Using the Cray XMT™ `for all streams` Pragmas

Abstract

This document describes the `for all streams` compiler directives and how to use them to execute a block of code on multiple streams.

RECORD OF REVISION

S–0038–14 Published October 2010 Supports 1.4 and later releases running on the Cray XMT hardware.

Using the Cray XMT `for all streams` Pragmas

Overview

In some programming situations it is useful to specify that a block of code should execute exactly once on each stream of a parallel region, allowing the application to manage data on a per-thread basis. Effective with the 1.4 release two pragma compiler directives were added that support this.

Description

The syntax of the `for all streams` pragmas is as follows:

`#pragma mta for all streams`

This directive starts up a parallel region (if the code is not already in a parallel region) and cause the next statement or block of statements to be executed exactly once on every stream allocated to the region. If the pragmas appear in code that would otherwise not be parallel, they cause it to go parallel. For example,

```
#pragma mta for all streams
   printf("Stream checking in\n");
```

would cause every stream to print the phrase "Stream checking in" once.

In this example the pragma executes a block of code that increments a counter before printing the phrase:

```
  int counter = 0;
#pragma mta for all streams
   {
   counter++;
   printf("%d streams checked in \n", counter)
   };
```

`#pragma mta for all streams` *i* `of` *n*

This directive is similar to the `for all streams` pragma except that it also sets the variable *n* to the total number of streams executing the region, and the variable *i* to a unique per-stream identifier between `0` and *n*`-1`. For example:

```
int i, n;
int check_in_array[MAX_PROCESSORS * MAX_STREAMS_PER_PROCESSOR];
for (int i = 0; i < MAX_PROCESSORS * MAX_STREAMS_PER_PROCESSOR; i++)
  check_in_array[i] = 0;


#pragma mta for all streams i of n
   {
   check_in_array[i] = 1;
   printf("Stream %d of %d checked in.\n", i, n);
   }
```

Note that the integer variables *i* and *n* must be declared separately from the pragma.

You can use the `for all streams` pragmas in conjunction with the `use` *n* `streams` pragma to ask the compiler to allocate a certain number of streams per processor to the parallel region executing the `for all streams` block.

```
#pragma mta use 100 streams
#pragma mta for all streams
   {// do something
   }
```

Be aware, however, that there is no guarantee that the runtime will grant the requested number of streams. For example, sufficient streams may not be available due to other jobs, the OS, or other simultaneous parallel regions in the current job.

Examples

In the following example, taken from a breadth-first search procedure, the `for all streams` pragma is used to divide a data structure between threads.

```
int processQueue(int *Q,unsigned &head, unsigned &tail, unsigned qcap,
   const Neighbor neighbors[],
   const int numNeighbors[], sync int *Marked)
{
#pragma mta trace "process"
#pragma mta noalias *Q, *Marked, *neighbors, *numNeighbors
  // elements [head,tail) are readonly
  // we can write to other elements of Q
  const unsigned oldtail = tail;
  const unsigned oldhead = head;
  unsigned newhead = head;
  unsigned stubbed = 0;

#pragma mta use 100 streams
#pragma mta for all streams
  {
    unsigned outhead = 0, outtail = 0;
    for(;;) {
      // grab INBLOCK nodes (& stubs) from the input
      unsigned inhead = int_fetch_add(&newhead, INBLOCK);
      // avoid overrun
      unsigned intail = std::min(inhead + INBLOCK, oldtail);

      if (inhead>=intail) break;   // stop if we ran out of work

#pragma mta assert nodep *Q,*numNeighbors,*neighbors
   for(int i=inhead; i<intail; i++) {
   int u = Q[i%qcap];                                  // |N|
   if (u>=0) {
      int begin = numNeighbors[u];                      // |N|
      int end = numNeighbors[u+1];                      // |N|

#pragma mta assert nodep *Q, *neighbors, *Marked
     for(int j=begin;j<end;j++) {
       int v = neighbors[j];                           // |E|
       int mark = Marked[v];                 // lock - |N|
       if (mark<0) {
         Marked[v] = u;                      // unlock and mark - |N|
```

```
         // we have set v's parent to u
         // reserve space for enqueueing OUTBLOCK nodes
         if (outhead>=outtail) {
       outhead = int_fetch_add(&tail, OUTBLOCK);
       outtail = outhead+OUTBLOCK;
         }
         Q[(outhead++)%qcap] = v;           // |N|
        }else {
        Marked[v] = mark;                  // unlock & keep mark
        }
       }
     }
     }
    }
#ifdef PHASES
    stubbed += outtail-outhead;
#endif
    // stub-out the rest of reserved space
    // <streams*levels*outblock memops
    for(;outhead<outtail; outhead++) Q[outhead] = -1;
  }
  head = oldtail;
  return tail - oldtail - stubbed;
}
```

In this code, each stream executes the block of code following the `for all streams` pragma exactly one time. The code maintains a per-stream pointer to an output block of the queue (`outhead`) which is updated by the line:

```
outhead = int_fetch_add(&tail, OUTBLOCK);
```

whenever the block runs out of room. A per-stream pointer to an input block of the queue (`inhead`) is updated by the line

```
unsigned inhead = int_fetch_add(&newhead, INBLOCK);
```

after the previous block has been processed.

For more information on parallelization and on using the pragma compiler directives, see *Cray XMT Performance Tools User's Guide*.