MAESTRO:
Program Thread and Synchronization Interface

(version 0.1)

TR-08-01

Allan Porterfield

March 18, 2008

RENCI Technical Report Series
http://www.renci.org/techreports

MAESTRO: Program Thread and Synchronization Interface
(version 0.1 - March 2008)

Allan Porterfield
Renaissance Computing Institute
Chapel Hill, NC 27517
akp@renci.org

March 27, 2008

Abstract

The MAESTRO API is intended to support a simple thread programming model for compilers and other automated tools. It is not expected to be visible to application programmers. MAESTRO must support other programming models, such as MPI and OpenMP, efficiently. The goal is to provide light weight threads than can run quickly and efficiently on a dynamic number of hardware resources running within a single address space. MAESTRO allows more threads to be created than resources exist and is responsible for mapping the work to existing resources. The goal is to allow easier parallel programming and porting between different machines by removing, knowledge of system size from the application. MAESTRO supplies explicit thread and parallel loop creation and uses an explicit join point for thread synchronization. In addition the MAESTRO API has a variety of synchronization mechanisms, ranging from optimistic point-to-point though global barriers. This draft is expected to be heavily modified as initial implementation shows the numerous shortfalls.

1 Overview

This is a working document. Implementation comments should be *italicized*.

1.1 Target

This prototype version of the threading and synchronization interface for MAESTRO is being designed with the intent that it will not be used by many application programmers. It is expected to be used by a small handful of runtime implementers to transform by hand a few applications to provide threads for use with the prototype runtime. If successful, the next step would be to expand the interface and use it between a compiler and the runtime. As a target for automated tools (and maniac early researchers) the MAESTRO API, can have painful syntax, but will need to support a complete programming environment and eventually other parallel programming models.

As MAESTRO implementation progresses, supporting applications will require that support for other threading packages be built on top of this MAESTRO interface. Over time this interface should be extended enough to allow most MPI and OpenMP calls to be implemented as library calls to the MAESTRO threading and synchronization functions. Though these libraries, most application programs should be able to run on MAESTRO.

1.2 Requirements

MAESTRO wants to provide a environment where the programmer can specify parallelism wherever it exists and not be concerned with the size of system the application on which it will be executed.

The requirements for this API will increase over time. For the initial MAESTRO prototype, for a small number of applications (or pseudo applications), it must be able to generate enough parallelism to saturate a single microprocessor. A mechanism to create threads from parallel regions and parallel loops will suffice. In preparation for later work, the first

version will be able to support nested parallelism for both regions and loops. Expressing parallelism is rarely enough to generate a correct program, some from of synchronization between the threads is needed. A general many-to-many mechanism is included to allow synchronization and data movement between the threads. In addition, to the general mechanism a simple one-to-one mechanism is also supported. This one-to-one mechanism should allow the exploration of specialized synchronization mechanisms that exist in some current microprocessors. For both synchronization protocols the programmer should not be concerned with the amount of fast resources the hardware provides. The interface should support the ability to remap synchronization to the best available hardware. Since we expect that hardware assist for various flavors of synchronization will become available in the future, separate mechanisms will allow easier exploitation of the features.

MAESTRO will map the expressed parallelism efficiently onto whatever hardware exists. By virtualizing software thread on top of hardware streams the mapping can be dynamic and change during an application. By allowing various flavors of synchronization to be expressed differently, the runtime can use the most efficient hardware solution available. Since the interface will virtualize any hardware assist, correctness can be guaranteed even when the application needs exceeds what the hardware provides.

1.3 Approach

To provide the greatest flexibility, the underlying threads must be as light as possible. The typical time to start a new thread should be hundreds of cycles or just a few references to non-cached memory. The heavier threads required by other models can be built on top of MAESTRO's lightweight versions. Much of the thinking is influenced by the experiences with the Tera MTA/Cray XMT programming model[1]. The model there is an explicit thread fork and an implicit thread join. Each thread has an associated variable and the join is implicit wherever the variable is used for the first time after thread creation. Although convenient for the application programmer, the implicit join point is hard to implement without compiler support. Since the initial implementation will be implemented as library calls, the interface will have an explicit join point for every non-speculative thread. This may increase the difficulty for expressing complicated parallelism but should be straightforward for most parallel constructs. The threads will be thought of as unnamed. Each thread will be associated with some memory location that can be used as an handle, but these handles will be reused and are only valid while a thread is active.

Many(most?) parallel threads will be created from independent execution of loop iterations. Using the same mechanism that we create a thread and placing it inside a loop is certainly feasible, but this loses information which the runtime can profitably use during execution. By allowing parallel loops to be explicit declared, the MAESTRO runtime can decide the amount of resources being applied to the loop. Knowing the number of active workers on a parallel loop, we can schedule better and allows simpler barrier implementations. Much of the power benefits from using MAESTRO will be built on the ability dynamically change the number of hardware streams executing a parallel loop.

One important feature that the MAESTRO runtime supports is nested parallelism. Efficiently mapping nested parallel threads to dynamic hardware resources will a challenge, but allowing nesting greatly simplifies programming (especially libraries) and increases performance (by reducing the likelihood of regions without enough threads to occupy the hardware resources). Supporting nested parallelism is not explicit anywhere in the interface, but is not excluded and any implementation needs to support.

2 Model

MAESTRO's programming model is intended to run on the multi-core microprocessors. The processors will have thread counts upwards from four to the low teens. A large quantity of shared address space will be available allowing the threads to work on a single application. A limited amount communication between threads for synchronization.

2.1 Terms and Concepts

- thread - software entity - defines a chunk of work - may or may not be actively executing at any point in time
- stream - hardware entity that executes a software thread

- future - a variable which will be available at some point time - can also be the thread that computes the value that will be in that variable

Not used yet – but will be needed later

- team - software entity - a group of threads working on a single parallel loop
- team - hardware entity - a group of streams running with a single address space working on the same application

Do barriers divide loops?
How to loop iterations get assigned?

3 Threading Interface

Initially, MAESTRO will have a very simple thread creation interface. This specification in written in a C-like C++, but straight C and Fortran versions will be created on demand following the same philosophy. Threads and loops can be created and you can wait on a thread's completion.

3.1 Classes

- **class thread**
  - enum ThreadStatus status
  - enum ThreadType type
  - int iterations
  - int active_streams
  - func
- **enum ThreadStatus**
  - RUNNING - actively executing on a stream
  - QUEUED - awaiting allocation to a stream
  - BLOCKED - execution started, blocked on a synchronization variable
  - WAITING - execution started, awaiting execution after being BLOCKED on a synchronization variable
  - MISSING - either thread has completed or bad thread id
- **enum ThreadType**
  - NORMAL - created with **CreateThread**
  - UNNAMED - created with **CreateUnnamedThread**
  - LOOP - created with **CreateParallelLoop**
  - NONE - thread data structure not associated with a software thread

3.2 Functions

- **void InitializeRuntime(int size)** - used before any threads or synchronization variables created. **size** specifies the initial size of the synchronization variable pool. The pool should be allowed to grow but startup sizing will be useful. It is possible that this will not be needed for threading, but here until it is determined to be superfluous.
- **class thread t CreateThread(func)** - standard software thread creation. Allocates a thread data structure from a preallocated pool. Function should quickly return and add a new execution thread for some stream to execute (hopefully in parallel with the current thread).

- **void CreateUnnamedThread(func)** - similar to **CreateThread**, except there is no checking to see if the computation will ever complete. **thread** data structure is allocated and used for the duration of execution but is released on thread termination. (This means that thread structure will need to know which type of call created it.) This thread type is intended for speculative executions. *This functionality may not be needed but seems easy to add and implement initial and may be useful. If no real uses of it occur, it can be removed during a future iteration.*

- **void CreateParallelLoop(func, int count)** - Quickly allocate threads for a parallel loop. Each iteration is assumed to be independent of all other iterations and can be executed in any order. Only one **thread** structure will be allocated for the loop. This will greatly reduce the parallel loop startup overhead. The number of streams executing the loop will be used to optimize the loop chunk size and barrier/termination computation. *Manipulating chunk size will be an interesting parameter when combined with power control; small - lots of overhead getting chunks, large - can not react to hardware saturation quickly.*

  By having the data structure present for the loop execution, streams can show up after the beginning of the loop and join the computation. Dynamic thread addition will allow better execution of complicated nested parallelism. More important to MAESTRO will be the ability to remove streams from a loop. This ability (only allowed while examining the thread structure for more iterations) will be critical in allowing correct execution even while the runtime is shutting down cores to save power.

  The **thread** data structure is free on completion of the loop.

- **void ThreadJoin(class thread)** have this thread wait for another thread to complete. It frees thread after determine that the execution was completed.

  Each **thread** data structure should have on indication of whether it is currently being executed. If a thread has not started when another thread attempts to **ThreadJoin** with it, the current thread should just pick it up and execute it. On completion the thread returns to its original thread rather than looking for work. If the thread is being executed, a scheduling problem of how long to wait arises. A two-phase protocol where we wait for the amount of time to remove ourselves from the stream and then still waiting go ahead and swap this thread out of the hardware resource is likely to be preferable. *Creating a new thread should be much lighter weight than swapping out an active thread. Swapping an active thread will involve saving all register state from the active and should require a large number of memory accesses.*

- **enum TheadStatus ThreadTest(class thread)** - test the status of a thread which can be running, queued, blocked, waiting or missing(completed). May be used in **ThreadJoin** to determine if entering the thread wait logic is appropriate or whether switching to the thread is best.

- **bool e Barrier(class thread)** - only valid for parallel loops - prevent any loop iteration form progressing beyond this point until all loop iterations have reached this point. When a thread reaches this point, it may return to the beginning of the loop (or last barrier) and start iterations that have not otherwise been started. If no iterations remain, wait on some mechanism for the last iteration to complete before continuing. *This is the only point where streams may be removed from the computation of a loop. By interrupting no iteration chunks, removing a stream is greatly simplified and switching to a new iteration should be much quicker than a full stream swap.*

  It may be easier to implement this as the completion of one loop and the start of a second loop. Whether implemented as one loop or two, some care should be taken to try and pass out the same iterations before and after the barrier. This should improve the performance of any private caches. This is not a hard rule, if streams are removed by the power control mechanism iteration layout will change.

3.3 Examples

3.3.1 Parallel Section

Figure 1 starts a simple parallel thread to search a tree with two threads. The parallelism is simple to add. Initialize the parallel runtime, wrap the parallel region within a function which is called by **CreateThread** and join the threads with **ThreadJoin** before using any value computed in the created thread. The performance of this would greatly depend on the balance of the two halves of the tree.

```
int main() {
    int left,right;
    InitializeRuntime(100);
    class Tree tree = ReadTree(inoutfile); // read in tree

    // create thread to search left half of the tree
    class thread t = CreateThread(&Search(&(tree->left), 10, &left));
    right = Search(&(tree->right), 10, &right);

    ThreadJoin(t); // wait for left thread to terminate

    printf("found value 10 \%d times in tree\\n\", val, left+right);
    return 0;
}
void Search(class Tree *tree, int value, int * result){
   int res = 0;
   if (tree==NULL) return;
   if (tree->value == value) res++;

   Search(&(tree->left),value,&res);
   Search(&(tree->right),value,&res);

   result = res;
   return;
}
```

Figure 1: Parallel Region Example

```
int main() {
    int result;
    InitializeRuntime(100);
    class Tree tree = ReadTree(inoutfile); // read in tree
    result = Search(&(tree), 10, &result);
    printf("found value 10 \%d times in tree\\n\", val, result);
    return 0;
}
void Search(class Tree *tree, int value, int * result){
   int left = 0,right = 0, result = 0;
   if (tree==NULL) return;
   if (tree->value == value) result++;

   // create new thread for left half of tree
   class thread t = CreateThread(&Search(&(tree->left), 10, &left));

   Search(&(tree->right),value,&right);

   ThreadJoin(t);
   result += left+right;
   return;
}
```

Figure 2: Nested Parallel Regions Example

By nesting parallel regions, we can greatly improve available parallelism and the ability of the runtime to adjust to where the work actually is located. in Figure 2, the same computation now forks off a thread at every node in the tree. For a big tree hundreds or thousands of threads can now be working on the problem, greatly decreasing time to completion. The change required is only moving the creation of new threads to with the created thread. From a performance viewpoint, generating new parallel threads would want to be terminated once the tree size was reduced to the point that a serial implementation would be faster.

3.3.2 Parallel Loop

Generating a parallel loop is also simple, as shown in Figure 3. The difficulty in generating parallel loops will be separate the loop body into a function that can be passed to **CreateParallelLoop**(). If the loop iterations are truly parallel this transform is simple, for reductions or linear recurrences the inter iteration synchronization or the tempera array to store the results until back to a serial region will complicate the implementation.

The time required by this transformation will limit the number and size of hand transformed applications, that MAESTRO is tested with. As a target for automated tools, this limitation should not be severe.

3.3.3 Nested Parallelism and Barriers

We can nest parallel loops as in Figure 4. Many applications will find it possible to identify much more parallelism than hardware resources exist to execute the loops

4 Synchronization

Initially, MAESTRO will support 2 different interfaces to synchronization. For simple single producer/single consumer problems, a hash table will be used to allow an unbounded software implementation to be placed efficiently on any hardware mechanisms the vendors may eventually provide. This method will have implicit allocation and deallocation.

```
int main () {
InitializeRuntime(100); // once near beginning of program

int Array[500];
InitArray(Array);

.
.
.

CreateParallelLoop(&DoubleIteration(),500);

.
.
.
}

// created function -- the body of the parallel loop
void DoubleIteration(int iteration) {
   Array[iteration] += Array[iteration];
}
```

Figure 3: Parallel Loop Example

For more involved synchronization needs, the second method will have explicit allocation and deallocation combined with a variety of read and write routines.

The second method will support many more complicated synchronization patterns than the first and may be more efficient particularly for places where arrays of synchronization variables are required.

Comment on extending this for specialized mechanisms for all 4 synchronization types in later API versions.

4.1 Classes

- **class sync**
  - enum SyncValues state;
  - void * value;
- **enum SyncValues**
  - FULL - a value is current in the variable
  - EMPTY - a value is not currently in the variable and no thread is waiting
  - WAITING - a value is not currently in the variable and a thread is waiting for this variable to become FULL
  - UNINITIALIZED - the variable has never been used

4.2 Functions

4.2.1 Producer/Consumer Synchronization

- **void PCInit(int size)** - initialize Producer/Consumer Array and hash table for use by synchronization where there is only one reader and the overheads cannot be reduced by preallocating an array. **size** is a hint about the size to initially set the hash table up to support.

```
\begin{figure}
\begin{verbatim}
int main () {
InitializeRuntime(100); // once near beginning of program

int Array[500];
InitArray(Array);

.
.
.

CreateParallelLoop(&DoubleArray(),500);

.
.
.
}

// created function -- the body of the parallel loop
void DoubleArray(int iteration) {

CreateParallelLoop(&DoubleRow(iteration),500);

}

// created function -- the body of the parallel loop
void DoubleRow(int row, int column) {
   Array[iteration][column] += Array[iteration][column];
}
```

Figure 4: Nested Loop Example

- **bool status PCSet(int hashkey, void *value)** - set hash entry in PC table. When the producer (this thread) is first, create a hash table entry using the **hashkey**, mark the entry as full and save the **value** in the entry. When the consumer has arrived and is waiting, place the **value** in the entry and wake the consumer thread. If the entry has already present and full return an error.

- **bool status PCAcquire(int hashkey, void *value)** - wait on hash entry in the PC table. When the consumer (this thread) is first, insert a handle to wake up this thread into the entry and wait for the entry to be **PCSet()**. When the producer has arrived (either with or without waiting), get the **value** from the entry, free it and continue. If there is already a thread waiting at this **hashkey** return an error.

4.2.2 Example

A simple producer consumer usage pattern is shown in Figure 5. One thread produces a series of values (square of iteration count) and the other thread prints the values to standard out. The programming overhead to creating two threads and passing data between them is relatively minor. It is important to note that the two threads need to generate the same sequence of hash values, to make sure that data is received in the correct order. When the consumer looks for data with a hash value that the producer never uses, the thread will block and the program will deadlock.

Can we use the address of a variable as a hash key to pass data in that variable?
Can we add debugging function to make the detection of deadlock easier to locate and correct?

4.2.3 General Synchronization

- **class sync * SyncAcquireArray(int hashkey, int size, enum status)** - acquire a pointer to a continuous array of synchronization variables of size **size**. If the **hashkey** is new, it will allocate a new array. If the **hashkey** already exists, it will return a pointer to the already existing array. If the new array request a different size or for some other reason the allocation cannot be satisfied, **status** is an enum that identifies the problem. **status** will also tell the programmer whether this was a new allocation or an existing one was found.

- **bool SyncFreeArray(int hashkey)** - frees the previously allocated **class sync** array allocated with the **hashkey**. Returns false if the item could not be freed (duplicate frees etc), true otherwise.

- **value SyncReadEmpty(class sync s)** - execution waits until the state of the **class sync** variable is full, then returns the value, changes the state to empty and wakes a waiting writer.

- **value SyncReadLeaveFull(class sync s)** - execution waits until the state of the **class sync** variable is full, returns the value and leaves the state full. This is used to implement 'future' semantics, see discussion later in the section.

- **void SyncWriteNoWait(class sync s, int value)** - write **value** into the **class sync** variable regardless of initial state of the variable and leave the state full. *should probably fail if a waiting thread exists.*

- **void SyncWriteWaitForEmpty(class sync s, int value)** - wait for **class sync** variable to be empty and then write **value** into the variable, leave the state full and awaken any waiting reader.

  If the head of the waiting list is a **SyncReadEmpty** *only awake one reader, if the head is a* **SyncReadLeaveFull** *awaken the whole list of waiters. This may not be optimal but awaking the whole list if only one can go generates continuation(bad) and not starting everyone if they leave the state alone can result in threads never restarted(BAD!).*

- **void SyncEmpty(class sync s)** - force state of the **class sync** variable to be empty and the value to be 0.

  A check to see if a thread is waiting on this variable and an error may be a good idea.

- **enum SyncValues SyncStatus(class sync s)** return the current status of the **class sync** variable **s** - empty, thread waiting, full, uninitialized

```
#define ItemsToGenerate 500

int main() {

    // create consumer thread
    class thread t = CreateThread(&PrintAnswer());

    // producer

    PCInit(ItemsToGenerate);

    for(x = 0; x < ItemsToGenerate; x++){

       void * v;
       // compute value for v
       v = (void*)(x*x);

       bool ok = PCSet(hash + x, v);
       if (ok == 1) printf("duplicate hash value");
    }

    ThreadJoin(t); // wait for consumer thread to terminate
    return 0;
}

void PrintAnswer(){

   //consumer

   for(x = 0; x < ItemsToGenerate; x++){

        void *v;
        bool ok = PCAcquire(hash + x, v)
        if (ok == 1) printf("duplicate hash value");

        int ans = (int)*v;
        printf("value \%d = \%d\n",x,ans);
   }
   return;
}
```

Figure 5: Simple Producer/Consumer Example

4.2.4 Examples

Array of Producer/consumer variables A producer consumer usage using an array of synchronization variables is shown in Figure 6. The same example as previous section.

Future The Cray XMT supports an extension to C/C++/FORTRAN inspired and named after the future construct in LISP. On the Cray XMT it allows a value to be computed in parallel to be specified. Any use of that variable will wait for the value to be computed. Once the value is present it may be used multiple times without penalty. It is implemented by having the synchronization mechanism not empty the value after reading. Allows a parallel region to compute a value that will be reused, without having to either copy out of the synchronization variable or even know where the first use will be(example value used inside arms of a switch statement. Don't know which one and don't want to force the parallelization to complete before evaluating the switch condition.)

One simple use variation for this mode is shown in Figure 7. In the example `Compute_s`, which fills s with a value, will run in parallel with the elided portion of the function and the main thread will wait for the value to be computed only when actually needed. One shortcoming of this example is that it assumes child thread continues to execute and joins back with this thread later, making thread scheduling hints difficult.

In Figure 8, a similar code sequence the **ThreadJoin()**'s can be used as hints to MAESTRO to help thread scheduling. Even though 3 parallel threads are started, there is a preferred ordering based on some switch condition. By using **ThreadJoin()**, we can execute the thread most need by the main thread first.

Wavefront

5 Bibliography & References Cited

[1] Gail A. Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton J. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *ICS*, pages 188–197, 1992.

```
int main() {

    // create consumer thread
    class thread t = CreateThread(PrintAnswer);

    // producer

    s = SyncAcquireArray(hash,ItemsToGenerate,status);
    if (status == NEW_SYNC) SyncEmpty(s);

    for ( x =0; x < ItemsToGenerate; x++) {

      // compute value
      value = x*x;

      SyncWriteWaitForEmpty(&s[x], value);
    }

    ThreadJoin(t); // wait for consumer thread to terminate
    bool ok = SyncFreeArray(hash);
    if(!ok) printf("error freeing sync array");

    return 0;
}


void PrintAnswer(){

    // consumer

    s = SyncAcquireArray(hash,ItemsToGenerate,status);
    if (status == NEW_SYNC) SyncEmpty(s);

    for ( x =0; x < ItemsToGenerate; x++) {
        v = SyncReadEmpty(&s[x]);
        int ans = (int)*v;
        printf("value \%d = \%d\n",x,ans);

    }
    return;
}
```

Figure 6: Array Producer/Consumer Example

```
class sync *s = SyncAcquireArray(hash,1,status);
if (status == NEW\_SYNC) SyncEmpty(s);

class thread t = CreateThread(Compute_s());

.
.
.

switch(cond) {
  case 1:
      ...
      foo = SyncReadLeaveFull(s);
      break;
  case 2:
      ...
      bar = SyncReadLeaveFull(s);
      break;
  case 3:
      ...
      baz = SyncReadLeaveFull(s);
      break;
  default:
      break;
}

foobar = SyncReadLeaveFull(s);

.
.
.
```

Figure 7: Example using **SyncReadLeaveFull**

```
class sync *r = SyncAcquireArray(hash,1,status);
if (status == NEW\_SYNC) SyncEmpty(r);
class sync *s = SyncAcquireArray(hash,1,status);
if (status == NEW\_SYNC) SyncEmpty(s);
class sync *t = SyncAcquireArray(hash,1,status);
if (status == NEW\_SYNC) SyncEmpty(t);

class thread t1 = CreateThread(Compute_r());
class thread t2 = CreateThread(Compute_s());
class thread t3 = CreateThread(Compute_t());

.
.
.

switch(cond) {
  case 1:
      ...
      ThreadJoin(t1);
      foo = SyncReadLeaveFull(r);
      break;
  case 2:
      ...
      ThreadJoin(t2);
      bar = SyncReadLeaveFull(s);
      break;
  case 3:
      ...
      ThreadJoin(t3);
      baz = SyncReadLeaveFull(t);
      break;
  default:
      break;
}

.
.
.

if(ThreadStatus(t1 $neq$ COMPLETED) ThreadJoin(t1);
if(ThreadStatus(t2 $neq$ COMPLETED) ThreadJoin(t2);
if(ThreadStatus(t3 $neq$ COMPLETED) ThreadJoin(t3);
foobar = SyncReadLeaveFull(r) + SyncReadLeaveFull(s) + SyncReadLeaveFull(t);
```

Figure 8: Example using **ThreadJoin**