## Table of Contents

1. **Producer-Consumer Model**
   - Startup, Atomic, Critical Sections, Locks
   - Producer/Consumer Code Example
2. **Summary/Concluding Remarks**
3. **Next Time**
Common use of task parallelism = Producer/consumer

Producer-Consumer Benefit from Use of Queues

**Queues**

- Can be viewed as 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 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.
First in First Out (FIFO) queue

Solving Producer/Consumers Problem with Message Passing

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

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

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

- Any thread can enqueue messages into another thread's queue
- Need to know front/head and back/tail of queue (FIFO)
- Need to develop an Enqueue function
- Critical Block to control access to front/head/and messages (send or recv)
Pseudo code for Send_msg()

Sending Messages

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

- Only owner can dequeue messages, so synchronization important.
- Need to develop a Dequeue function that controls access to
- Critical Block to control access to front/head/and messages (send or recv)
  - use two variables count number of messages enqueued and dequeued.
  - $queue\_size = enqueued - dequeued$
A thread is responsible for dequeuing its messages. Other threads can only add messages.
Messages added to end and removed from front (FIFO).
Synchronization needed after last entry.
Producer-Consumer Queues

Thread Queue Program Startup

- At start of execution, one thread (e.g. master) gets command line arguments.
- Master allocates array of message queues: one for each thread.
- Array needs to 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.
Message Queue Data

- list of messages
- pointer to rear of queue
- pointer to front of queue
- count of messages dequeued
Thread Queue Program Startup

- Threads are started with a parallel block directive
- In some cases, one or more threads may finish allocating their queues before some other threads.
- Explicit barrier is needed to block threads until all threads in team have arrived.
- After all the threads have reached the barrier all the threads in the team can proceed.
- Accomplished by using:  
  `# pragma omp barrier`
Source: http://en.wikipedia.org/wiki/OpenMP
Thread Queue Program complete

- When done, threads increments `done_sending`
- This is a critical section:  `# pragma omp critical`
- But, OpenMP has intrinsic function:  `# pragma omp atomic`
- higher performance but limited
The Atomic Directive (1)

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.
  
  ```c
  # pragma omp atomic
  ```

- Further, the statement must have one of the following forms:
  
  ```c
  x <op>= <expression>;
  x++; 
  ++x;
  x--; 
  --x;
  ```
Atomic Directive

\[ x < op > = < expression > \]

- Only the load and store of \( x \) are guaranteed to be protected
- What happens here?

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

- \( x \) update is protected: \( < op > \) is a \(+\) operation, so it is protected.
- \( y++ \) update may not be safe
The Atomic Directive (2)

- Here <op> can be one of the binary operators:
  
  \(+, \ast, -, /, \&, ^, |, \langle\langle, \text{ or } \rangle\rangle\)

- 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.
A Deeper Look at Critical Sections

- For the case of multiple critical sections, OpenMP provides the option of adding a name to a critical directive:

  \[ \# \text{pragma omp critical}(\text{name}) \]

- Parallelism: two blocks protected with critical directives with different names can be executed simultaneously.

- Problem: names are set during compilation - how to set different critical section for each threads queue?

  \[ \Rightarrow \text{Locks} \]
OpenMP Locks

- A lock is composed of a *data structure* and *functions* that allow the programmer to explicitly enforce mutual exclusion in a critical section.
- It is shared among threads that will exec critical section
  - one thread will init the lock (e.g. master)
  - other threads will try to enter the block:
    - on success, a thread *obtains* the lock
    - when done the thread will *relinquish* the lock.
  - the last thread using lock will destroy lock
- Two types of locks:
  - *simple*: can only be set once before it is unset
  - *nested*: can be multiple times by same thread before it is unset
OpenMP Lock Functions

void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
initializes lock; the initial state is unlocked, for the
nestable lock the initial count is zero. These functions
should be called from serial portion.
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
the argument should point to initialized lock variable that is unlocked
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
ownership of lock is granted to the thread executing the function; with
nestable lock the nesting count is incremented if the (simple) lock is set
when the function is executed the requesting thread is blocked until the
lock can be obtained
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
the argument should point to initialized lock in possession of the invoking thread,
otherwise the results are undefined. For the nested lock the function decrements
the nesting count and releases the ownership when the count reaches 0
OpenMP Lock Functions

```c
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

these functions attempt to get the lock in the same way as `omp_set_(nest)_lock`, except these functions are non-blocking.

for a simple lock, the function returns non-zero if the lock is successfully set, otherwise it returns 0

for a nestable lock, the function returns the new nesting count if the lock is successfully set, otherwise it returns 0.

Code Example:
```c
#include <omp.h>

omp_lock_t *lck;
omp_init_lock(lck);

/* spin until the lock is granted */
while( !omp_test_lock(lck));
```
Locks.

```c
/* 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;
```
Using Locks with Queue-based Message-Passing Programs.

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

/* q_p = msg_queues[dest] */
onp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
onp_unset_lock(&q_p->lock);
```
Message Queue Structure Data

- list of messages
- pointer to rear of queue
- pointer to front of queue
- count of messages dequeued
- lock variable: `omp_lock_t`
Which is best?

- We have looked at the *critical* and *atomic* directives and *locks*.
- *atomic*: fast because it only does one thing; but OpenMP allows all atomic directives to enforce mutual exclusion across all atomic directives.
- *critical*: easy to use, for multiple they should be named.
- *locks*: performance and function similar to named critical; best used for structures
Caveats

- You shouldn’t mix the different types of mutual exclusion for a single critical section - they don’t share exclusive actions:

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

- There is no guarantee of fairness in mutual exclusion constructs.

- It can be dangerous to nest mutual exclusion constructs.

  ```
  #  pragma omp critical
  y  +=  f(x)
  ```

  ..

  ```
  double f(double x) { 
    #pragma omp critical
    z  +=  g(x) /* z is shared */
  }
  ```
Caveats (2)

- Can improve things with naming
  ```
  #pragma omp critical(one)
  y  +=  f(x)
  ```
  ```
  double f(double x) {
    #pragma omp critical(two)
    z  +=  g(x) /* z is shared */
  }
  ```

- but this won’t help in all situations; example of **deadlock**

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread u</th>
<th>Thread v</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Enter crit. sect. one</td>
<td>Enter crit. sect. two</td>
</tr>
<tr>
<td>1</td>
<td>Attempt to enter two</td>
<td>Attempt to enter one</td>
</tr>
<tr>
<td>2</td>
<td>Block</td>
<td>Block</td>
</tr>
</tbody>
</table>
Main

/* File: producer_consumer.c
 * Purpose: Implement a producer-consumer program in which some of the threads are producers
 and others are consumers. The producers read text from a collection of files, one per producer.
 They insert lines of text into a single shared queue. The consumers take the lines of text and
 tokenize them -- i.e., identify strings of characters separated by whitespace from the rest of the
 line. When a consumer finds a token, it writes to stdout.
 */

int main(int argc, char* argv[]) {
    int prod_count, cons_count;
    FILE* files[MAX_FILES];
    int file_count;

    if (argc != 3) Usage(argv[0]);
    prod_count = strtol(argv[1], NULL, 10);
    cons_count = strtol(argv[2], NULL, 10);

    /* Read in list of filenames and open files */
    Get_files(files, &file_count);

    # ifdef DEBUG
    printf("prod_count = %d, cons_count = %d, file_count = %d\n",
           prod_count, cons_count, file_count);
    # endif

    /* Producer-consumer */
    Prod_cons(prod_count, cons_count, files, file_count);

    return 0;
} /* main */
/ * Function: Prod_cons * Purpose: Divides tasks among threads */
void Prod_cons(int prod_count, int cons_count, FILE* files[], int file_count) {
    int thread_count = prod_count + cons_count;
    struct list_node_s* queue_head = NULL; struct list_node_s* queue_tail = NULL;
    int prod_done_count = 0;

    # pragma omp parallel num_threads(thread_count) default(none) \
    shared(file_count, queue_head, queue_tail, files, prod_count, \
           cons_count, prod_done_count)
    { int my_rank = omp_get_thread_num(), f;
        if (my_rank < prod_count) { /* Producer code */
            /* A cyclic partition of the files among the producers */
            for (f = my_rank; f < file_count; f += prod_count) {
                Read_file(files[f], &queue_head, &queue_tail, my_rank);
            }

            # pragma omp atomic
            prod_done_count++;
        } else { /* Consumer code */
            struct list_node_s* tmp_node;

            while (prod_done_count < prod_count) {
                tmp_node = Dequeue(&queue_head, &queue_tail, my_rank);
                if (tmp_node != NULL) {
                    Tokenize(tmp_node->data, my_rank);
                    free(tmp_node);
                }

                while (queue_head != NULL) {
                    tmp_node = Dequeue(&queue_head, &queue_tail, my_rank);
                    if (tmp_node != NULL) {
                        Tokenize(tmp_node->data, my_rank);
                        free(tmp_node);
                    }
                }
            }
        }
    } /* pragma omp parallel */
} /* Prod_cons */
Producer-Consumer Code Example

Read_file

/*--------------------------------------------------------------------
* Function: Read_file
* Purpose: read text line from file into the queue linkedlist
* In arg: file, my_rank
* In/out arg: queue_head, queue_tail
*/

void Read_file(FILE* file, struct list_node_s** queue_head,  
               struct list_node_s** queue_tail, int my_rank) {

    char* line = malloc(MAX_CHAR*sizeof(char));
    while (fgets(line, MAX_CHAR, file) != NULL) {
        printf("Th %d > read line: %s", my_rank, line);
        Enqueue(line, queue_head, queue_tail);
        line = malloc(MAX_CHAR*sizeof(char));
    }

    fclose(file);
} /* Read_file */
Producer-Consumer Model

Producer/Consumer Code Example

Enqueue

/*--------------------------------------------------------------------
* Function:    Enqueue
* Purpose:     create data node, add into queue linkedlist
* In arg:      line
* In/out arg:  queue_head, queue_tail
*/
void Enqueue(char* line, struct list_node_s** queue_head, 
             struct list_node_s** queue_tail) {
    struct list_node_s* tmp_node = NULL;
    tmp_node = malloc(sizeof(struct list_node_s));
    tmp_node->data = line;
    tmp_node->next = NULL;
    #pragma omp critical
    if (*queue_tail == NULL) { // list is empty
        *queue_head = tmp_node;
        *queue_tail = tmp_node;
    } else {
        (*queue_tail)->next = tmp_node;
        *queue_tail = tmp_node;
    }
} /* Enqueue */
Producer-Consumer Code Example

Dequeue

```c
/*-----------------------------*/
* Function:   Dequeue
* Purpose:    remove a node from queue linkedlist and tokenize them
* In arg:     my_rank
* In/out arg: queue_head, queue_tail
* Ret val:    Node at head of queue, or NULL if queue is empty
*/
struct list_node_s* Dequeue(struct list_node_s** queue_head,
                            struct list_node_s** queue_tail, int my_rank) {
  struct list_node_s* tmp_node = NULL;
  if (*queue_head == NULL) // empty
    return NULL;

  # pragma omp critical
  {
    if (*queue_head == *queue_tail) // last node
      *queue_tail = (*queue_tail)->next;

    tmp_node = *queue_head;
    *queue_head = (*queue_head)->next;
  }

  return tmp_node;
} /* Dequeue */
```
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.

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

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.

A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
OpenMP Interface Model

Directives and pragmas
- Control structures
- Work sharing
- Synchronization
- Data scope attributes:
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

Runtime library routines
- Control and query routines:
  - number of threads
  - throughput mode
  - nested parallelism
- Lock API

Environment Variables
- Runtime environment:
  - schedule type
  - max #threads
  - nested parallelism
  - throughput mode
Run Time Library

   subroutine omp_set_num_threads(scalar)
   sets the number of threads to use for subsequent parallel region
   integer function omp_get_num_threads()
   should be called from parallel segment. Returns \# of threads currently executing
   integer function omp_get_max_threads()
   can be called anywhere in the program. Returns max number of threads that can
   be returned by omp_get_num_threads()
   integer function omp_get_thread_num()
   returns id of the thread executing the function. The thread id lies in between 0 and
   omp_get_num_threads()-1
   integer function omp_get_num_procs()
   maximum number of processors that could be assigned to the program
   logical function omp_in_parallel()
   returns .TRUE. (non-zero) if it is called within dynamic extent of a parallel region
   executing in parallel; otherwise it returns .FALSE. (0).
   subroutine omp_set_dynamic(logical)
   logical function omp_get_dynamic()
   query and setting of dynamic thread adjustment; should be called only from serial
   portion of the program
Next class: 12/04/14
HW #5 (OpenMP) Due: TBA
HW #6 (GPU/CUDA) Due: TBA
No more quizzes, only projects.