上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS433 Parallel and Distributed Computing

Lecture 4 OpenMP, Cont'd

Zhuoran Song

2023/9/26

Message Passing

- Suppose we have several "producer" threads and several "consumer" threads.
  - *Producer threads might "produce" requests for data.*
  - *Consumer threads might "consume" the request by finding or generating the requested data.*

- Each thread could have a shared message queue, and when one thread wants to "send a message" to another thread, it could enqueue the message in the destination thread's queue.

- A thread could receive a message by dequeuing the message at the head of its message queue.

Barrier

- One or more threads may finish allocating their queues before some other threads.
- We need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.
- After all the threads have reached the barrier all the threads in the team can proceed.

```
# pragma omp barrier
```

Synchronization

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (!Done())
    Try_receive();
```

Synchronization

```
    mesg = random();
    dest = random() % thread_count;
#   pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

Use synchronization mechanisms to update FIFO
What is the implementation of Enqueue?

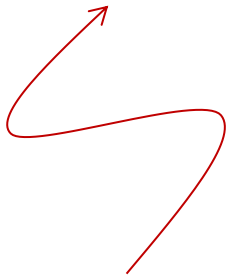
Synchronization

```
    if (queue_size == 0) return;
    else if (queue_size == 1)
#       pragma omp critical
        Dequeue(queue, &src, &mesg);
    else
        Dequeue(queue, &src, &mesg);
    Print_message(src, mesg);
```

This thread is the only one to dequeue its messages. Other threads may only add more messages. Messages added to end and removed from front. Therefore, only if we are on the last entry is synchronization needed.

Synchronization

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```



each thread increments this after completing its for loop

More synchronization needed on “done_sending”

Named Critical Sections

- Three critical sections:
  - *done_sending*
  - *Enqueue*
  - *Dequeue*

- Need to differential critical sections
  - *Using **Named** critical sections*

```
# pragma omp critical(name)
```

Atomic

- “done_sending” is critical，critical overhead
- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.

```
# pragma omp atomic
```

- Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

- What is the difference with reduction?

Atomic

- Here <op> can be one of the binary operators

  +, *, −, /, &, ^, |, <<, or >>

- Many processors provide a special load-modify-store instruction.

- A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

Locks

- How to differential in one critical section? (enqueue1, enqueue2…)
- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

Locks

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

Locks

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Summary of Mutual Exclusion

- **"critical"**
  - *Blocks of code*
  - *Easy to use*
  - *Limited named critical section (naming at compiler time)*
- **"atomic"**
  - *Single expression*
  - *Faster speed*
  - *Variable name differentiate different critical sections*
- **"lock"**
  - *Locks for data structures*
  - *Flexible to use*
  - *deadlock*

Deadlock

Thread 1

```
a += 5;
b += 7;
a += b;
a += 11;
```

Thread 2

```
b += 5;
a += 7;
a += b;
b += 11;
```

Deadlock

An implementation that can cause deadlock

Thread 1

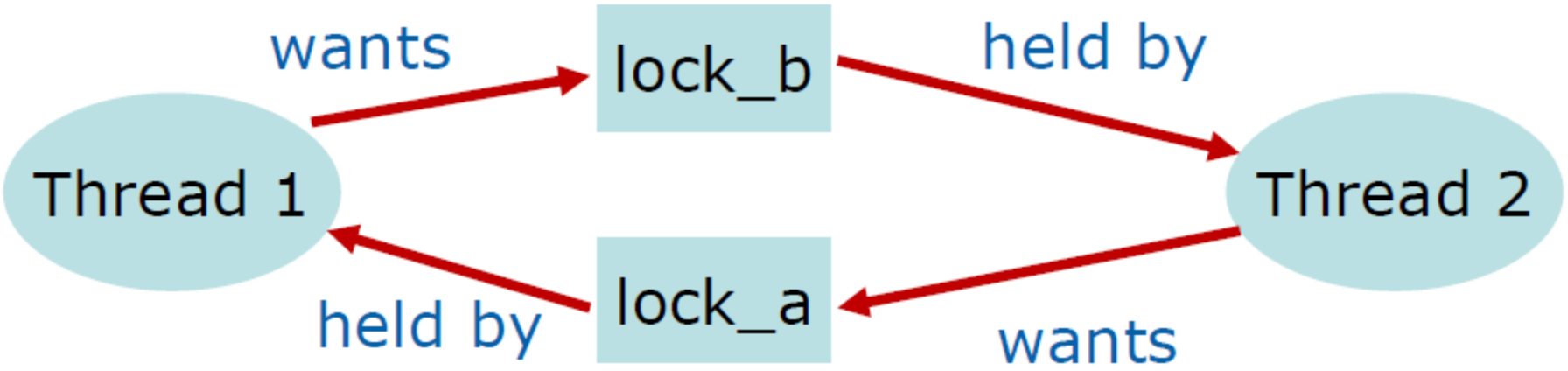
```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

Deadlock!

Thread 2

```
lock (lock_b);
b += 5;
lock (lock_a);
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Deadlock



A graph of deadlock contains a cycle

Deadlock

- A program exhibits a *global deadlock if every thread is blocked*

- A program exhibits *local deadlock if only some of the threads in the program are blocked*

- A deadlock is another example of a nondeterministic behavior exhibited by a parallel program

- Change timing can change deadlock occurring

Deadlock

Enforce order for locks to eliminate deadlock

Thread 1

```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

Thread 2

```
lock (lock_a);
lock (lock_b);
b += 5;
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Livelock

```
// Thread 1
getLocks12(lock1, lock2) {
  lock1.lock();
  while (lock2.locked()) {
    // attempt to yield to other thread
    lock1.unlock(); wait(); lock1.lock();
  } lock2.lock();
}
```

```
// Thread 2
getLocks21(lock2, lock1) {
  lock2.lock();
  while (lock1.locked()) {
    // attempt to yield to other thread
    lock2.unlock(); wait(); lock2.lock();
  } lock1.lock();
}
```

Introduce randomness in wait() can eliminate livelock

Recovery From Deadlock

So, the deadlock has occurred. Now, how do we get the resources back and gain forward progress?

- PROCESS TERMINATION:
  - *Could delete all the processes in the deadlock -- this is expensive.*
  - *Delete one at a time until deadlock is broken (time consuming).*
  - *Select who to terminate based on priority, time executed, time to completion, needs for completion, or depth of rollback*
  - *In general, it's easier to preempt the resource, than to terminate the process.*

- RESOURCE PREEMPTION:
  - *Select a victim - which process and which resource to preempt.*
  - *Rollback to previously defined "safe" state.*
  - *Prevent one process from always being the one preempted (starvation).*