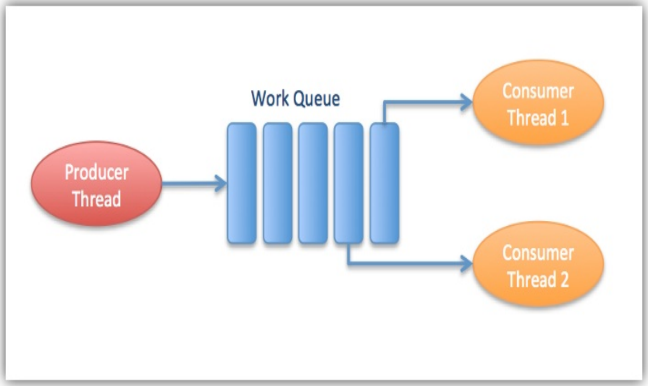
COMP 605: Introduction to Parallel Computing
Topic: OpenMP: Producer-Consumers

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Presented: 04/18/17
Last Update: 04/18/17

Producer-Consumers: Queueing Systems



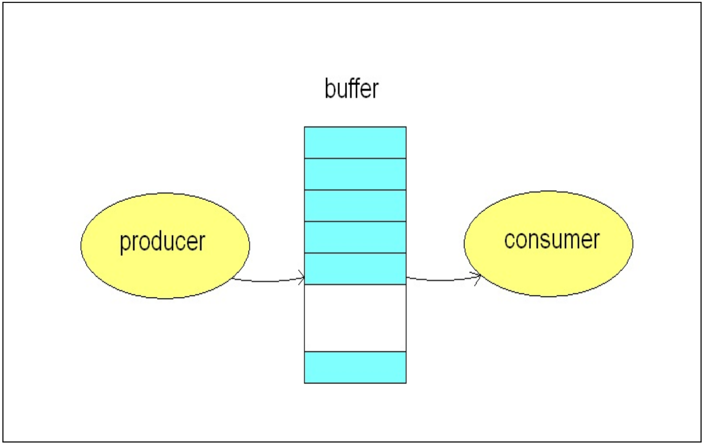
Producer-Consumer Benefit from Use of Queues

Queues

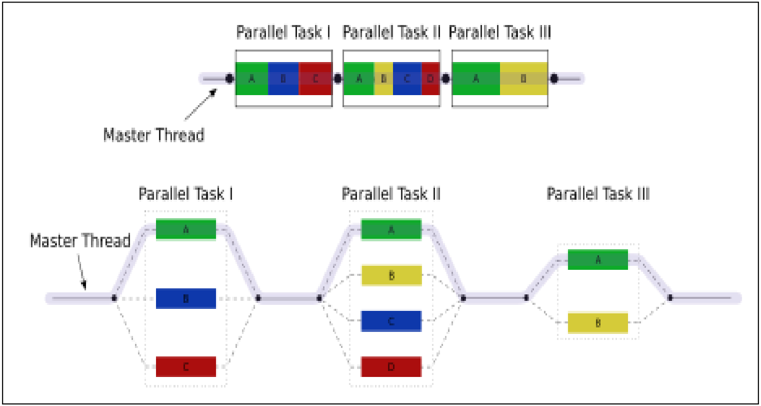
- Can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.
- A natural data structure to use in many multithreaded applications.
- For example, suppose we have several "producer" threads and several "consumer" threads.
  - Producer threads might "produce" requests for data.
  - Consumer threads might "consume" the request by finding or generating the requested data.

Copyright © 2010, Elsevier Inc. All rights Reserved

First in First Out (FIFO) queue



Source: `http://www.cs.oberlin.edu/~jdonalds/341/lecture04.html`



Source: `http://en.wikipedia.org/wiki/OpenMP`

Using Queues for 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.

Copyright © 2010, Elsevier Inc. All rights Reserved

Message-Passing

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

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Sending Messages

- Any thread can enqueue messages into another threads 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

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

Copyright © 2010, Elsevier Inc. All rights Reserved

Receiving Messages

- Only owner can dequeue messages, so synchronization important.
- Need to develop a Dequeue function that controls access to messages.
- Critical Block to control access to front/head/and messages (send or recv)
  - use two variables count number of messages enqueued and dequeued.
  - $queue\_size = enqueued - dequeued$

Pseudo code for Try_receive()

Receiving Messages

```
   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);
```

Copyright © 2010, Elsevier Inc. All rights Reserved

A thread is responsible for dequeueing its messages. Other threads can only add messages.
Messages added to end and removed from front (FIFO).
Synchronization needed after last entry.

Message Queue Data

- list of messages
- pointer to rear of queue
- pointer to front of queue
- count of messages dequeued

Thread Queue Program complete

- When done, threads increments *done_sending*
- This is a critical section: # *pragma omp critical*
- OpenMP has intrinsic function: # *pragma omp atomic*
- higher performance but limited

Thread Queue Program Startup

- Threads are started with a parallel block directive
- At start of execution, one thread gets command line arguments
- Master allocates *array of message queues*: one for each thread.
  - array needs to be shared among the threads
  - any thread can send to any other thread
  - any thread can enqueue a message in any of the queues.

Thread Queue Program Startup

- Queue elements:
  - list of messages
  - pointer index to rear of queue
  - pointer index to front of queue
  - count of enqueued messages
  - count of dequeue messages

Thread Queue Program Startup

- 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*
- When thread is done, it increments its *done_sending* variable, which is a critical section.

OpenMP Execution Model

sgi™

OpenMP Program starts like any sequential program:

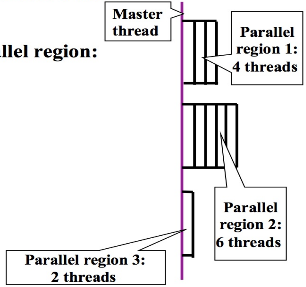
single threaded

***To create additional threads user starts a* parallel region**

- **Additional slave threads are launched to create a team**
- **Master thread is part of the team**
- **Threads "go away" at the end of the parallel region: usually sleep or spin**

Repeat parallel regions as necessary

- **Fork-join model**



Source: http://sc.tamu.edu/help/power/powerlearn/presentations/OpenMPnw.ppt

The 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;
```

Copyright © 2010, Elsevier Inc. All rights Reserved

The Atomic Directive (2)

- 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.

Copyright © 2010, Elsevier Inc. All rights Reserved

Atomic Directive

$$x < op > = < expression >$$

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

  $\# \; pragma \; omp \; atomic$

  $$x \quad += \qquad y++$$

- x update is protected: $< op >$ is a $+$ operation, so it is protected.
- $y++$ update may not be safe

```
/* File:     queue_lk.c
 * Purpose:  Implement a queue with locks using a linked list and OpenMP.
 *           Operations are Enqueue, Dequeue, Print, Search, and Free.
 *
 * To be used with omp_msglk.c
 *
 * Compile:  gcc -g -Wall -DUSE_MAIN -fopenmp -o queue_lk queue_lk.c
 *
 * Usage:    ./queue_lk
 *
 * Input:    Operations (first letter of op name) and, when necessary, keys
 * Output:   Prompts for input and results of operations
 *
 * IPP:      Section 5.8.9 (pp. 248 and ff.)
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "queue_lk.h"
#ifdef USE_MAIN
int main(void) {
   char op;
   int  src, mesg, not_empty;
   struct queue_s* q_p = Allocate_queue();

    printf("Op? (e, d, p, s, f, q)\n");
   scanf(" %c", &op);   switch (op) {
         case 'e':
         case 'E':
            printf("Src? Mesg?\n");
             scanf("%d%d", &src, &mesg);
            omp_set_lock(&q_p->lock);
            Enqueue(q_p, src, mesg);
            omp_unset_lock(&q_p->lock);
            break;

         case 'd':
         case 'D':
            omp_set_lock(&q_p->lock);
            not_empty = Dequeue(q_p, &src, &mesg);
            omp_unset_lock(&q_p->lock);
            if (not_empty)
               printf("Dequeued src = %d,
                    mesg = %d\n", src, mesg);
            else
               printf("Queue is empty\n");
            break;
```

```
         case 's':
           case 'S':
              printf("Mesg?\n");  scanf("%d", &mesg);
              if (Search(q_p, mesg, &src))
                 printf("Found %d from %d\n", mesg, src);
              else
                 printf("Didn't find %d\n", mesg);
              break;
           case 'p':
           case 'P':
              Print_queue(q_p);      break;
           case 'f':
           case 'F':
              omp_set_lock(&q_p->lock);
              Free_queue(q_p);
              omp_unset_lock(&q_p->lock);
              break;
           default:
              printf("%c isn't a valid command\n", op);
              printf("Please try again\n");
        }  /* switch */
        printf("Op? (e, d, p, s, f, q)\n");  scanf(" %c", &op);
     }  /* while */
     Free_queue(q_p);
     omp_destroy_lock(&q_p->lock);
     free(q_p);
     return 0;
}  /* main */
```

```
struct queue_s* Allocate_queue() {
   struct queue_s* q_p = malloc(sizeof(struct queue_s));
   q_p->enqueued = q_p->dequeued = 0;
   q_p->front_p = NULL;
   q_p->tail_p = NULL;
   omp_init_lock(&q_p->lock);
   return q_p;
}  /* Allocate_queue */
```

```
/* Frees nodes in queue:  leaves queue struct allocated and lock
 * initialized */
void Free_queue(struct queue_s* q_p) {
   struct queue_node_s* curr_p = q_p->front_p;
   struct queue_node_s* temp_p;

   while(curr_p != NULL) {
      temp_p = curr_p;
      curr_p = curr_p->next_p;
      free(temp_p);
   }
   q_p->enqueued = q_p->dequeued = 0;
   q_p->front_p = q_p->tail_p = NULL;
}   /* Free_queue */
```

```
void Print_queue(struct queue_s* q_p) {
   struct queue_node_s* curr_p = q_p->front_p;

   printf("queue = \n");
   while(curr_p != NULL) {
      printf("   src = %d, mesg = %d\n", curr_p->src, curr_p->mesg);
      curr_p = curr_p->next_p;
   }
   printf("enqueued = %d, dequeued = %d\n", q_p->enqueued, q_p->dequeued);
   printf("\n");
}  /*  Print_Queue */
```

```
void Enqueue(struct queue_s* q_p, int src, int mesg) {
   struct queue_node_s* n_p = malloc(sizeof(struct queue_node_s));
   n_p->src = src;
   n_p->mesg = mesg;
   n_p->next_p = NULL;
   if (q_p->tail_p == NULL) { /* Empty Queue */
      q_p->front_p = n_p;
      q_p->tail_p = n_p;
   } else {
      q_p->tail_p->next_p = n_p;
      q_p->tail_p = n_p;
   }
   q_p->enqueued++;
}  /* Enqueue */
```

```
int Dequeue(struct queue_s* q_p, int* src_p, int* mesg_p) {
   struct queue_node_s* temp_p;

   if (q_p->front_p == NULL) return 0;
   *src_p = q_p->front_p->src;
   *mesg_p = q_p->front_p->mesg;
   temp_p = q_p->front_p;
   if (q_p->front_p == q_p->tail_p)  /* One node in list */
      q_p->front_p = q_p->tail_p = NULL;
   else
      q_p->front_p = temp_p->next_p;
   free(temp_p);
   q_p->dequeued++;
   return 1;
}  /* Dequeue */
```

```
int Search(struct queue_s* q_p, int mesg, int* src_p) {
   struct queue_node_s* curr_p = q_p->front_p;
   while (curr_p != NULL)
      if (curr_p->mesg == mesg) {
         *src_p = curr_p->src;
         return 1;
      } else {
         curr_p = curr_p->next_p;
      }
   return 0;
}  /* Search */
```

Critical Sections

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

  # pragma omp critical(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?

  $\Longrightarrow$ **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 set multiple times by same thread before it is unset.

OpenMP Lock Functions

Thread Based Parallelism:

```
void omp_init_lock(omp_lock_t *lock):  Initializes a lock
                    associated with the lock variable

void omp_set_lock(omp_lock_t *lock):  Acquires ownership of a lock

void omp_destroy_lock(omp_lock_t *lock): Disassociates the given lock variable
                    from any locks

int omp_test_lock(omp_lock_t *lock):        Attempts to set a lock, but does not
                   block if the lock is unavailable.
```

Nested Parallelism:

```
The API provides for the placement of parallel regions
inside other parallel regions. Implementations may or may not support this feature.

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_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_destroy_nest_lock(omp_nest_lock_t *lock);  the argument should
    point to initialized lock variable that is unlocked
```

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;
```

OpenMP Lock Functions

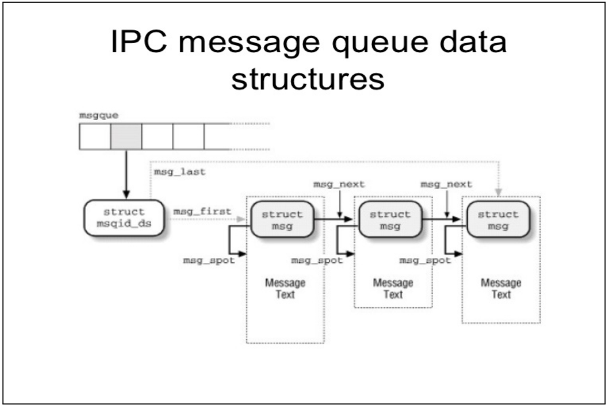
```
Code Example:
#include <omp.h>
. . .
omp_lock_t *lck;
. . .
omp_init_lock(lck);
. . .
/* spin until the lock is granted */
while( !omp_test_lock(lck));
{
        do some work
}
omp_destroy_lock(lck);
. . .
```

Using Locks with Queue-based Message-Passing Programs.

```
# 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 Queue Data Structure Example



Source: `https://image.slidesharecdn.com/ipcinlinux-120215010012-phpapp01/95/ipc-in-linux-34-728.jpg?cb=1329269028`

Message Queue Structure Data

- list of messages
- pointer to rear/end of queue
- pointer to front/start of queue
- pointer to next message in queue
- count of messages dequeued
- lock variable: *omp_lock_t*

Which access control mechanism 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

Things to consider

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

$$\begin{array}{rcl} \#pragma\ omp\ atomic & & \\ x & += & f(y) \\ \#\ pragma\ omp\ critical & & \\ x & = & g(x) \end{array}$$

- The second critical block does not contain a valid OMP expression, so it has *critical*, but that does not override what the *atomic* block can do so it is not protected.
- either rewrite *g(x)*, or use two critical blocks.

More things to consider

- There is no guarantee of fairness in mutual exclusion constructs: a thread can be blocked forever waiting for the unlock to occur.

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

- It can be dangerous to nest mutual exclusion constructs: deadlock example:

```
#  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)*

  $y \quad += \quad f(x)$

  .....

  double f(double x) {

  \#*pragma omp critical(two)*

  $z \quad += \quad g(x)$ /* z is shared */

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

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

Main

```
/* File:     producer_comsumer.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 */
```

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 */
```

Enqueue

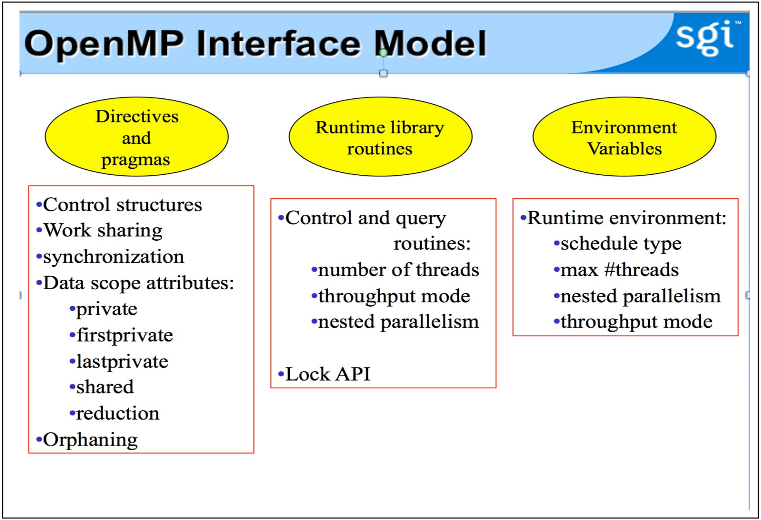
```
/*-------------------------------------------------------------------
 * Function:    Enqueue
 * Purpose:     create data node, add into queue linkedlist
 * In arg:      line
 * In/out arg:  queue_head, queue_tail
 */
void Enqueue(char* line, struct list_node_s** queue_head,
      struct list_node_s** 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 */
```

Dequeue

```
  /*-------------------------------------------------------------------
 * 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 */
```

Summary and Conclusions

- 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
sgi
Directives and pragmas
Runtime library routines
Environment Variables
•Control structures
•Work sharing
•synchronization
•Data scope attributes:
•private
•firstprivate
•lastprivate
•shared
•reduction
•Orphaning
•Control and query routines:
•number of threads
•throughput mode
•nested parallelism
•Lock API
•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
```