CS4961 Parallel Programming

Lecture 5: Introduction to Threads (Pthreads and OpenMP)

Mary Hall
September 6, 2011

Homework 2: Due Before Class, Thursday, Sept. 8

'handin cs4961 hw2 <file>'

Problem 1: (Coherence) #2.15 in textbook

(a) Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable x in its cache, and it executes the assignment x=5. Finally, suppose that core 1 doesn't have x in its cache, and after core 0's update to x, core 1 tries to execute y=x. What value will be assigned to y? Why?

(b) Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to y? Why?

(c) Can you suggest how any problems you found in the first two parts might be solved?

Homework 2, cont.

Problem 2: (Bisection width/bandwidth)

(a) What is the bisection width and bisection bandwidth of a 3-d toroidal mesh.

(b) A planar mesh is just like a toroidal mesh, except that it doesn't have the wraparound links. What is the bisection width and bisection bandwidth of a square planar mesh.

Problem 3 (in general, not specific to any algorithm): How is algorithm selection impacted by the value of λ?

Homework 2, cont.

Problem 4: (λ concept) #2.10 in textbook

Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $10^{12}/p$ instructions, and each processor must send 10^9 (p-1) messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all its messages, and there won't be any delays due to things such as waiting for messages.

(a) Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?

(b) Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors?

Reading for Today

- Chapter 2.4-2.4.3 (pgs. 47-52)

  2.4 Parallel Software
  - Caveats
  - Coordinating the processes/threads
  - Shared-memory
- Chapter 4.1-4.2 (pgs. 151-159)

  4.0 Shared Memory Programming with Pthreads
  - Processes, Threads, and Pthreads
  - Hello, World in Pthreads
- Chapter 5.1 (pgs. 209-215)

  5.0 Shared Memory Programming with OpenMP
  - Getting Started

Today's Lecture

- Review Shared Memory and Distributed Memory Programming Models
- Brief Overview of POSIX Threads (Pthreads)
- Data Parallelism in OpenMP
  - Expressing Parallel Loops
  - Parallel Regions (SPMD)
  - Scheduling Loops
  - Synchronization
- Sources of material:
  - Textbook
  - Jim Demmel and Kathy Yelick, UCB
  - openmp.org

Shared Memory vs. Distributed Memory Programs

- Shared Memory Programming
  - Start a single process and fork threads.
  - Threads carry out work.
  - Threads communicate through shared memory.
  - Threads coordinate through synchronization (also through shared memory).
- Distributed Memory Programming
  - Start multiple processes on multiple systems.
  - Processes carry out work.
  - Processes communicate through message-passing.
  - Processes coordinate either through message-passing or synchronization (generates messages).

Review: Predominant Parallel Control Mechanisms

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`

Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code:
		`if ($myid == 0) {`
		`}`

Shared Memory

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.
- Static threads
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.

Thread Safety

- Chapter 2 mentions thread safety of shared-memory parallel functions or libraries.
  - A function or library is thread-safe if it operates "correctly" when called by multiple, simultaneously executing threads.
  - Since multiple threads communicate and coordinate through shared memory, a thread-safe code modifies the state of shared memory using appropriate synchronization.
  - Some features of sequential code that may not be thread safe?

Programming with Threads

Several thread libraries, more being created

- PThreads is the POSIX Standard
  - Relatively low level
    - Programmer expresses thread management and coordination
    - Programmer decomposes parallelism and manages schedule
  - Portable but possibly slow
  - Most widely used for systems-oriented code, and also used for some kinds of application code
- OpenMP is newer standard
  - Higher-level support for scientific programming on shared memory architectures
    - Programmer identifies parallelism and data properties, and guides scheduling at a high level
    - System decomposes parallelism and manages schedule
  - Arose from a variety of architecture-specific pragmas

Overview of POSIX Threads (Pthreads)

- POSIX: *Portable Operating System Interface for UNIX*
  - 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

Slide source: Jim Demmel and Kathy Yelick

Forking Pthreads

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 to nonzero if the create operation fails

Slide source: Jim Demmel and Kathy Yelick

Forking Pthreads, cont.

- The effect of pthread_create
  - Master thread actually causes the operating system to create a new thread
  - Each thread executes a specific function, thread_fun
    - The same thread function is executed by all threads that are created, representing the thread's *computation decomposition*
  - For the program to perform different work in different threads, the arguments passed at thread creation distinguish the thread's "id" and any other unique features of the thread.

Simple Threading Example

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

Compile using gcc … –lpthread

This code creates 16 threads that execute the function "ParFun".

Note that thread creation is costly, so it is important that ParFun do a lot of work in parallel to amortize this cost.

Slide source: Jim Demmel and Kathy Yelick

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
- Shared data often a result of creating a large "thread data" struct
  - Passed into all threads as argument
  - Simple example:

```
char *message = "Hello World!\n";
pthread_create( &thread1,
        NULL,
        (void*)&print_fun,
        (void*) message);
```

Slide source: Jim Demmel and Kathy Yelick

"Hello World" in Pthreads

- Some preliminaries
  - Number of threads to create (threadcount) is set at runtime and read from command line
  - Each thread prints "Hello from thread <X> of <threadcount>"
- Also need another function
  - `int pthread_join(pthread_t *, void **value_ptr)`
  - From Unix specification: "suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated."
  - The second parameter allows the exiting thread to pass information back to the calling thread (often NULL).
  - Returns nonzero if there is an error

Hello World! (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

declares the various Pthreads functions, constants, types, etc.

Hello World! (2)

```
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
} /* main */
```


Hello World! (3)

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
} /* Hello */
```

Explicit Synchronization: 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 object attribute; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

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

- This barrier could have been statically initialized by assigning an initial value created using the macro

```
PTHREAD_BARRIER_INITIALIZER(3).
```

Slide source: Jim Demmel and Kathy Yelick

Mutexes (aka Locks) in Pthreads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
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 deadlock:

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

Slide source: Jim Demmel and Kathy Yelick

Additional Pthreads synchronization described in textbook

- Semaphores
- Condition variables
- More discussion to come later in the semester, but these details are not needed to get started programming

Summary of Programming with Threads

- Pthreads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most programmers
  - Ability to shared data is convenient
- Pitfalls
  - Data races are difficult to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- **OpenMP** is commonly used today as a simpler alternative, but it is more restrictive
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence

OpenMP: Prevailing Shared Memory Programming Approach

- Model for shared-memory parallel programming
- Portable across shared-memory architectures
- Scalable (on shared-memory platforms)
- Incremental parallelization
  - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
  - Compiler generates thread program and synchronization
- Extensions to existing programming languages (Fortran, C and C++)
  - mainly by directives
  - a few library routines

See http://www.openmp.org

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 concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single process (**master thread**)
- Start of a parallel construct:
  - Master thread creates team of threads (**worker threads**)
- Completion of a parallel construct:
  - Threads in the team synchronize -- **implicit barrier**
- Only master thread continues execution
- Implementation optimization:
  - Worker threads spin waiting on next fork

OpenMP uses Pragmas

- Pragmas are special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.
- The interpretation of OpenMP pragmas
  - They modify the statement immediately following the pragma
  - This could be a compound statement such as a loop

#pragma omp …

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
  - Default is **shared**
  - Loop index is **private**

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

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```


OpenMP directive format C (also Fortran and C++ bindings)

- Pragmas, format

  `#pragma omp` directive_name [clause [clause] ...] new-line
- Conditional compilation

```
#ifdef _OPENMP
   block,
   e.g., printf("%d avail.processors\n",omp_get_num_procs());
#endif
```

- Case sensitive
- Include file for library routines

```
#ifdef _OPENMP
#include <omp.h>
#endif
```


OpenMP runtime library, Query Functions

`omp_get_num_threads`:

Returns the number of threads currently in the team executing the parallel region from which it is called

```
int omp_get_num_threads(void);
```

`omp_get_thread_num`:

Returns the thread number, within the team, that lies between `0` and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread `0`

```
int omp_get_thread_num(void);
```


OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the **same code redundantly (SPMD)**
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax

  `#pragma omp parallel` [clause [clause] ...] new-line

  structured-block
- clause can include the following:

  `private` (list)

  `shared` (list)

Hello World in OpenMP

- Let's start with a parallel region construct
- Things to think about
  - As before, number of threads is read from command line
  - Code should be correct without the pragmas and library calls
- Differences from Pthreads
  - More of the required code is managed by the compiler and runtime (so shorter)
  - There is an implicit thread identifier

gcc –fopenmp …

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```


In case the compiler doesn't support OpenMP

```
# include <omp.h>
```

```
#ifdef _OPENMP
# include <omp.h>
#endif
```


In case the compiler doesn't support OpenMP

```
# ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
# e l s e
   int my_rank = 0;
   int thread_count = 1;
# endif
```


OpenMP Data Parallel Construct: Parallel Loop

- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning of Res
- Synchronization also automatic (barrier)

Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel Program:

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```


Summary of Lecture

- OpenMP, data-parallel constructs only
  - Task-parallel constructs later
- What's good?
  - Small changes are required to produce a parallel program from sequential (parallel formulation)
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor
- What is missing?
  - Not completely natural if want to write a parallel code from scratch
  - Not always possible to express certain common parallel constructs
  - Locality management
  - Control of performance