InfoLab

IC621: Distributed and Parallel Computing

Lecture 08 : Shared-Memory Programming with OpenMP (II)

Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

InfoLab

PRODUCERS AND CONSUMERS

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Queues

- **List abstract datatype**
  - enqueue: new elements are inserted at the “rear” of the queue
  - dequeue: elements are removed from the “front” of the queue
  - *c.f.*, a line of customers waiting to pay for their groceries in a supermarket
    - the elements of the list are the customers
    - new customers go to the end or “rear” of the line
    - the next customer to check out is the one at the “front” of the line

- **Producers & consumers using queues**
  - several “producer” threads and several “consumer” threads
    - producer threads could enqueue the requested items
    - consumer threads could dequeue them

Message-passing

- **Message-passing on a shared-memory system**
  - each thread have a shared message queue
  - thread can "send a message" to another thread by enqueuing the message in the destination thread's queue
  - thread can "receive a message" by dequeuing the message at the head of its message queue
- **Example**
  - each thread generates random integer "messages" and random destinations for the messages
  - user specifies the number of messages each thread should send

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

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

Sending messages

- **Accessing a message queue to enqueue a message is a critical section**
  - because we need to check and update the rear pointer, when enqueuing a new message

- **Send_msg()**

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

Receiving messages

- **Only the owner of the queue (that is, the destination thread) dequeues from a given message queue**
  - `dequeued` variable is updated only by the owner of the queue
- **One thread can update enqueued while another thread is using it to compute `queue_size`**
  - it may compute a `queue_size` of 0 or 1 when `queue_size` should actually be 1 or 2
  - so, we need to execute the critical section directive when `queue_size` is 1
  - but, there is no need to do if `queue_size` is larger than 2

Try_receive()

```
    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

- **Naïve implementation of `Done()` will have problems**
  - if thread u executes the code, thread v could send a message to thread u after u has computed `queue_size = 0`
  - the message sent by thread v will never be received

```
queue_size = enqueued – dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

- **Correct implementation**
  - add a counter `done_sending`, and each thread increments this counter after completing the `for` loop for sending messages

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

Startup

- **Master thread allocates the array of queues**
- **We can start the threads using a `parallel` directive**
  - such that each thread allocate storage for its individual queue
- **We need to make sure that none of the threads starts sending messages until all the queues are allocated**
  - OpenMP directives provide *implicit barriers* when they're completed
  - but, we'll be in the middle of a `parallel` block in this case
  - so, we can't rely on an implicit barrier (we need an *explicit barrier*)
- **`barrier` directive**
  - when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier

```
# pragma omp barrier
```

The `atomic` directive

- **Each thread increments `done_sending` before proceeding to its final loop of receives**
- **Incrementing `done_sending` is a critical section**
  - we could protect it with a `critical` directive
  - OpenMP provides a potentially higher performance directive : `atomic` directive

`atomic` directive

```
# pragma omp atomic
```

- unlike the `critical` directive, it can only protect critical sections that consist of a single C assignment statement
- only the load and store of x are guaranteed to be protected
- <expression> must not reference x

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

<op> := +, *, -, /, &, ^, |, <<, or >>

Bad example

- update to x will be completed before any other thread updates x
- update to y may be unprotected and the results may be unpredictable

```
#     pragma omp atomic
      x += y++;
```

Critical sections and locks

- **Three blocks of code preceded by a `critical` or an `atomic` directive**
  - `done_sending++`
  - `Enqueue(q_p, my rank, mesg)`
  - `Dequeue(q_p, &src, &mesg)`
- **Default behavior of OpenMP about `critical` directive**
  - treat all critical blocks as part of one composite critical section
  - can be highly detrimental to the program's performance
- **OpenMP provides the option of adding a name to a `critical` directive**
  - two blocks protected with `critical` directives with different names can be executed simultaneously

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

- **Named critical directive isn't sufficient**
  - the names are set during compilation
  - we want a different critical section for each thread's queue
  - we need to set the names at run-time
  - we want to allow simultaneous access to the same block of code by threads accessing different queues
- **Alternative : using locks**
  - one of the threads (e.g., the master thread) initializes the lock
  - when all the threads are done using the lock, one of the threads destroys it

```
/* 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;
```

Two types of locks

- simple lock : can only be set once before it is unset
- nested lock : can be set multiple times by the same thread before it is unset

- `omp_lock_t`

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

Using locks in the message-passing program

- **We want to insure mutual exclusion in each individual message queue, not in a particular block of source code**
  - include a data member with type `omp_lock_t` in our queue struct
  - simply call `omp_set_lock` each time we want to insure exclusive access to a message queue

```
#  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);
```

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

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

Effects of using locks

- in original implementation, only one thread could send at a time, regardless of the destination
- but, now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue
  - since different message queues have different locks

`critical` directives, `atomic` directives, or locks?

- **Three mechanisms for enforcing mutual exclusion in a critical section**
- **`atomic` directives**
  - ➢ it has the potential to be the fastest method of obtaining mutual exclusion
  - ➢ OpenMP specification allows the `atomic` directive to enforce mutual exclusion across all `atomic` directives in the program
    - like the unnamed `critical` directive behaves
  - ➢ e.g., one thread executes the left while another executes the right
    - even if x and y are unrelated memory locations, if one thread is executing x++, then no thread can simultaneously execute y++

```
#  pragma omp atomic
   x++;
```

```
#  pragma omp atomic
   y++;
```

`critical` directive vs. locks

- both named and unnamed `critical` directives are very easy to use
- there is no large difference between the performance of
  - critical sections protected by a `critical` directive, and
  - critical sections protected by locks
- the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code

Some caveats

- **Shouldn't mix the different types of mutual exclusion for a single critical section**
  - the critical directive won't exclude the action executed by the atomic block
  - it's possible that the results will be incorrect
  - the programmer needs to either
    - rewrite the function g so that its use can have the form required by the atomic directive, or
    - protect both blocks with a critical directive
  - e.g.,

```
#  pragma omp atomic
   x += f(y);
```

```
#  pragma omp critical
   x = g(x);
```

- **No guarantee of fairness in mutual exclusion constructs**
  - it's possible that a thread can be blocked forever in waiting for access to a critical section
  - it's possible that thread 1 can block forever waiting to execute `x = g(my rank)`, while the other threads repeatedly execute the assignment
  - e.g.,

```
    while(1) {
        . . .
#       pragma omp critical
        x = g(my_rank);
        . . .
    }
```

Dangerous to “nest” mutual exclusion constructs

- e.g., the following code is guaranteed to deadlock
  - when a thread attempts to enter the second critical section, it will block forever
  - if thread u is executing code in the first critical block, no thread can execute code in the second block
  - in particular, thread u can’t execute this code
  - however, if thread u is blocked waiting to enter the second critical block, then it will never leave the first, and it will stay blocked forever

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

- we can solve the problem by using named critical sections

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

- however, there are many examples when naming won't help
  - if a program has two named critical sections, say one and two, and threads can attempt to enter the critical sections in different orders,
  - then deadlock can occur

Time	Thread *u*	Thread *v*
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

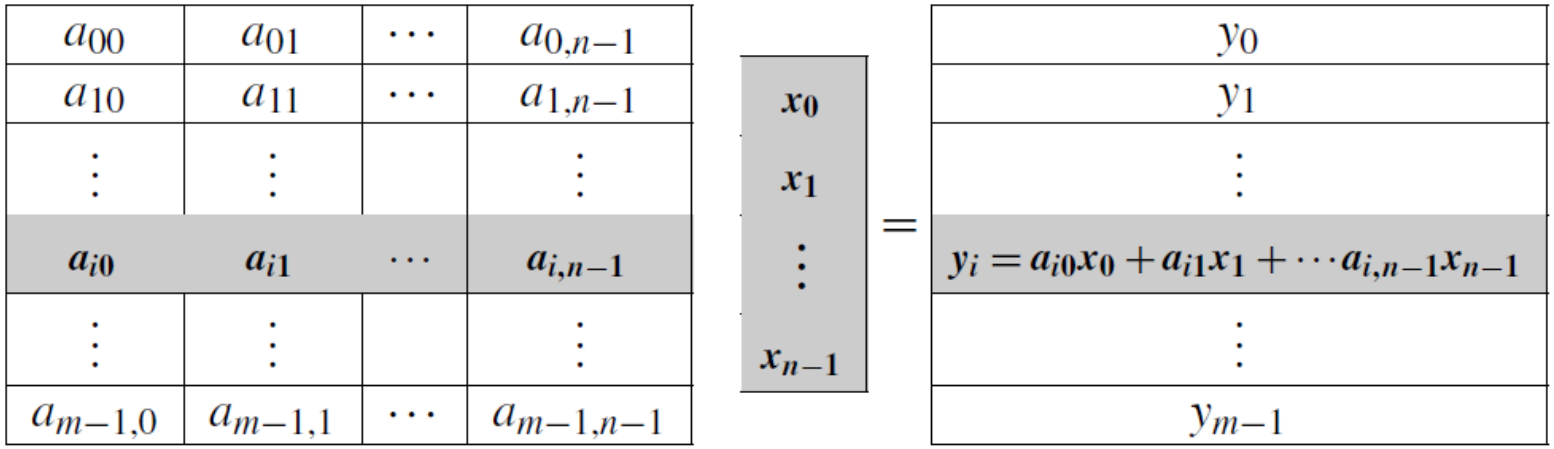
InfoLab

CACHES, CACHE COHERENCE, AND FALSE SHARING

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Matrix-vector multiplication

- **No loop-carried dependences in the outer loop**
  - A and x are never updated and iteration `i` only updates `y[i]`




```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

- **8M×8: a lot of write-misses occur at Line 4**
- **8×8M: a lot of read-misses occur at Line 6 (due to x)**
- **Bad efficiency of 8×8M due to false sharing**
  - cache coherence is enforced at the “cache-line level”
  - entire y fits in a single cache line (when y is double)

Table 5.4 Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

- **c.f., performance of using Pthreads**

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology