Parallel Computing

Lecture 15: OpenMP - IV

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
PRODUCERS AND CONSUMERS
Queues

• A natural data structure to use in many multithreaded applications.
• The two main operations: enqueue and dequeue
• For example, 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.
Example of Usage: 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.
Example of Usage: Message-Passing

Each thread executes the following:

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

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

mesg = random();
dest = random() % thread_count;

#pragma omp critical
Enqueue(queue, dest, my_rank, mesg);
Try_receive()

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

When queue size is 1, dequeue affects the tail pointer.
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 begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.

• This array needs to be shared among the threads.

• Each thread allocates its queue in the array.
Startup (2)

• 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.

```c
#pragma omp barrier
```
The Atomic Directive

• Higher performance than critical

• It can only protect critical sections that consist of a single C assignment statement.

The statement must have one of the following forms:

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

Must not reference `X`

+,*,-,/,&,^,|,<<,or>>
Critical Sections

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

- **OpenMP** provides the option of adding a name to a critical directive:
- **When we do this**, two blocks protected with critical directives with different names can be executed simultaneously.
- **However**, the names are set during compilation, and we may want a different critical section 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: main actions

/* 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: main actions

void omp_init_lock(omp_lock_t * lock_p);
void omp_set_lock(omp_lock_t * lock_p);
void omp_unset_lock(omp_lock_t * lock_p);
void omp_destroy_lock(omp_lock_t * lock_p);
Using Locks in the Message-Passing Program

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

/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, msg);
omp_unset_lock(&q_p->lock);
```
Using Locks in the Message-Passing Program

```c
#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);
```
Some Caveats

1. You shouldn’t mix the different types of mutual exclusion for a single critical section.
   - i.e. do not mix atomic and critical for the same variable update

2. There is no guarantee of fairness in mutual exclusion constructs.
   - A thread can be blocked forever!

3. It can be dangerous to “nest” mutual exclusion constructs.
Dividing Work Among Threads
Dividing Work Among Threads

```
#pragma omp parallel for
```

for_loop
Dividing Work Among Threads

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    structured_block

    #pragma omp section
    structured_block
}
```
#pragma omp parallel
{
  #pragma omp sections
  {{
    a=...;
    b=...;
  }}
  #pragma omp sections
  {
    c=...;
    d=...;
  }
  #pragma omp sections
  {
    e=...;
    f=...;
  }
  #pragma omp sections
  {
    g=...;
    h=...;
  }
} /*omp end sections*/
} /*omp end parallel*/
Tasks

- Feature added to 3.0 version of OpenMP
- A task is: an independent unit of work
- A thread is assigned to perform a task.

Source: Ruud van der Pass
SC’13
#pragma omp parallel {

#pragma omp single {
    node *p = head_of_list;
    while (p) {
        #pragma omp task private(p)
        process(p);
        p = p->next;
    } // end while
} // end pragma single
} // end pragma parallel  

Implicit barrier
Task Synchronization

#pragma omp barrier

#pragma omp taskwait
• explicitly waits on the completion of child tasks
Example:
Write a program that prints either "A Race Car" or "A Car Race"

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

int main(int argc, char *argv[]) {
    printf("A ");
    printf("race ");
    printf("car ");

    printf("\n");
    return(0);
}
```

$ cc -fast hello.c
$ ./a.out
A race car

What will this program print?

Source:
Ruud van der Pass
SC’13
Example:
Write a program that prints either “A Race Car” or “A Car Race”

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

int main(int argc, char *argv[]) {

    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    } // End of parallel region

    printf("\n");
    return(0);
}
```

What will this program print using 2 threads?

Source:
Ruud van der Pass
SC’13
Example:
Write a program that prints either “A Race Car” or “A Car Race”

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

int main(int argc, char **argv) {

    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            printf("race ");
            printf("car ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
```

What will this program print using 2 threads?

Source:
Ruud van der Pass
SC’13
Example:
Write a program that prints either “A Race Car” or “A Car Race”

```c
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race ");}
            #pragma omp task
            {printf("car ");}
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
```

Source:
Ruud van der Pass
SC’13
Example:
Write a program that prints either “A Race Car” or “A Car Race”

```c
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {printf("race ");}
            #pragma omp task
            {printf("car ");}
            printf("is fun to watch ");
        }
    } // End of parallel region

    printf("\n");
    return(0);
}
```

What will this program print using 2 threads?

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race

Source:
Ruud van der Pass
SC’13
Example:
Write a program that prints either "A Race Car" or "A Car Race"

```c
int main(int argc, char **argv)
{
#pragma omp parallel
{
    #pragma omp single
    {
        printf("A ");
        #pragma omp task
        {printf("car ");}
        #pragma omp task
        {printf("race ");}
        #pragma omp taskwait
        printf("is fun to watch ");
    }
} // End of parallel region

printf("\n"); return(0);
}
```

What will this program print using 2 threads?

A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
Dividing Work Among Threads

Specifies that the enclosed code is to be executed by only one thread in the team.

```plaintext
#pragma omp single [clause ...]
structured_block
```
Conclusions

- We have seen three mechanisms to enforce mutual exclusion: atomic, critical, and locks
  - atomic is fastest but with limitations
  - critical can name sections but at compile time
  - locks better be used when we need mutual exclusion for data structure instead of code