A “Hands-on” Introduction to OpenMP*

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

Outline

- Unit 1: Getting started with OpenMP
  - Mod1: Introduction to parallel programming
  - Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - Disc 1: Hello world and how threads work
- Unit 2: The core features of OpenMP
  - Mod 3: Creating Threads (the Pi program)
  - Disc 2: The simple Pi program and why it sucks
  - Mod 4: Synchronization (Pi program revisited)
  - Disc 3: Synchronization overhead and eliminating false sharing
  - Mod 5: Parallel Loops (making the Pi program simple)
  - Disc 4: Pi program wrap-up
- Unit 3: Working with OpenMP
  - Mod 6: Synchronize single masters and stuff
  - Mod 7: Data environment
  - Disc 5: Debugging OpenMP programs
  - Mod 8: Skills practice … linked lists and OpenMP
  - Disc 6: Different ways to traverse linked lists
- Unit 4: a few advanced OpenMP topics
  - Mod 8: Tasks (linked lists the easy way)
  - Disc 7: Understanding Tasks
  - Mod 8: The scary stuff … Memory model, atomics, and flush (pairwise synch).
  - Disc 8: The pitfalls of pairwise synchronization
  - Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - Disc 9: Random number generators
- Unit 5: Recapitulation

Outline

- Unit 1: Getting started with OpenMP
  - Mod1: Introduction to parallel programming
  - Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - Disc 1: Hello world and how threads work
- Unit 2: The core features of OpenMP
  - Mod 3: Creating Threads (the Pi program)
  - Disc 2: The simple Pi program and why it sucks
  - Mod 4: Synchronization (Pi program revisited)
  - Disc 3: Synchronization overhead and eliminating false sharing
  - Mod 5: Parallel Loops (making the Pi program simple)
  - Disc 4: Pi program wrap-up
- Unit 3: Working with OpenMP
  - Mod 6: Synchronize single masters and stuff
  - Mod 7: Data environment
  - Disc 5: Debugging OpenMP programs
  - Mod 8: Skills practice … linked lists and OpenMP
  - Disc 6: Different ways to traverse linked lists
- Unit 4: a few advanced OpenMP topics
  - Mod 8: Tasks (linked lists the easy way)
  - Disc 7: Understanding Tasks
  - Mod 8: The scary stuff … Memory model, atomics, and flush (pairwise synch).
  - Disc 8: The pitfalls of pairwise synchronization
  - Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - Disc 9: Random number generators
- Unit 5: Recapitulation

Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

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

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
          do_many_things();
#pragma omp master
          {     exchange_boundaries();   }
#pragma omp  barrier
          do_many_other_things();
}
```

Single worksharing Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{

   #pragma omp sections
   {
   #pragma omp section
            X_calculation();
   #pragma omp section
            y_calculation();
   #pragma omp section
            z_calculation();
   }

}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

Synchronization: Lock routines

- **Simple Lock routines:**
  - **A simple lock is available if it is unset.**
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()

A lock implies a memory fence (a "flush") of all thread visible variables

- **Nested Locks**
  - **A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function**
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Synchronization: Simple Locks

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
      omp_init_lock(&hist_locks[i]);    hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
    ival = (int)  sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
        hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
  }

for(i=0;i<NBUCKETS; i++)
  omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Runtime Library routines

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - **Are we in an active parallel region?**
    - omp_in_parallel()
  - **Do you want the system to dynamically vary the number of threads from one parallel construct to another?**
    - omp_set_dynamic, omp_get_dynamic();
  - **How many processors in the system?**
    - omp_num_procs()

…plus a few less commonly used routines.

Runtime Library routines

- **To use a known, fixed number of threads in a program, (1)tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.**

```
#include <omp.h>
void main()
{   int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
   {     int id=omp_get_thread_num();
#pragma omp single
              num_threads = omp_get_num_threads();
           do_lots_of_stuff(id);
   }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.