Lecture 12: OpenMP

Producers/Consumers – Queues

- An abstraction of a line of customers waiting to pay for their groceries in a supermarket
- A natural data structure to use in many multithreaded applications
- For several "producer" and "consumer" threads
  - Producer threads might "produce" requests for data
  - Consumer threads might "consume" the request by finding or generating the requested data

Message-Passing

- 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

Message-Passing: Pseudo Code for Each Thread

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

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

Pseudo Code for Send_msg

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

Receiving Messages

- Only the destination queue-owner thread can Dequeue the enqueued message
- If the queue contains only a single message then Dequeue must be performed in a critical section so that no other thread can call Enqueue
- If the queue contains at least two messages then Dequeue can be performed without the use of a critical section

Receiving Messages

- A correct implementation of the previous synchonization scheme can be achieved by introducing shared variables enqueued and dequeued that counts the number of messages sent and received, respectively

Pseudo Code for Try_receive

- By introducing the derived variable

  queue_size = enqueued – dequeued

  the Try_receive function can be implemented as:

```
queue_size = enqueued – dequeued ;
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) ;
```

Pseudo Code for Try_receive

- Synchronization issues: recall enqueued and dequeued are shared variables
  - Suppose queue_size = 0 or 1 when the actual values should be 1 or 2
  - If queue_size = 0, Try_receive returns but tries again later
  - If queue_size = 1, Dequeue is called under a critical section but Try_receive is called again if the actual queue_size should have been 2

```
queue_size = enqueued – dequeued ;
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) ;
```

Termination Detection: The Done function

```
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*

Startup (1)

- When the program execution begins, the master thread, will get command line arguments and allocate an array of message queues: one for each thread
- This array will be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues

Startup (2)

- One or more threads may finish allocating their queues before some other threads
- An explicit barrier is needed to blocks all threads in the team that reach the barrier
- Once all threads have reached the barrier, all threads in the team can then proceed

```
#pragma omp barrier
```

Atomic Directive (1)

- 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;

Atomic Directive (2)

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

  +, *, −, /, &, ^, |, <<, or >>
- Note that <*expression*> <u>must not</u> reference x
- Example

  # pragma omp *atomic*

  x += y++

  a thread's update of x will be completed before any other thread can begin updating x

Atomic Directive (3)

- 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 used to protect more general critical sections

Critical (named) Directive

- OpenMP provides the option of adding a name to a *critical* directive:

  \# pragma omp *critical* (name)

- This allows the simultaneous execution of two blocks protected with *critical* directives with different names
- The names are set during compilation, so a different *critical* section is needed for each thread's queue

Locks

- 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 one thread */
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 ;
```

Message Passing Program: Using 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) ;
```

Message Passing Program: Using Locks

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

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

Some Caveats

- Do not mix the different types of mutual exclusion for a single critical section
- There is no guarantee of fairness in mutual exclusion constructs
- It can be dangerous to "nest" mutual exclusion constructs

Concluding Remarks (1)

- OpenMP is a standard for programming shared-memory systems
- OpenMP uses both special functions and preprocessor directives called pragmas
- OpenMP programs start multiple threads rather than multiple processes
- Many OpenMP directives can be modified by clauses

Concluding Remarks (2)

- A major problem in the development of shared memory programs is the possibility of race conditions
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Concluding Remarks (3)

- By default most systems use a block-partitioning of the iterations in a parallelized *for loop*
- OpenMP offers a variety of scheduling options
- In OpenMP the scope of a variable is the collection of threads to which the variable is accessible

Concluding Remarks (4)

- A reduction process repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.