CS315A/EE382B: Lecture 3

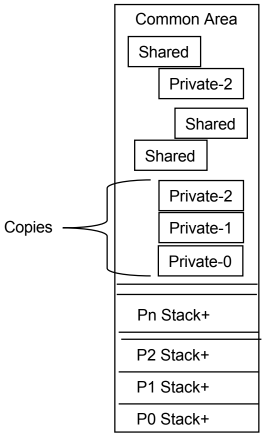
Application Parallelization I: Tasks

Kunle Olukotun Stanford University

http://eeclass.stanford.edu/cs315a

Review: “Lightweight” Thread Model

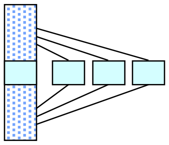
Common **Virtual** Address Space



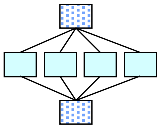
- Each thread is just a PC, registers, and stack
  - Often made with `pthread_create()`
- Usually *all* memory shared
  - Same page table, so no separation
  - Globals are completely shared
- **Pros:**
  - Easier sharing
    - No need for separate `malloc()` calls
    - Now pointer usage controls sharing
  - *Much* less OS overhead!!!
- **Con:** Non-shared data just by copying vars
  - Pointer errors may be able to corrupt other processors’ data (ouch!)

Review: So how do we use them?

- First, figure out where there is parallel work in an application
  - Main topic of the next two lectures
- Next, choose a programming model
  - **Pthreads**: Low-level threading *library*
    - Uses fork-join model, like processes
    - Allows arbitrary code division



  - **OpenMP**: *Compiler directives* for parallel programming
    - Uses "parallel region" model to simplify threads
    - Higher-level, "parallel for"
    - Is often much easier to use, but not as general



Review: Coordinating Access to Shared Data: Locks

- We must be able to *control* access to *shared* memory
  - Unpredictable results can happen if we don't (Ex. `x++`)

CPU 1	CPU 2
ld r1, x	. . .
add r1, r1, 1	ld r1, x
—	add r1, r1, 1
—	st r1, x
st r1, x	. . .

CPU 1	CPU 2
LOCK X	. . .
ld r1, x	LOCK X
add r1, r1, 1	stall
st r1, x	stall
UNLOCK X →	unstall
	ld r1, x
	etc.

- Locks are a simple primitive to assert control
  - Put lock/unlock (acquire/release) pair *around* each critical region
  - Basis of all more complex variable control & synchronization
    - Semaphores, monitors, condition variables

Review: Locks: Performance vs. Correctness

- Few locks
  - Coarse grain locking
  - Easy to write parallel program
  - Processors spend a lot of time stalled waiting to acquire locks
  - Poor performance
- Many locks
  - Fine grain locking
  - Difficult to write parallel program
  - Higher chance of incorrect program (deadlock)
  - Good performance
- Make parallel programming difficult
  - How do you know what level of lock granularity to use?
  - Will discuss further in upcoming lectures . . . .

Review: Coordinating Access to Shared Data: Synchronization

- We often want to control *sequencing* of *parts* of threads:
  - To impose a sequential order on a code block
    - When a few lines just can't be parallelized
  - To wake up stalled threads
    - When stalled at a lock, for example
  - To control producer-consumer access to data
    - Producer signals consumer when output is ready
    - Consumer signals producer when it needs more input

  - To globally get all processors to the same point in the program
    - Divides a program into easily-understood *phases*
    - Generally called a **barrier**

Pthreads Synchronization: Condition Variables

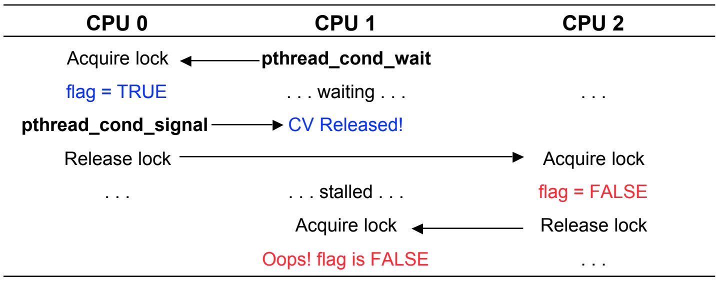
- Pthreads offers a lower-level interface to synchronization: *Condition Variables*
  - Provide simple “can I go?” and “go now” signaling calls
    - Should be thought of as “go if X” and “X has changed”
  - Can be used to build:
    - Barriers
    - Producer-consumer queues
    - Read-write locks
    - And just about any other communication primitive . . . .
- Is tied implicitly to a single lock & flag variable
  - Lock protects the condition variable during use
  - Flag allows condition to be tested independently

CV API

- `pthread_cond_wait(CV, lock)` to say “can I go?”
  - Always use *inside* the associated lock
  - Always use in a `while` loop that tests the flag variable
- `pthread_cond_signal(CV)` to say “next CPU go!”
  - Always use within the lock (*writing* to CV!)
  - Always *set the flag variable* before leaving the lock
- `pthread_cond_broadcast(CV)` to say “all CPUs go!”
  - Same restrictions as above
  - Useful for building barriers, but . . .
  - Still a delay after broadcast due to readers getting lock
    - All broadcast receivers must serialize on the lock acquisition
    - Could be lengthy if a lot of receivers
    - May want to consider a single-writer model in this case

Condition Variables in Action

```
while(!flag)
 pthread_cond_wait(&my_cv, &my_cv_lock);
```



CPU 0		CPU 1		CPU 2
Acquire lock	←	**pthread_cond_wait**		
flag = TRUE		. . . waiting . . .		. . .
pthread_cond_signal	→	CV Released!		
Release lock	→		→	Acquire lock
. . .		. . . stalled . . .		flag = FALSE
		Acquire lock	←	Release lock
		Oops! flag is FALSE		. . .

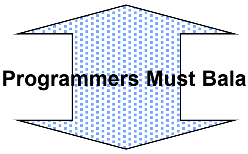
- `pthread_cond_timedwait(CV, lock, time)` limits waits
  - Allows you to do something else after awhile

Summary & A Look Ahead

- Three main portions of parallel programming models
  - Threads to divide up work
  - Locks to protect shared data
  - Synchronization primitives for sequencing/scheduling threads
- These constructs are the basis of shared memory programming
  - All programming assignments will build upon this
  - Some assignments will have you examine details
- We will see how these concepts get used in full applications
  - Dividing up applications into threads
  - Dividing up data to minimize communication and synchronization
  - Avoiding common bugs

The Two Sides of Parallelization

- **Dividing Work:** Need to chop computation into parallel tasks
  - What are smallest independent units in a program?
  - How must they be sequenced?



Parallel Programmers Must Balance These

- **Partitioning Data:** Localizing data onto processors
  - Required on message passing machines
  - Very helpful on shared-memory machines
  - Need to minimize expensive interprocessor *communication*

Today’s Outline: Tasks

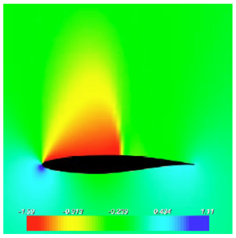
- Fixed task breakdown
  - Regular patterns
  - Graph patterns

- Dynamic task management
  - Unknown number of tasks
  - Unknown size of tasks
  - Task queues
  - Master-slave tasking

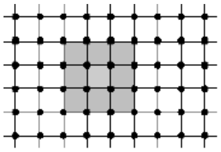
- Pipeline parallelism: *Intra*-task parallelism
  - Feedback loops within tasks
  - Stream parallelism

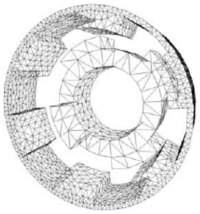
Static Task Decompositions

- Many applications decompose into tasks easily
  - Fixed-size tasks
  - Known number of tasks
  - Both are important!

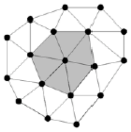


Regular Arrays



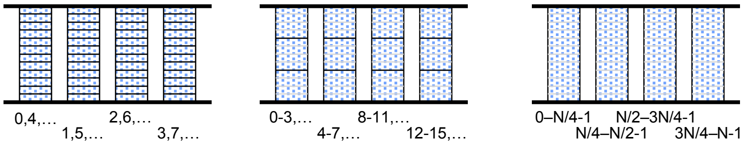


***Fixed* Irregular Data Structures**



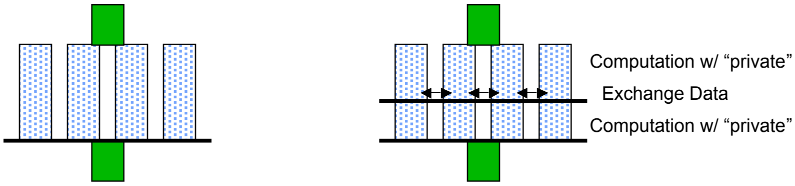
Dividing Up the Work

- Easy to allocate to processors
  - Fork off `n_procs` looping pthreads or use a `parallel for`
  - Allocate by:
    - Loop iteration (many tasks!)
    - Chunks of loop iterations (medium)
    - 1/`n_procs` iterations/processor (fewest)
  - Decide allocation based on algorithm and architecture
    - Does it have a “natural” chunk size?
    - Does it have a particular communication pattern btw. iterations?
    - How expensive is communication?



Static Task Synchronization

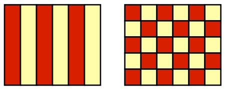
- Barriers are great!
  - Barriers at the end of each parallel region
    - Sync up before back-to-serial
  - Barriers in the middle of parallel regions for “phasing”
    - Sync up after each global data exchange
    - Can dramatically reduce the number of locks needed!
    - Create “private” data within each phase
  - Efficient because all processors execute ~same work



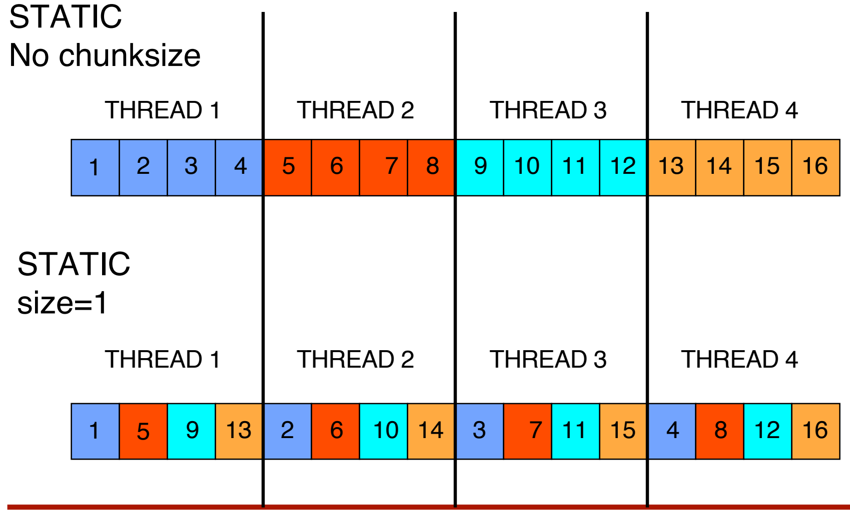
Static Partitioning with OpenMP & pthreads

- Do it manually with pthreads
  - Choose how to pass iterations to threads

- OpenMP offers simple options for loops
  - `schedule(static, size)` distributes *size* iterations/CPU
    - Simple and clear
    - Nesting works in some environments
      - Works under Solaris 10
      - Usually use entire rows/columns of multi-D arrays
    - Can get stuck if you (# *iterations*)/(*size•n_procs*) not an integer
      - Some “extra” processors during last batch of blocks
  - This covers most common cases



Static Partitioning Comparison



STATIC
No chunksize

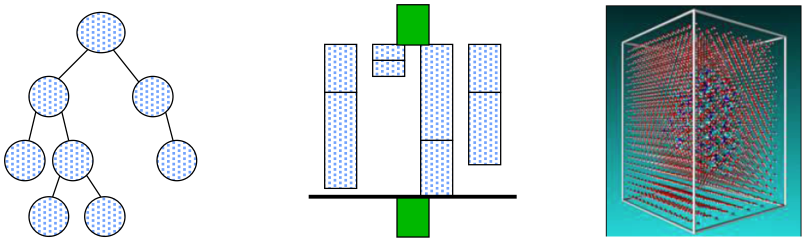
THREAD 1				THREAD 2				THREAD 3				THREAD 4			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

STATIC
size=1

THREAD 1				THREAD 2				THREAD 3				THREAD 4			
1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	16

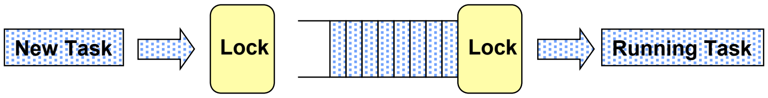
Problems with Static Partitioning

- Sometimes static task partitioning just won't work:
  - Unknown number of tasks
    - Dependent upon a complex data structure
    - Tasks generated dynamically, as we work
  - Unknown size of tasks
    - Data-dependent execution time
    - Need to balance among processors at runtime



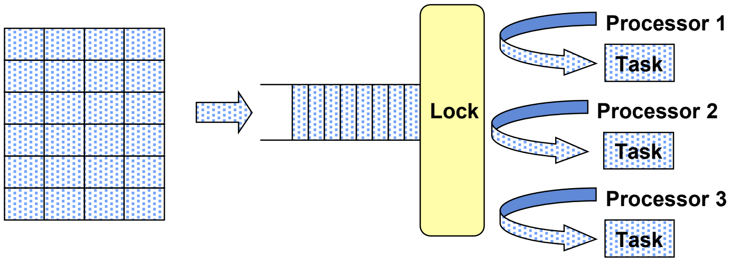
Solution: Dynamic Partitioning

- Use *real* threads (pthreads) for **large** parallel tasks
  - *Examples:* Entire database queries, web page lookups
  - Let the underlying thread system handle scheduling
    - Pthreads includes many routines to control scheduling
    - Saves you a lot of work
    - Allows *pre-emption* of long running tasks
- Use hand-built *task queues* for smaller parallel tasks
  - *Examples:* Tree nodes, blocks of pixels, etc.
  - Avoids often overly general thread schedule model
  - You can custom-build a queue to hold your tasks *efficiently*



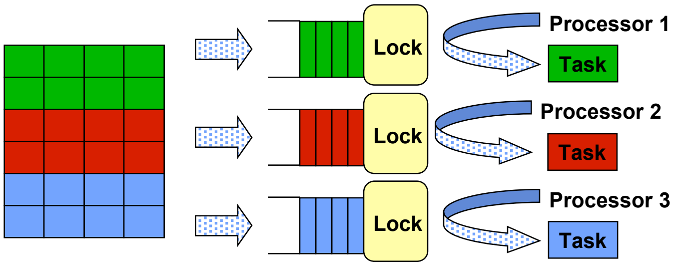
Kinds of Task Queues I: Global

- Global task queues: One per application
  - **Pro:** Excellent load balancing
  - **Con:** Can get "any" task . . . more communication!
  - **Con:** Contention for the lock protecting the queue
    - Not scalable beyond $T_{task}/T_{dequeue}$ processors



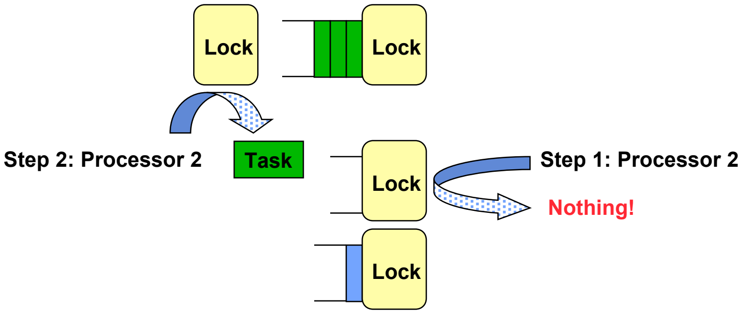
Kinds of Task Queues II: Distributed

- Could also have one queue per processor:
  - **Pro:** No lock contention, since it's private
  - **Pro:** Infinitely scalable
  - **Pro:** Can selectively put "related" items in the same queue
  - **Con:** Doesn't solve our load balancing problem!



Task "Stealing"

- **Solution:** Allow processors to borrow from other queues
  - Should only need to do occasionally
  - Can grab from the queue tail
    - Usually a different lock from the head, avoids contention



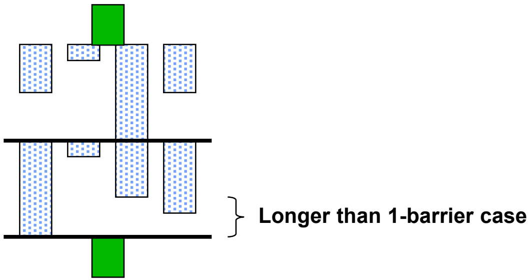
Other Queue Details

- Hybrid queues: *Subset* of processors sharing
  - Splits pros and cons between two basic models

- Dynamic task generation
  - Generate tasks as we compute
  - Common with large, graph-like structures of variable size
  - Must be careful how we *add* to distributed queues
    - Probably want to add to our own queue the most
      - Improves locality, reduces cache misses
    - But need to “fill in” short queues when ours is long
      - Algorithm for finding short queues needs to be scalable!

Things to Avoid

- Barriers!
  - Minimize, since they eliminate advantage of queues
  - Exacerbate load imbalance



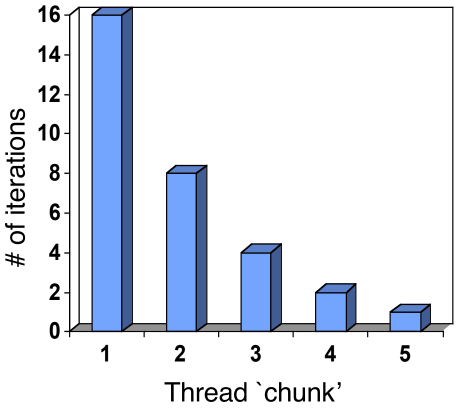
- Explicit inter-task sequencing
  - Order of tasks is hard to determine
  - Don’t make your program dependent upon it

Dynamic Tasking with OpenMP & pthreads

- Pthreads lets you make your own
  - Can easily customize to fit your application
  - Use locks and (optional) condition variables
  - Or just fork off new threads if tasks are large
    - Also, tasks should be safely pre-emptable

- OpenMP is a mixed bag
  - `schedule(dynamic, *size*)` is a dynamic equivalent to the `static` directive
    - Master passes off values of iterations to the workers of size size
    - Automatically handles dynamic tasking of simple loops
  - Otherwise must make your own
    - Includes many commonly used cases, unlike `static`
    - Just like pthreads, except *must be lock-only*

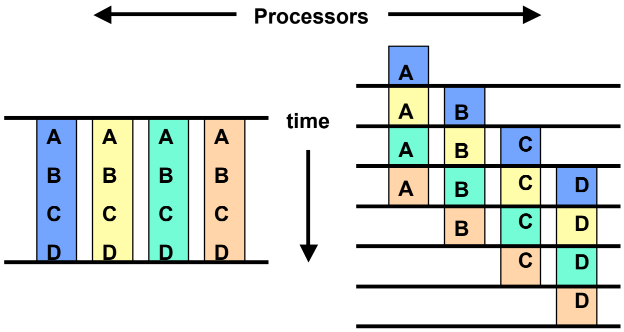
OpenMP Guided Scheduling



- `schedule(guided, *size*)`
- Guided scheduling is a compromise
- Iteration space is divided up into exponentially decreasing chunks
- Final `size` is usually 1, unless set by the programmer
- Chunks of work are dynamically obtained
- Works quite well provided work per iteration is constant – if unknown dynamic is better

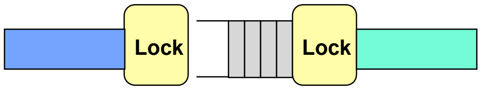
Pipeline Parallelism: Another Approach

- There are two common ways to parallelize:
  - Execute same task on different processors
    - Processor executes whole task on different data (data parallelism)
  - Pipeline task across processors
    - Processor executes a piece of a task (functional parallelism)

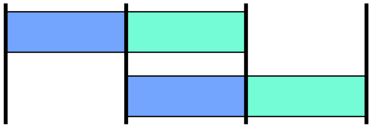


How to Pipeline

- You can pipeline in two ways
  - *Asynchronous:* Producer-consumer buffers/queues
    - More overhead, but localized — generally best



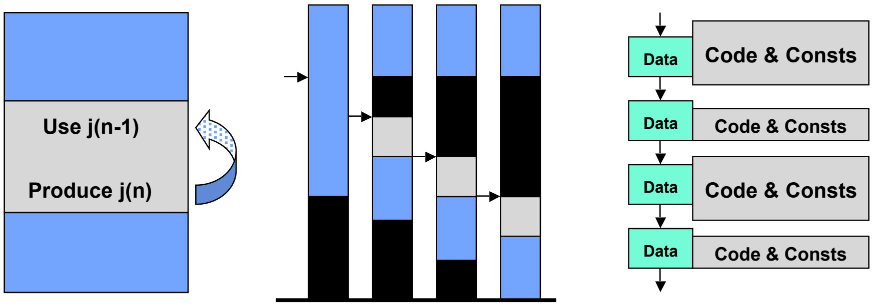
  - *Synchronous:* Barriers between pipe stages
    - Usually less overhead/sync, but global — use seldom



- Need to be careful to keep pipe stages ~same length

When to Use Pipelining

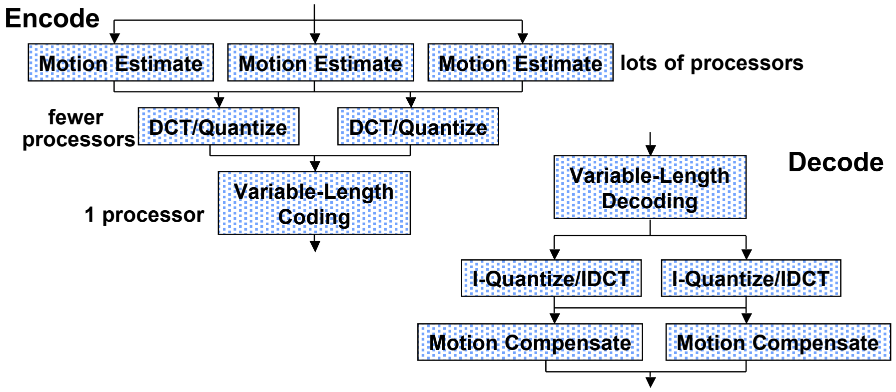
- Loop carried dependencies are the main reason:



- Can also sometimes be more natural with streaming data
  - Data arrives over time
  - As new data arrives, old moves on through
  - Code, constants & loop carried information doesn't move!

MPEG: A Good Pipeline Example

- Most stages have obvious task parallelism . . .
- . . . But final/first stage is hard-to-parallelize stream compression



Pipeline Parallelism: Example

```
#pragma omp parallel for private(i) num_threads(4) \
                 ordered
for (i = 0; i < n; i++) {
   a[i] = foo(i);
   b[i] = bar(i);
   if (b[i] < 0.0) b[i] = 0.0;
#pragma omp ordered
   c[i] = c[i] + c[i-1]
}
```

- Poor performance

Pipeline Parallelism: Example Improved

```
#pragma omp parallel sections private(i) num_threads(4)
#pragma parallel section
for (i = 0; i < n; i++)
   a[i] = foo(i);
#pragma parallel section
for (i = 0; i < n; i++){
   b[i] = bar(i);
   b_flag[i] = TRUE;
}
#pragma parallel section
for (i = 0; i < n; i++)
   while(b_flag[i]);
   if (b[i] < 0.0) b[i] = 0.0;
#pragma parallel section
for (i = 0; i < n; i++)
        c[i] = c[i] + c[i-1];
```

- Better performance

Summary & A Look Ahead

- Often have “regular” tasks
  - Schedule statically among processors
- But must often deal with “irregular” tasks
  - Use work queues to dynamically schedule
- Use pipelining to avoid serialization or for “streams” of data
- Will next examine how data affects parallelism
  - Ways to divide regular arrays
  - How data in trees affects tasking
  - Minimizing communication

Appendix 1

```
/**************************************************************************
 *                   SAMPLE DYNAMIC TASK QUEUEING CODE
 *
 *                        by Lance Hammond, 4/4/05
 *
 *  A brief demonstration of dynamic tasking with both pthreads and OpenMP.
 *  This just divides up a numbered series of tasks into a set of even-sized
 *  "queues" of tasks, while allowing the processors to "steal" from the
 *  other queues as necessary to allow dynamic load balancing.  More complex
 *  "queues" can be constructed to contain more complex sets of tasks, but
 *  the basic idea should stay the same.
 *
 *  This is done with pthreads.  The equivalent version with OpenMP is much
 *  simpler, just requiring the use of the "schedule(dynamic [, chunk_size])"
 *  flag on the #pragma statement.
 **************************************************************************/
```

Appendix 2

```
#include <pthread.h>

/* System Constants */
#define NUM_PROCESSORS             8        /* Number of processors to use */
#define CACHE_LINE_SIZE            64       /* Size of system's cache lines, in
   bytes */
#define NUM_TASKS                  10000    /* Number of tasks to divide up */
#define CHUNK_SIZE                 5        /* Number of tasks to pull from queue
   at once */
#define STEP_SIZE                  NUM_TASKS/NUM_PROCESSORS

/* Global type and variable definitions */

typedef struct ProcQueueStruct
{
   int procID;                     /* Processor ID # */
   int nextItem;                   /* Next item # on the queue */
   int endOfQueue;                 /* One past end of this processor's queue */
   int emptyFlag;                  /* This queue has emptied */
   pthread_mutex_t queueLock;      /* Lock to protect this queue */
   char padding[CACHE_LINE_SIZE];  /* Anti-false sharing padding */
} ProcQueue;
ProcQueue procQueues[NUM_PROCESSORS];
```

Appendix 3

```
/* work_from_queue function
 *
 * This pulls chunks of items off of the work queue and processes them until
 * the given work queue is emptied.  */

void work_from_queue(ProcQueue *q)
{
   int currentItem, lastItem, i;

   /* Take initial set of tasks off of the queue */
   pthread_mutex_lock(&(q->queueLock));
   currentItem = q->nextItem;
   q->nextItem += CHUNK_SIZE;
   lastItem = q->nextItem;
   pthread_mutex_unlock(&(q->queueLock));

   /* Eat through tasks until queue emptied */
   while (currentItem < q->endOfQueue)
   {
         /* Do a chunk of my own work */
         for (i=currentItem; (i < lastItem) && (i < q->endOfQueue); i++)
         {
                  /* Do something useful here with item "i" */
         }
```

Appendix 4

```
          /* Get another chunk of work */
          pthread_mutex_lock(&(q->queueLock));
          currentItem = q->nextItem;
          q->nextItem += CHUNK_SIZE;
          lastItem = q->nextItem;
          pthread_mutex_unlock(&(q->queueLock));
    }

    /* Set the "empty" flag for this queue */
    pthread_mutex_lock(&(q->queueLock));
    q->emptyFlag = 1;
    pthread_mutex_unlock(&(q->queueLock));
}

/* worker_function function
 *
 * This works on tasks from this processor's work queue, and then
 * steals from others, completing only after it has verified that *all*
 * work queues are completely empty.  For simplicity, each processor only
 * steals from one other queue at a time, instead of trying to steal in
 * more "fair" fashion across processors.  */
```

Appendix 5

```
void *worker_function(void *input)
{
    ProcQueue *myQueue;
    int i;

    myQueue = (ProcQueue*) input;

    /* Use my own queue to do work, initially */
    work_from_queue(myQueue);

    /* Done with my work, loop through other queues and try to steal */
    /* NOTE: This algorithm is very simple, and could be improved */
    for (i = (myQueue->procID + 1) % NUM_PROCESSORS;
          i != myQueue->procID;  i = (i + 1) % NUM_PROCESSORS)
    {
          /* Work on this queue if not empty (no lock, OK if we misread since
           * it's just a performance optimization and will work anyway) */
          if (procQueues[i].emptyFlag != 1) work_from_queue(&(procQueues[i]));
    }
    /* Now done -- we've checked all other queues for work */
}
```

Appendix 6

```
/* main function
 * This initializes the work queues and then forks/joins the worker threads.
 * This program uses the "master thread sleeps during parallel region" model. */
void main(){
    int i, start;
    pthread_t myThreads[NUM_PROCESSORS];
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* Set up the thread queues */
    for (i=0, start=0; i < NUM_PROCESSORS; i++){
          procQueues[i].procID = i;
          procQueues[i].nextItem = start;
          start += STEP_SIZE;
          if (i == NUM_PROCESSORS - 1) start = NUM_TASKS;
          procQueues[i].endOfQueue = start;
          procQueues[i].emptyFlag = 0;
          pthread_mutex_init(&(procQueues[i].queueLock), NULL);
    }
    /* And start/join the parallel threads */
    for (i=0; i < NUM_PROCESSORS; i++)
          pthread_create(&myThreads[i], &attr, worker_function, (void*) &procQueues[i]);
    for (i=0; i < NUM_PROCESSORS; i++)
          pthread_join(myThreads[i], NULL);
}
```