CS4961 Parallel Programming

Lecture 12: Advanced Synchronization (Pthreads)

Mary Hall
October 4, 2011

Administrative

- Thursday's class
  - Meet in WEB L130 to go over programming assignment
- Midterm on Thursday October 20, in class
  - Review on Tuesday October 18
  - Now through Monday, Oct. 17, please send me questions for review
    - What would you like to discuss further on 10/18
  - Test format
    - 5 short definitions
    - 6 short answer
    - 3 problem solving
  - Opportunity: Submit questions that you think would be good exam questions to me before Wednesday AM, October 19 for either short answer or problem solving
    - I may use up to two of these!

Programming Assignment 2: Due Friday, Oct. 7

To be done on water.eng.utah.edu

In OpenMP, write a task parallel program that implements the following three tasks for a problem size and data set to be provided. For M different inputs, you will perform the following for each input

TASK 1: Scale the input data set by 2*(i+j)

TASK 2: Compute the sum of the data

TASK 3: Compute the average, and update max avg if it is greater than previous value

Like last time, I've prepared a template

Report your results in a separate README file.

- What is the parallel speedup of your code? To compute parallel speedup, you will need to time the execution of both the sequential and parallel code, and report speedup = Time(seq) / Time (parallel)
- You will be graded strictly on correctness. Your code may not speed up, but we will refine this later.
- Report results for two different numbers of threads.

Simple Producer-Consumer Example (from L9)

```
// PRODUCER: initialize A with random data
void fill_rand(int nval, double *A) {
for (i=0; i<nval; i++) A[i] = (double) rand()/1111111111;
}

// CONSUMER: Sum the data in A
double Sum_array(int nval, double *A) {
double sum = 0.0;
for (i=0; i<nval; i++)  sum = sum + A[i];
return sum;
}
```

Key Issues in Producer-Consumer Parallelism (from L9)

- Producer needs to tell consumer that the data is ready
- Consumer needs to wait until data is ready
- Producer and consumer need a way to communicate data
  - output of producer is input to consumer
- Producer and consumer often communicate through First-in-first-out (FIFO) queue

One Solution to Read/Write a FIFO (from L9)

- The FIFO is in global memory and is shared between the parallel threads
- How do you make sure the data is updated?
- Need a construct to guarantee ***consistent*** view of memory
  - Flush: make sure data is written all the way back to global memory

Example:

```
Double A;
A = compute();
Flush(A);
```


Solution to Producer/Consumer (from L9)

```
flag = 0;
#pragma omp parallel
{
   #pragma omp sections {
   #pragma omp section
   {
      fillrand(N,A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush(flag)
   }

   #pragma omp section
   {
      while (!flag)
         #pragma omp flush(flag)
      #pragma omp flush
      sum = sum_array(N,A);
   }
}
```


Is this a good way to parallelize this code?

- Flush has high overhead
- Task parallelism only supports 3 concurrent threads
- Computation does not have high granularity

- Purpose of assignment:
  - Understand the mechanisms
  - See the cost of synchronization
  - Use in subsequent assignment

Today's Lecture

- Read Chapter 4.5-4.9
  - All about synchronizing threads in Pthreads
- A primer on Pthreads and related synchronization

Summary of Lecture

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.
- Busy-waiting can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- A mutex can be used to avoid conflicting access to critical sections as well.
- A semaphore is the third way to avoid conflicting access to critical sections.
  - It is an unsigned int together with two operations: sem_wait and sem_post. Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.
- A barrier is a point in a program at which the threads block until all of the threads have reached it.
- A read-write lock is used when it's safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

Recall from Proj1: Pthreads Mutexes

- Used to guarantee that one thread "excludes" all other threads while it executes the critical section.
- The Pthreads standard includes a special type for mutexes: pthread_mutex_t.

```
int pthread_mutex_init(
      pthread_mutex_t*            mutex_p   /* out */
      const pthread_mutexattr_t*  attr_p    /* in  */);
```

- When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```


Mutexes

- To gain access to a critical section a thread calls

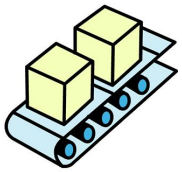
```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

- When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

Global sum that uses a mutex

```
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

PRODUCER-CONSUMER SYNCHRONIZATION AND SEMAPHORES

Semaphores for Producer-Consumer Parallelism

- The textbook uses semaphores to implement producer-consumer parallelism (Chapter 4.7)
- Definition: A semaphore is a special variable, accessed atomically, that controls access to a resource. A binary semaphore can take on the values of 0 or 1. It was named after the mechanical device that railroads use to control which train can use a track.
- We use binary semaphores in the following way:
  - Post – set the state of the semaphore to 1
  - Wait – wait until the state of the semaphore is 1
- This allows finer control than processors reaching a mutex

A first attempt at sending messages using pthreads

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

Syntax of the various semaphore functions

Semaphores are not part of Pthreads; you need to add this.

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);


int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```


Let's fix this with semaphores

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```


How would you do your assignment with semaphores?

TASK 1: Scale the input data set by 2*(i+j)
TASK 2: Compute the sum of the data
TASK 3: Compute the average, and update max avg if it is greater than previous value

BARRIERS AND CONDITION VARIABLES

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.
- In OpenMP, barriers are implicit at the end of each parallel construct
- Textbook shows how to implement barriers with semaphores
- Pthreads also has its own barriers

Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Busy-waiting and a Mutex

- Implementing a barrier using busy-waiting and a mutex is straightforward.
- We use a shared counter protected by the mutex.
- When the counter indicates that every thread has entered the critical section, threads can leave the critical section.

Busy-waiting and a Mutex

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

We need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```


Condition Variables

- A condition variable is a data object that allows a thread to suspend execution until a certain event or condition occurs.
- When the event or condition occurs another thread can signal the thread to "wake up."
- A condition variable is always associated with a mutex.

Condition Variables

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```


Implementing a barrier with condition variables

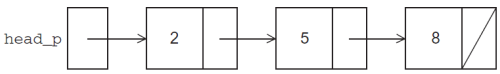
```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

READ-WRITE LOCKS

Controlling access to a large, shared data structure

- Let's look at an example.
- Suppose the shared data structure is a sorted linked list of ints, and the operations of interest are Member, Insert, and Delete.

Linked Lists

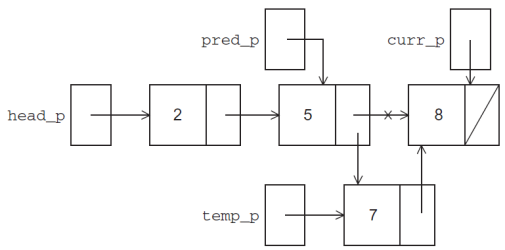



```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```


Linked List Membership

```
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

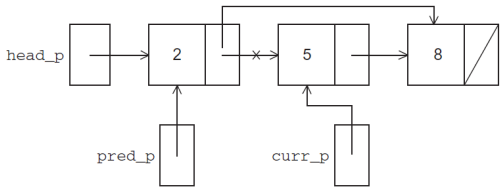
Inserting a new node into a list



Inserting a new node into a list

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
} /* Insert */
```


Deleting a node from a linked list



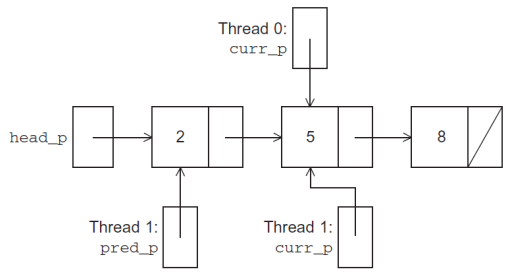
Deleting a node from a linked list

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_pp = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else {  /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

A Multi-Threaded Linked List

- Let's try to use these functions in a Pthreads program.
- In order to share access to the list, we can define head_p to be a global variable.
- This will simplify the function headers for Member, Insert, and Delete, since we won't need to pass in either head_p or a pointer to head_p: we'll only need to pass in the value of interest.

Simultaneous access by two threads



Solution #1

- An obvious solution is to simply lock the list any time that a thread attempts to access it.
- A call to each of the three functions can be protected by a mutex.

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

In place of calling Member(value).

Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A "finer-grained" approach.

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```


Issues

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.

Implementation of Member with one mutex per list node (1)

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }
```


Implementation of Member with one mutex per list node (2)

```
   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
- The first lock function locks the read-write lock for reading, while the second locks it for writing.

Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
- Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.

Pthreads Read-Write Locks

- If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Protecting our linked list functions

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```


THE UNIVERSITY OF UTAH