Shared Memory Programming:

Threads and OpenMP

Lecture 6

James Demmel
www.cs.berkeley.edu/~demmel/cs267_Spr14/

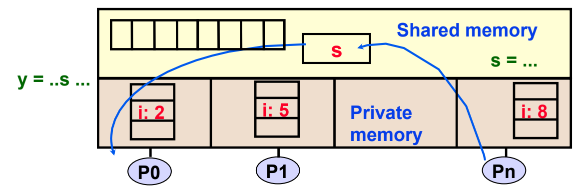
Outline

- Parallel Programming with Threads
- Parallel Programming with OpenMP
  - See parlab.eecs.berkeley.edu/2012bootcampagenda
    - 2 OpenMP lectures (slides and video) by Tim Mattson
  - openmp.org/wp/resources/
  - computing.llnl.gov/tutorials/openMP/
  - portal.xsede.org/online-training
  - www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf
  - Slides on OpenMP derived from: U.Wisconsin tutorial, which in turn were from LLNL, NERSC, U. Minn, and OpenMP.org
    - See tutorial by Tim Mattson and Larry Meadows presented at SC08, at OpenMP.org; includes programming exercises
- (There are other Shared Memory Models: CILK, TBB…)
- Performance comparison
- Summary

Parallel Programming with Threads

Recall Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



Shared Memory Programming

Several Thread Libraries/systems
- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - openmp.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C "ilk"
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Common 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 procedure runs in parallel
  - Wait at join point if it's not finished
- future

```
v = future(job1(a1));
… = …v…;
```

  - Future expression evaluated in parallel
  - Attempt to use return value will wait
- Cobegin cleaner than fork, but fork is more general
- Futures require some compiler (and likely hardware) support

Overview of POSIX Threads

- POSIX: ***P**ortable **O**perating **S**ystem **I**nterface*
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

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
  - Sample attributes: minimum stack size, priority
- 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

Simple Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Loop Level Parallelism

- Many scientific application have parallelism in loops
  - With threads:

```
… my_stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (update_cell[i][j], …,
                      my_stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_cell should have a significant amount of work
  - 1/p-th if possible

Some More Pthread Functions

- `pthread_yield();`
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- `pthread_exit(void *value);`
  - Exit thread and pass value to joining thread (if exists)
- `pthread_join(pthread_t *thread, void **result);`
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- `pthread_t me; me = pthread_self();`
  - Allows a pthread to obtain its own identifier pthread_t thread;
- `pthread_detach(thread);`
  - Informs the library that the thread's exit status will not be needed by subsequent pthread_join calls resulting in better thread performance. For more information consult the library or the man pages, e.g., man -k pthread

Recall Data Race Example

static int s = 0;

Thread 1

for i = 0, n/2-1
s = s + f(A[i])

Thread 2

for i = n/2, n-1
s = s + f(A[i])

- Problem is a race condition on variable s in the program
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Basic Types of Synchronization: Barrier

Barrier -- global synchronization

- Especially common when running multiple copies of the same function in parallel
  - SPMD "Single Program Multiple Data"
- simple use of barriers -- all 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 all thread libraries

Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an attribute object for finer control; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```


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);
```

- Locks only affect processors using them:
  - If a thread accesses the data without doing the acquire/release, locks by others will not help
- Java and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks
  - Can't forgot to say "release"
- Semaphores generalize locks to allow k threads simultaneous access; good for limited resources

Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
// or pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to problems:

```
thread1         thread2
lock(a)         lock(b)
lock(b)         lock(a)
```

- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative

Parallel Programming in OpenMP

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - "Standard" API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.
  - See parlab.eecs.berkeley.edu/2012bootcampagenda
    - 2 OpenMP lectures (slides and video) by Tim Mattson
  - computing.llnl.gov/tutorials/openMP/
  - portal.xsede.org/online-training
  - www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf
- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C, C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

Motivation – OpenMP

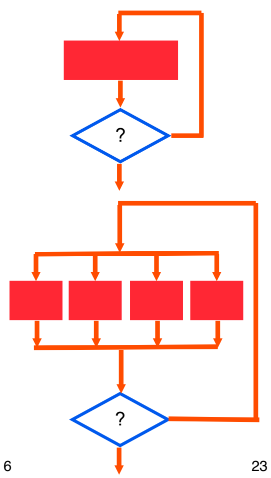
```
int main() {

  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```


Motivation – OpenMP

```
int main() {
  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```


Programming Model – Concurrent Loops

- OpenMP easily parallelizes loops
  - Requires: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds for each thread directly from *serial* source




```
#pragma omp parallel for
for( i=0; i < 25; i++ )
{
  printf("Foo");
}
```


Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team; no one best way
  - **`static([chunk])`** divides iterations statically between threads (default if no hint)
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - `shared` variables are shared
  - `private` variables are private

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
  // private, stack
  int tid;
  #pragma omp parallel \
  shared ( bigdata ) \
  private ( tid )
  /* Calculation goes
  here */
  {
    /* Calc. here */
  }
}
```


Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks / mutexes
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require `flush` directive
  - Single-thread regions *within* parallel regions
    - `master, single` directives

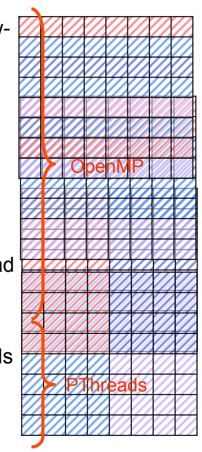
```
#pragma omp critical
{
  /* Critical code here */
}

#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );

#pragma omp single
{
  /* Only executed once */
}
```


Microbenchmark: Grid Relaxation (Stencil)

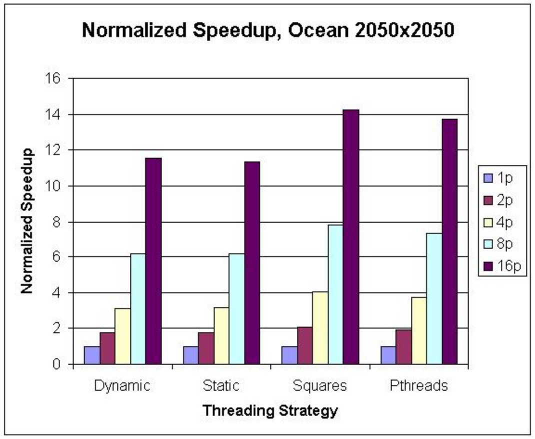
```
for( t=0; t < t_steps; t++) {
  #pragma omp parallel for \
   shared(grid,x_dim,y_dim) private(x,y)

  for( x=0; x < x_dim; x++) {
    for( y=0; y < y_dim; y++) {
      grid[x][y] = /* avg of neighbors */
    }
  }
   // Implicit Barrier Synchronization
  temp_grid = grid;
  grid = other_grid;
  other_grid = temp_grid;
}
```


Microbenchmark: Structured Grid

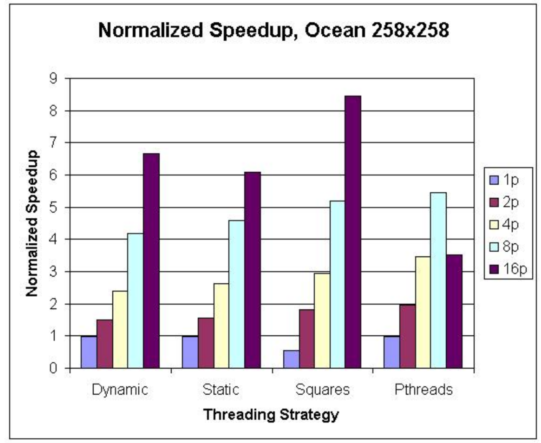
- `ocean_dynamic` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.
- `ocean_static` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.
- `ocean_squares` – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.
- `ocean_pthreads` – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.



Microbenchmark: Ocean



Microbenchmark: Ocean



Evaluation

- OpenMP scales to 16-processor systems
  - Was overhead too high?
    - In some cases, yes (when too little work per processor)
  - Did compiler-generated code compare to hand-written code?
    - Yes!
  - How did the loop scheduling options affect performance?
    - `dynamic` or `guided` scheduling helps loops with variable iteration runtimes
    - `static` or predicated scheduling more appropriate for shorter loops
- OpenMP is a good tool to parallelize (at least some!) applications

OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all (more irregular) applications
- Lots of detailed tutorials/manuals on-line