]
• External force
  • ocean current to sharks and fish world (S&F 1).
  • externally imposed electric field in electron beam.
• Nearby force
  • sharks attracted to eat nearby fish (S&F 5).
  • balls on a billiard table bounce off of each other.
  • Van der Waals forces in fluid (1/r^6).
• Far-field force
  • fish attract other fish by gravity-like (1/r^2) force (S&F 2).
  • gravity, electrostatics
  • forces governed by elliptic PDE.

Parallelism in External Forces
• External forces are the simplest to implement.
  • The force on each particle is independent of other particles.
  • Called "embarrassingly parallel".
• Evenly distribute particles on processors
  • Any even distribution works.
  • Locality is not an issue, no communication.
• For each particle on processor, apply the external force.

Parallelism in Nearby Forces
• Nearby forces require interaction and therefore communication.
• Force may depend on other nearby particles:
  • Example: collisions.
  • simplest algorithm is \( O(n^2) \); look at all pairs to see if they collide.
• Usual parallel model is decomposition* of physical domain:
  • \( O(n^p) \) particles per processor if evenly distributed.

Parallelism in Nearby Forces
• Challenge 1: interactions of particles near processor boundary:
  • need to communicate particles near boundary to neighboring processors.
  • surface to volume effect means low communication.
  • Which communicates less: squares (as below) or slabs?

*often called "domain decomposition," but the term also refers to a numerical technique.

Communicate particles in boundary region to neighbors
Parallelism in Nearby Forces

- Challenge 2: load imbalance, if particles cluster:
  - galaxies, electrons hitting a device wall.
- To reduce load imbalance, divide space unevenly.
  - Each region contains roughly equal number of particles.
  - Quad-tree in 2D, oct-tree in 3D.

Example: each square contains at most 3 particles

See: http://njord.umiacs.umd.edu/~brabec/quadtree/points/prquad.html

Parallelism in Far-Field Forces

- Far-field forces involve all-to-all interaction and therefore communication.
- Force depends on all other particles:
  - Examples: gravity, protein folding
  - Simplest algorithm is $O(n^2)$ as in S&F 2, 4, 5.
  - Just decomposing space does not help since every particle needs to "visit" every other particle.

Implement by rotating particle sets.
  - Keeps processors busy
  - All processor eventually see all particles

Use more clever algorithms to beat $O(n^2)$.

Far-field Forces: Particle-Mesh Methods

- Based on approximation:
  - Superimpose a regular mesh.
  - "Move" particles to nearest grid point.
- Exploit fact that the far-field force satisfies a PDE that is easy to solve on a regular mesh:
  - FFT, multigrid (described in future lecture)
- Accuracy depends on the fineness of the grid and the uniformity of the particle distribution.

1) Particles are moved to mesh (scatter)
2) Solve mesh problem
3) Forces are interpolated at particles (gather)

Far-field forces: Tree Decomposition

- Based on approximation.
  - Forces from group of far-away particles "simplified" -- resembles a single large particle.
  - Use tree; each node contains an approximation of descendants.
  - $O(n \log n)$ or $O(n)$ instead of $O(n^2)$.
  - Several Algorithms
    - Barnes-Hut.
    - Fast multipole method (FMM) of Greengard/Rokhlin.
    - Anderson’s method.
  - Discussed in later lecture.

Summary of Particle Methods

- Model contains discrete entities, namely, particles
- Time is continuous -- is discretized to solve

- Simulation follows particles through timesteps
  - All-pairs algorithm is simple, but inefficient, $O(n^2)$
  - Particle-mesh methods approximates by moving particles
  - Tree-based algorithms approximate by treating set of particles as a group, when far away

- May think of this as a special case of a “lumped” system

Creating Parallelism with Threads
Programming with Threads

Several Thread Libraries
• PTHREADS is the Posix Standard
  • Solaris threads are very similar
  • Relatively low level
  • Portable but possibly slow
• P4 (Parmacs) is a widely used portable package
  • Higher level than Pthreads
  • http://www.netlib.org/p4/index.html
• OpenMP is newer standard
  • Support for scientific programming on shared memory
  • http://www.openMP.org

Language Notions of Thread Creation

• cobegin/coend
  cobegin
  job1(a1);
  job2(a2);
coend
  • Statements in block may run in parallel
  • cobegins may be nested
  • Scoped, so you cannot have a missing coend

• fork/join
  tid1 = fork(job1, a1);
  job2(a2);
  join tid1;
  • Forked function runs in parallel with current
  • join waits for completion (may be in different function)

Forking Posix Threads

Signature:
int pthread_create(pthread_t *,
const pthread_attr_t *,
void *(void *)*,
void *);

Example call:
errcode = 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

Posix Thread Example

#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,
                  &print_fun,
                  message1);
  pthread_create(&thread2,
                  NULL,
                  &print_fun,
                  message2);
  return(0);
}

Compile using gcc -lpthread
See Millennium/Seaborg docs for paths/modules

Note: There is a race condition in the print statements

SPMD Parallelism with Threads

Creating a fixed number of threads is common:

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 != worker) {
    . . .
  }
}

Creating Parallelism in OpenMP

• General form of an OpenMP command
  #pragma omp directive-name [clause ... ] newline
  • For example:
    #pragma omp parallel
      { statement1
        statement2 }
  • The statements will be executed by all processors
    • The master (0), is the thread that executed the pragma
    • Others are numbers 1 to p-1
    • The number of threads is set at compile time:
      • setenv OMP_NUM_THREADS 4
OpenMP Example
#include <omp.h>
main () {
  int nthreads, tid;
  /* Fork threads with own copies of variables */
  #pragma omp parallel private(nthreads, tid)
  { /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } /* All created threads terminate */
}

General Parallelism in OpenMP
• May spawn independent operations in OpenMP that uses different code
  #pragma omp section
  structured_block
  #pragma omp section
  structured_block
• These may contain two different blocks of code

Loop Level Parallelism
• Many scientific applications have parallelism in loops
  • With threads:
    - ocean[n][n];
    - for (int i = 0; i < n; i++)
      - for (int j = 0; j < n; j++)
      - pthread_create (update_cell, ..., ocean);
    - in OpenMP:
      #pragma omp for
      for (i=0; i < n; i++)
      { update_cell( ocean... );
      }
  • But overhead of thread creation is nontrivial

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
    - place root in a bag (unordered queue)
    - at each divide point, put children
    - why isn’t this the same as forking them?
  • Imagine sharks and fish that spawn colonies, each simulated as a unit

Communication:
Creating Shared Data Structures
Shared Data and Threads

- Variables declared outside of main are shared
- Object allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- 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

Shared Data and OpenMP

- May designate variables as shared or private
- creature old_ocean [n][n];
- creature new_ocean [n][n];

```c
#pragma omp parallel for
default(shared) private(i, j)
schedule(static, chunk)
for (…) …
```

All variables shared, except i, and j.

Synchronization

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 used
  - 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 -- a 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
- Implicit in OpenMP blocks (unless nowait is specified)

Pairwise Synchronization

- Sharks and Fish example needs only barriers

- Imagine other variations in which pairs of processors would synchronization:
  - World divided into independent “ponds” with creatures rarely moving between them in 1 direction
    - 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
  ```c
  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 give guarantees on "fairness" in getting the lock, but the same idea of mutual exclusion
- Locks only affect processors using them:
  - pair-wise synchronization

---

**Mutual Exclusion in OpenMP**

- Can ensure only 1 processor runs code:
  - `#pragma omp single`
- Or that the master (thread 0) only runs the code:
  - `#pragma omp master`
- Or that all execute it, one at a time
  - `#pragma omp critical`

---

**Summary**

- Problem defines available parallelism and locality
  - External forces are trivial to parallelize
  - Far-field are hardest (require a lot of communication)
  - Near-field are in between
- Shared memory parallelism does not require explicit communication
  - Reads and writes to shared data structures
- Threads are common OS-level library support
  - Need to package shared data an pass it to each thread
- OpenMP provides higher level parallelism constructs
  - Loop level and parallel blocks
- Problem-dependent (not SPMD) expression of parallelism: runtime systems or OS maps to processors
- Mutual exclusion synchronization provided