Limiting Loop Parallelism in Cray XMT™ Applications

June 21, 2010

Abstract

Two new pragmas were added to the XMT C/C++ compiler that enable users to limit the amount of concurrency and/or the max number of processors used by a parallel loop. The max processors pragma can be used to limit the number of processors used by a multiprocessor loop. The max concurrency pragma can be used to limit either the total number of streams used by a single or multiprocessor loop, or to limit the number of futures created by a loop that uses loop future parallelism.

Table of Contents

Introduction

Users writing applications for the Cray XMT would like to be able to limit the amount of parallelism used by a loop for multiple reasons, such as to prevent contention on resources or to improve load balancing across multiple concurrently running threads in applications that use nested parallelism. Currently, one of the easiest ways the user can limit the parallelism of a loop is by switching the parallel modes, such as by explicitly using single processor parallelism instead of multiprocessor parallelism. However, this can be very limiting and the user may want to be able to easily use multiple processors without having to use all processors.

As mentioned above, a user may want to limit the amount of parallelism used by a parallel loop to prevent contention on resources. In some cases, a loop may cause hotspotting on shared resources, so a user may want to limit the number of processors used by the loop to prevent or reduce the contention. Using the pragmas to prevent or limit contention could become more important as the machine size scales up.

Another situation where the user may want to limit the amount of parallelism used by a loop could be in cases of nested parallelism. Applications on the XMT would like to use nested parallelism in order to have multiple parallel loops running concurrently. This is done to improve the performance and scalability of the application by doing the following:

- limiting the amount of resources used by one parallel loop to prevent it from hotspotting the system
- improving load balance by distributing resources across multiple parallel loops
- increasing utilization by having multiple parallel loops running concurrently

Over the years of the MTA and XMT, users have struggled to use nested parallelism efficiently. The cost of nested parallelism is very high, and the users currently have no way to control the distribution of resources, which can easily lead to poor performance due to load imbalances. Users have often been forced to either not use nested parallelism, or to modify their applications to make it possible for the compiler to perform a loop collapse on the loops.

One way that users have tried to use nested parallelism is by creating a small number of threads with explicit futures and within each future, executing one or more parallel loops. The user tries to start these futures at approximately the same time to try to ensure that the hardware resources are distributed across the running loops.

The problem with this approach is that there is no way to ensure that the futures start at the same time, and even if they do, there is no easy way to limit the number of streams or processors the runtime could assign to executing a parallel loop. It is possible that one parallel loop could get a large number of the available streams, which could cause the remaining parallel loops to have to wait for streams to become available or possibly be starved for resources and execute a loop with a small number of streams. For example, if each of the running threads created with the futures has a loop that uses multiprocessor parallelism, one thread could request all available streams before the others requested any, which could cause the remaining threads to wait. Another possibility would be that a thread gets only a small

number of the streams it requests because one or more of the other threads acquired the majority of the streams. This could greatly impact the performance of the thread that only got a small number of streams.

To make it easier for the user to limit the amount of resources used by a loop to help prevent hotspotting and to help the user better manage nested parallelism in their application, two new pragmas were added to the XMT C/C++ compiler to allow the user to limit the max number of processors and/or the max concurrency used by a parallel loop. The **max *n* processors** pragma can be used to limit the number of processors used by a multiprocessor parallel loop. The **max concurrency *c*** pragma can be used to limit either the total number of streams used by a single or multiprocessor parallel loop, or to limit the number of futures created for a loop that uses loop future parallelism.

The new **max *n* processors** and **max concurrency *c*** pragmas should not be confused with the **use *n* streams** pragma. The **use *n* streams** pragma can be used to specify a minimum number of streams to request per processor, but the pragma does not guarantee that the loop will get ***n*** streams. For example, the runtime may not be able to grant the requested number of streams if resources are not available. Also, the compiler could request more streams for the loop than what is specified by the **use *n* streams** pragma. However, the new **max *n* processors** and **max concurrency *c*** pragmas specify limits on the number of processors to use, the total number of streams to use, or the number of futures to create. For the case of the **max *n* processors** pragma, the runtime will guarantee that no more than ***n*** processors are used by the multiprocessor loop. For a multiprocessor loop with a **max concurrency *c*** pragma, the runtime will ensure that the number of processors used is the minimum needed to have enough streams to meet the specified concurrency limit of ***c***. For a single processor parallel loop with a **max concurrency *c*** pragma, the runtime will ensure that no more than ***c*** streams are used. Finally, for a loop futures parallel loop with a **max concurrency *c*** pragma, the runtime will ensure that no more than ***c*** futures are created.

Max Processors Pragma

The **max *n* processors** pragma can be used to limit the number of processors used by a multiprocessor parallel loop. This can be useful in applications that have multiple parallel loops running concurrently that want to limit the parallelism for each loop to help improve load balancing and prevent starvation. It can also be useful in applications that wish to limit the number of processors used by a loop to reduce or prevent hotspotting.

Syntax

```
#pragma mta max n processors
```

Usage and Limitations

- Loop level directive so the pragma can only be applied to a loop
- Limits the number of processors used by a multiprocessor loop to *n*
- *n* must be a compile-time unsigned integer constant greater than 0
- This pragma can only be used on a multiprocessor parallel loop
- If multiple **max *n* processors** pragmas are specified on one loop, the value of *n* specified by the last pragma will be used
- For collapsible loop nests, the max processors value specified by the outer loop (if any) will be used for the collapsed loop

Example

The following is an example of using the **max *n* processors** pragma on a multiprocessor parallel loop.

```
/* Use at most 4 processors with default number of streams per
proc. */
#pragma mta max 4 processors
  for(i = 0; i < size; i++) {
    array[i] += array[i] + (size + i);
  }
```

Max Concurrency Pragma

The **max concurrency *c*** pragma can be used to limit the max concurrency of any parallel loop. This pragma will limit the total number of streams used by either a single or multiprocessor parallel loop and will limit the number of futures created by a loop that uses loop future parallelism. This pragma is similar to the **max *n* processors** pragma in that it can be useful in applications that have multiple parallel loops running concurrently that want to better manage the nested parallelism, and in applications that want to limit resources used by a loop to help reduce or prevent contention.

Syntax

```
#pragma mta max concurrency c
```

Usage and Limitations

- Loop level directive so the pragma can only be applied to a loop.
- Limits the number of streams used by a single processor parallel loop to the *min(c, <num_streams_per_processor>)*, where *<num_streams_per_processor>* is the number of streams the compiler requests.

- Limits the number of processors used by a multiprocessor parallel loop to *max(1, c / <num_streams_per_processor>)*, where *<num_streams_per_processor>* is the number of streams the compiler requests for each processor used by the parallel loop.
  - If c is larger than or equal to *<num_streams_per_processor>*, the total number of streams used by the parallel loop will be at most c.
  - If c is less than *<num_streams_per_processor>*, one processor will be used and *<num_streams_per_processor>* streams will be requested by the compiler.
- Limits the number of futures created for a loop that uses loop future parallelism to c.
- If multiple **max concurrency *c*** pragmas are specified on one loop, the value of c specified by the last pragma will be used.
- For collapsible loop nests, the max concurrency value specified by the outer loop (if any) will be used for the collapsed loop.
- The **max concurrency *c*** pragma is not allowed to be used on a loop that also uses the **use *n* streams** pragma.

Examples

The following example illustrates using the **max concurrency *c*** pragma on a single processor parallel loop.

```
   /* Use at most 95 streams.  */
#pragma mta loop single processor
#pragma mta max concurrency 95
  for(i = 0; i < size; i++) {
         array[i] += array[i] + (size + i);
 }
```

The following example illustrates using the **max concurrency *c*** pragma on a multiprocessor parallel loop.

```
  /* Use at most 512 streams across all processors.  */
#pragma mta max concurrency 512
  for(i = 0; i < size; i++) {
    array[i] += array[i] + (size + i);
  }
```

The following example illustrates using the **max concurrency *c*** pragma on a loop that uses loop future parallelism.

```
  /* Create at most 512 futures.  */
#pragma mta loop future
#pragma mta max concurrency 512
  for(i = 0; i < size; i++) {
         array[i] += array[i] + (size + i);
 }
```

Multiprocessor parallel loops are allowed to use both the **max *n* processors** and **max concurrency *c*** pragmas, and can use both on a single loop. In cases where both pragmas are used, the lower bound of the number of processors estimated by the two limits will be the limit used on the loop. For example, the following code illustrates the use of both pragmas on one multiprocessor parallel loop.

```
  /* Use at most 512 streams across all processors or
   * at most 8 processors, whichever is smaller.
   */
#pragma mta max concurrency 512
#pragma mta max 8 processors
  for(i = 0; i < size; i++) {
         array[i] += array[i] + (size + i);
 }
```

In the above example, if the compiler were to request 64 streams per processor, then the **max concurrency 512** would estimate that 8 processors should be used for the loop (i.e., 512/64). The **max 8 processors** has the same limit on the number of processors so the loop would be limited to 8 processors. If the compiler instead requested 32 streams per processor, then the **max concurrency 512** would estimate that 16 processors should be used, which is more than the limit of 8 specified by the **max 8 processors**, so the loop would be limited to 8 processors. Because the **use *n* streams** pragma cannot be used on the same loop as a **max concurrency *c*** pragma, the loop will use the default number of streams determined by the compiler. The user will need to look at the canal details for a loop to determine the default number of streams being requested by the compiler.

Effect of Pragmas on Loop Fusion and Parallel Region Merging

The new pragmas can prevent the compiler from fusing loops if the loops involved do not have the same limits for the max processors and max concurrency. This is because the compiler will need to put the loops into different parallel regions in order to limit the processors and/or concurrency as requested by the user. This could potentially have a negative impact on the performance of a user's application, so users may need to look at the canal output to see what loops the compiler fused.

The pragmas could also prevent the compiler from merging the parallel regions for different loops into a single parallel region. The limitation for concurrency or processors specified by the new pragmas applies to the current parallel region that contains the loop with the pragmas. The compiler must ensure that all loops in a parallel region have the same limits for max processors and max concurrency. If the loops do not have matching limits, the compiler will put them in different parallel regions to ensure the user's limits on processors and/or concurrency can be correctly applied. This could potentially have a negative impact on the performance of a user's application because more time will be spent tearing down and starting new parallel regions. In the case of nested parallel regions, any limitations for concurrency or processors specified with the pragmas on either region do not affect the other region. For example, if the outer parallel region has a **max 8 processors**, that pragma will not affect the inner parallel region because the pragmas apply to the current parallel region only. The user can determine what loops the compiler placed in a parallel region by looking at the canal output. The "Additional Loop Details" shows which parallel region a loop is in, and the details for parallel regions state what limits for processors or concurrency (if any) are being applied to the region.

The following is an example of two loops that have matching limits for **max *n* processors** that could be fused and placed into one parallel region by the compiler.

```
#pragma mta max 64 processors
  for(i = 0; i < size; i++)
    array[i] = i;

#pragma mta max 64 processors
  for(i = 0; i < size; i++) {
    array[i] += array[i] + (size + i);
  }
```

The following is an example of two loops that cannot be fused or put into one parallel region because the loops specify different limits for the max processors.

```
#pragma mta max 256 processors
  for(i = 0; i < size; i++)
    array[i] = i;

#pragma mta max 512 processors
  for(i = 0; i < size; i++) {
    array[i] += array[i] + (size + i);
  }
```

The following is another example of two loops that cannot be fused or put into one parallel region because the loops specify different limits for the max processors. The first loop does not use the **max *n* processors** pragma, which implies there is no user specified limit.

```
  for(i = 0; i < size; i++)
    array[i] = i;
```

```
#pragma mta max 512 processors
  for(i = 0; i < size; i++) {
    array[i] += array[i] + (size + i);
  }
```

Use Case: Applying Max Processors Pragma to GraphCT

An example application that uses nested parallelism to improve system utilization and reduce contention on shared data structures is GraphCT (Graph Characterization Toolkit) [1]. GraphCT consists of multiple kernels that perform operations on a graph and the kernel focused on in this example is betweenness centrality.

The betweenness centrality kernel of GraphCT is executed concurrently by a small number of threads using loop future parallelism, and each thread uses multiprocessor parallelism to compute the betweenness centrality of a node. The betweenness centrality kernel of GraphCT can see significant variance in performance due to issues with load balancing across the threads. The **max *n* processors** pragma can be used to help improve load balancing and increase utilization by evenly distributing the processors across the threads.

The betweenness centrality kernel of GraphCT consists of two functions, `kcentrality` and `kcent_core`. The `kcentrality` function creates a small number of threads using loop future parallelism, and each of those threads calls `kcent_core` to compute the betweenness centrality for the nodes in the graph. Both of these functions were updated to make use of the new **max *n* processors** pragma.

The changes to `kcent_core` are limited to applying the **max *n* processors** pragma to each parallel loop in the function. The limit for the number of processors to use per thread was determined experimentally based on the default number of threads created in `kcentrality` in the release version 0.4 of GraphCT, which is 20. This would give each thread approximately 6 processors on a 128P XMT system if each thread got the same number of processors. This led to trying a limit of 8 processors per thread in `kcent_core`. Experiments showed that using 8 processors per thread performed better than the release version of GraphCT with 20 threads and no **max *n* processors** pragmas. A power of two was chosen so the number of processors in the system could be easily divided by the number of processors used per thread. A limit of 16 processors per thread was also tested and was shown to have reasonable performance that could be very similar to the performance with a limit of 8, especially for larger graphs (scale >= 28). The following code snippets show how the **max *n* processors** pragma was used for each loop in `kcent_core`. In these examples, `MAX_PROCS` is a preprocessor macro that has been defined as 8.

```
<...>
#pragma mta max MAX_PROCS processors
#pragma mta assert nodep
   for (j = 0; j < NV; j++) {marks[j] = sigma[NV*(K+1) + j] =
0;}
<...>
```

```
#pragma mta max MAX_PROCS processors
#pragma mta assert nodep
    for (j = 0; j < (K+1)*NV; j++) {
         dist[j] = -1;
         sigma[j] = child_count[j] = 0;
   }
<...>
#pragma mta max MAX_PROCS processors
#pragma mta assert no dependence
#pragma mta block dynamic schedule
#pragma mta use 100 streams
      for (j = Qstart; j < Qend; j++) {
<...>
#pragma mta max MAX_PROCS processors
#pragma mta assert nodep
#pragma mta assert no alias *sigma *Q *child *start *QHead
#pragma mta use 100 streams
            for (n = QHead[p]; n < QHead[p+1]; n++) {
<...>
#pragma mta max MAX_PROCS processors
      for (j=0; j<(K+1)*NV; j++) delta[j] = 0.0;
<...>
#pragma mta max MAX_PROCS processors
#pragma mta assert nodep
#pragma mta block dynamic schedule
#pragma mta assert no alias *sigma *Q *BC *delta *child *start
*QHead
#pragma mta use 100 streams
            for (n = Qstart; n < Qend; n++) {
<...>
```

The pragma was used on all parallel loops in the function to ensure that each thread that calls `kcent_core` is limited to the desired number of processors, which is 8 in this case. Also, because all of the parallel loops in `kcent_core` have the same limit for the max processors, the compiler will not need to put the loops into different parallel regions because of a mismatch in limits. Grouping the loops into one region can help reduce the cost of going parallel and improve performance by avoiding starting and tearing down multiple parallel regions.

The `kcentrality` function was modified to compute the number of threads at runtime based on the number of processors used by the application and the number of processors used per thread in `kcent_core`. The number of threads, `INC`, is a preprocessor macro in version 0.4 of GraphCT. However, the modifications to `kcentrality` changed `INC` to a variable that is computed at runtime. The following code snippet shows the changes made to `kcentrality`. Again, `MAX_PROCS` used in the example below has been defined as 8.

```
<...>
/*Compute INC based on the number of processors we're using
and limiting each thread to MAX_PROCS processors (in
kcent_core()).*/
   int INC;
   INC = mta_get_max_teams();
   INC = INC / MAX_PROCS;
   INC = MTA_INT_MAX(1, INC);
<...>
#pragma mta loop future
   for(x=0; x<INC; x++)
   {
     <...>
      for (int claimedk = int_fetch_add (&k, 1);
           claimedk < Vs;
           claimedk = int_fetch_add (&k, 1))
      {
         <...>
         kcent_core(G, BC, K, s, Q, dist, sigma, marks, QHead,
                    child, child_count);
         <...>
      }
    }
<...>
```

These changes to GraphCT helped the betweenness centrality kernel have better load balancing across the threads and achieve higher system utilization, which improved the performance and scalability of the kernel.

References

[1] “GraphCT – Streaming Graph Analysis”, http://trac.research.cc.gatech.edu/graphs/wiki/GraphCT, May 4, 2010.