CS4230 Parallel Programming

Lecture 12: More Task Parallelism

Mary Hall
October 4, 2012

Homework 3: Due Before Class, Thurs. Oct. 18

handin cs4230 hw3 <file>

- Problem 1 (Amdahl's Law):
  (i) Assuming a 10 percent overhead for a parallel computation, compute the speedup of applying 100 processors to it, assuming that the overhead remains constant as processors are added. (ii) Given this speedup, what is the efficiency? (iii.) Using this efficiency, suppose each processor is capable of 10 Gflops peak performance. What is the best performance we can expect for this computation in Gflops?
- Problem 2 (Data Dependences):
  Does the following code have a loop-carried dependence? If so, identify the dependence or dependences and which loop carries it/them.

```
for (i=1; i < n-1; i++)
  for (j=1; j < n-1; j++)
    A[i][j] = A[i][j-1] + A[i-1][j+1];
```


Homework 3, cont.

Problem 3 (Locality):

Sketch out how to rewrite the following code to improve its cache locality and locality in registers. Assume row-major access order.

```
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    a[j][i] = a[j-1][i-1] + c[j];
```

Briefly explain how your modifications have improved memory access behavior of the computation.

Homework 3, cont.

Problem 4 (Task Parallelism):

Construct a producer-consumer pipelined code in OpenMP to identify the set of prime numbers in the sequence of integers from 1 to n. A common sequential solution to this problem is the sieve of Erasthones. In this method, a series of all integers is generated starting from 2. The first number, 2, is prime and kept. All multiples of 2 are deleted because they cannot be prime. This process is repeated with each remaining number, up until but not beyond sqrt(n). A possible sequential implementation of this solution is as follows:

```
for (i=2; i<=n; i++) prime[i] = true; // initialize
for (i=2; i<= sqrt(n); i++) {
  if (prime[i]) {
    for (j=i+i; j<=n; j = j+i)  prime[j] = false;
  }
}
```

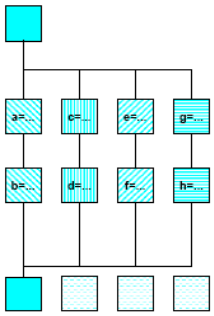
The parallel code can operate on different values of i. First, a series of consecutive numbers is generated that feeds into the first pipeline stage. This stage eliminates all multiples of 2 and passes remaining numbers onto the second stage, which eliminates all multiples of 3, etc. The parallel code terminates when the "terminator" element arrives at each pipeline stage.

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
```


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);
   }
}
```


Reminder: What is Flush?

- Flush
  - Specifies that all threads have the same view of memory for all shared objects.
- Flush(var)
  - Only flushes the shared variable "var"

Another (trivial) producer-consumer example

```
for (j=0; j<M; j++) {
  sum[j] = 0;
  for(i = 0; i < size; i++) {
    // TASK 1: scale result
    out[i] = _iplist[j][i]*(2+i*j);
    // TASK 2: compute sum
    sum[j] += out[i];
  }
  // TASK 3: compute average and compare with max
  res = sum[j] / size;
  if (res > maxavg) maxavg = res;
  }
  return maxavg;
```


Another Example from Textbook

- Implement Message-Passing on a Shared-Memory System for Producer-consumer
- 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