CS4961 Parallel Programming

Lecture 9: Task Parallelism in OpenMP

Mary Hall
September 22, 2011

Administrative

- Homework 3 will be posted later today and due Thursday before class
  - We'll go over the assignment on Tuesday
- Preview of Homework (4 problems)
  - Determine dependences in a set of computations
  - Describe safety requirements of reordering transformations
  - Use loop transformations to improve locality of simple examples
  - Use a collection of loop transformations to improve locality of a complex example
- Experiment: Break at halfway point

Today's Lecture

- Go over programming assignment
- A few OpenMP constructs we haven't discussed
- Discussion of Task Parallelism in Open MP 2.x and 3.0
- Sources for Lecture:
  - Textbook, Chapter 5.8
  - OpenMP Tutorial by Ruud van der Pas

    http://openmp.org/mp-documents/ntu-vanderpas.pdf
  - Recent OpenMP Tutorial

    http://www.openmp.org/mp-documents/omp-hands-on-SC08.pdf
  - OpenMP 3.0 specification (May 2008):

    http://www.openmp.org/mp-documents/spec30.pdf

OpenMP Data Parallel Summary

- Work sharing
  - parallel, parallel for, TBD
  - scheduling directives: static(CHUNK), dynamic(), guided()
- Data sharing
  - shared, private, reduction
- Environment variables
  - **OMP_NUM_THREADS, OMP_SET_DYNAMIC, OMP_NESTED, OMP_SCHEDULE**
- Library
  - E.g., omp_get_num_threads(), omp_get_thread_num()

Conditional Parallelization

if (scalar expression)

- ✓ ***Only execute in parallel if expression evaluates to true***
- ✓ ***Otherwise, execute serially***

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)   {
#pragma omp for
for (i=0; i<n; i++)
    x[i] += y[i];
} /*-- End of parallel region --*/
```

Review:

- parallel
- for
- private
- shared

SINGLE and MASTER constructs

- Only one thread in team executes code enclosed
- Useful for things like I/O or initialization
- No implicit barrier on entry or exit

```
#pragma omp single {
    <code-block>
}
```

- Similarly, only master executes code

```
#pragma omp master {
    <code-block>
}
```

Also, more control on synchronization

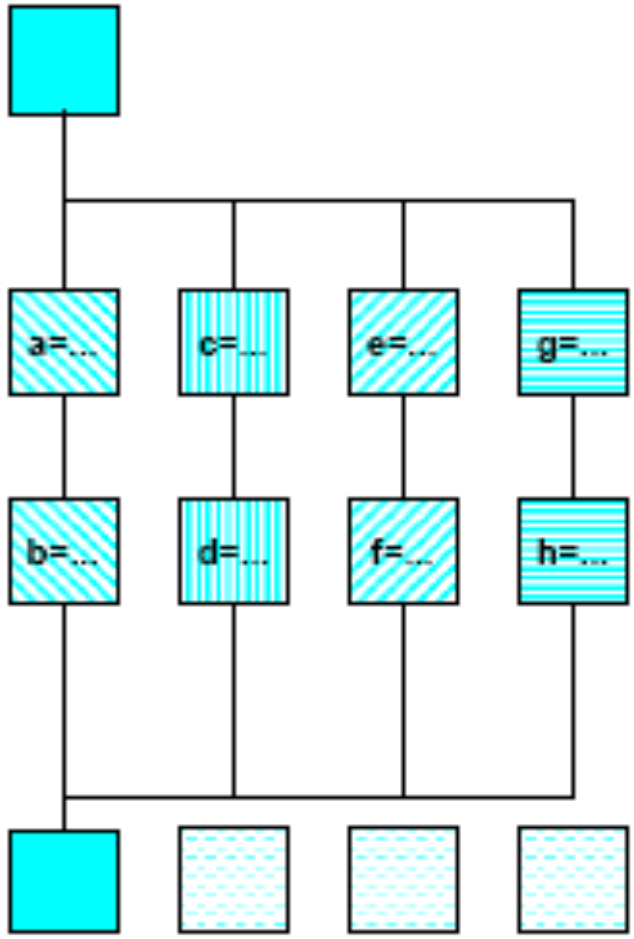
```
#pragma omp parallel shared(A,B,C)
{
   tid = omp_get_thread_num();
   A[tid] = big_calc1(tid);
  #pragma omp barrier
  #pragma omp for
  for (i=0; i<N; i++) C[i] = big_calc2(tid);
  #pragma omp for nowait
   for (i=0; i<N; i++) B[i] = big_calc3(tid);
  A[tid] = big_calc4(tid);
}
```

General: Task Parallelism

- Recall definition:
  - A task parallel computation is one in which parallelism is applied by performing distinct computations – or tasks – at the same time. Since the number of tasks is fixed, the parallelism is not scalable.
- OpenMP support for task parallelism
  - Parallel sections: different threads execute different code
  - Tasks (NEW): tasks are created and executed at separate times
- Common use of task parallelism = Producer/consumer
  - A producer creates work that is then processed by the consumer
  - You can think of this as a form of pipelining, like in an assembly line
  - The "producer" writes data to a FIFO queue (or similar) to be accessed by the "consumer"

Simple: OpenMP sections directive

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



Parallel Sections, Example

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
  #pragma omp sections nowait    {
    #pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
    #pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
  } /*-- End of sections --*/
} /*-- End of parallel region
```

Tasks in OpenMP 3.0

- A task has
  - Code to execute
  - A data environment (shared, private, reduction)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task
  - Some thread in the team executes the thread at a later time

Simple Producer-Consumer Example

```
// PRODUCER: initialize A with random data
void fill_rand(int nval, double *A) {
for (i=0; i<nval; i++) A[i] = (double) rand()/1111111111;
}

// CONSUMER: Sum the data in A
double Sum_array(int nval, double *A) {
double sum = 0.0;
for (i=0; i<nval; i++)  sum = sum + A[i];
return sum;
}
```

Key Issues in Producer-Consumer Parallelism

- Producer needs to tell consumer that the data is ready
- Consumer needs to wait until data is ready
- Producer and consumer need a way to communicate data
  - output of producer is input to consumer
- Producer and consumer often communicate through First-in-first-out (FIFO) queue

One Solution to Read/Write a FIFO

- The FIFO is in global memory and is shared between the parallel threads
- How do you make sure the data is updated?
- Need a construct to guarantee ***consistent*** view of memory
  - Flush: make sure data is written all the way back to global memory

Example:

```
Double A;
A = compute();
Flush(A);
```

Solution to Producer/Consumer

```
flag = 0;
#pragma omp parallel
{
   #pragma omp section
   {
      fillrand(N,A);
      #pragma omp flush
      flag = 1;
      #pragma omp flush(flag)
   }

   #pragma omp section
   {
       while (!flag)
          #pragma omp flush(flag)
      #pragma omp flush
      sum = sum_array(N,A);
   }
}
```

Another Example from Textbook

- Implement Message-Passing on a Shared-Memory System
- A FIFO queue holds messages
- A thread has explicit functions to Send and Receive
  - Send a message by enqueuing on a queue in shared memory
  - Receive a message by grabbing from queue
  - Ensure safe access

Message-Passing

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

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

Sending Messages

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

Use synchronization mechanisms to update FIFO
"Flush" happens implicitly
What is the implementation of Enqueue?

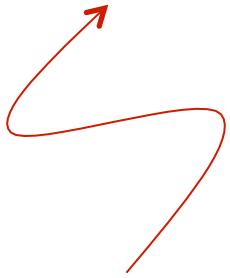
Receiving Messages

```
      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);
```

This thread is the only one to dequeue its messages.
Other threads may only add more messages.
Messages added to end and removed from front.
Therefore, only if we are on the last entry is synchronization needed.

Termination Detection

```
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

More synchronization needed on “done_sending”

Task Example: Linked List Traversal

```
.....
while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
.....
```

- How to express with parallel for?
  - Must have fixed number of iterations
  - Loop-invariant loop condition and no early exits

OpenMP 3.0: Tasks!

```
my_pointer = listhead;
#pragma omp parallel    {
  #pragma omp single nowait      {
     while(my_pointer) {
       #pragma omp task firstprivate(my_pointer)          {
         (void) do_independent_work (my_pointer);
        }
        my_pointer = my_pointer->next ;
     }
  } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier here
```

firstprivate = private and copy initial value from global variable
lastprivate = private and copy back final value to global variable

Summary

- Completed coverage of OpenMP
  - Locks
  - Conditional execution
  - Single/Master
  - Task parallelism
    - Pre-3.0: parallel sections
    - OpenMP 3.0: tasks
- Coming soon:
  - OpenMP programming assignment, mixed task and data parallel computation