Introduction to openMP and Shared-Memory Programming

Joey Bernard, ACEnet CRC, UNB - 2009

What is shared-memory?

- Parallel programs have multiple execution units talking to each other
- Shared-memory has this communication happening in the memory of the computer
- This means that all of the execution units need to be on one physical machine (with some exceptions)
- Forms:
  - openMP
  - POSIX threads
  - Home made memory sharing

How does OpenMP work?

- Compiler directives in the source code
- If the compiler doesn't recognize them, they act as comments

- In C/C++ these are pragmas
  #pragma omp parallel

- In FORTRAN these are directives
  !$OMP PARALLEL

How do we compile?

- Compiling simply requires a command line option

PGI Compilers	-mp
Sun Studio 12	-openmp –O3
GNU 4	-fopenmp

How do we run?

- We need to tell the program how many instances to run
  - Bash – export OMP_NUM_THREADS=4
  - Csh – setenv OMP_NUM_THREADS 4
- Is OpenMP allowed to change this?
  - Bash – export OMP_DYNAMIC=FALSE
  - Csh – setenv OMP_DYNAMIC TRUE
- Can we nest the parallel parts?
  - Bash – export OMP_NESTED=TRUE
  - Csh – setenv OMP_NESTED FALSE

Learning about ourselves

- We may need to know how many processors and how many threads there are
  int omp_get_num_procs()
  int omp_get_num_threads()

- Find out who we are
  int omp_get_thread_num()

- Set the number of threads to use
  void omp_set_num_threads(int t)

- FORTRAN has the same subroutines

Loops – for (C/C++)

- Most common form of parallelism is through the use of loops

```
#pragma omp parallel for
    for (i = first; i < size; i += prime) marked[i] = 1;
```

- At runtime, we need to know how many iterations
- This means we can't use break, return or exit from inside the loop, or goto's that lead to outside of the loop
- Continue is OK since the loop count doesn't change

Loops – do (FORTRAN)

- In FORTRAN, the most common loop is the do loop

```
!$omp do
      do loop statements
!$omp end do
```

Privatizing Variables

- Normally, all variables are shared among threads

```
#pragma omp parallel for
   for (i=0; i<max; i++)
      for (j=0; j<max2; j++)
            do_something
```

- The variable j will get badly abused by access from all threads

Privatizing Variables

- We can make variables private to fix this

```
#pragma omp parallel for private(j)
   for (i=0; i<max; i++)
      for (j=0; j<max2; j++)
            do_something
```

- A new variable j will get created for each thread

Privatizing Variables

- What if we want the value to enter the loop?

```
#pragma omp parallel for private(j) firstprivate(j)
    for (i=0; i<max; i++)
        for (j=0; j<max2; j++)
                do_something
```

- What if we want the value to leave the loop?

```
#pragma omp parallel for private(j) lastprivate(j)
```

Critical Sections

- What if we have a statement in our loop that needs to be done sequentially

  area += add_on;
- In this statement, area is read and added to. If this is done by two threads at the same time, they read in the same value, then add their “add_on” value.
- We miss one of the “add_on” values – race condition
- How do we fix this?

Critical Sections

```
#pragma omp parallel for private(x)
   for (i=0; i<n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
   }
```

- This change puts a lock on the line calculating area so that only one thread has access at a time

```
!$omp critical
   …
!$omp end critical
```

General parallelism

- What if we need to have some chunk of code executed by each thread?

```
#pragma omp parallel
   {
      do_subroutine();
      a = b + c;
   }
```

- We can use private here as well to get private variables within the block
- We can use critical here if we have code that is sensitive to a race condition

How to drop out of parallel code

- What if we have a statement in a parallel section that should only be executed by one thread
  e.g. a print statement should only be printed once

```
#pragma omp parallel
   {
   …
#pragma omp single
   printf(“The answer is %d”, answer);
   …
   }
```

Coordination

- Sometimes threads need to meet at a common point so that they can coordinate
- We can set a barrier – threads can't get past the barrier until everyone gets there

```
!$omp barrier
```

```
#pragma omp barrier
```

POSIX Threads

- Threads are more low level then OpenMP
- The programmer is responsible for more work
- This means the programmer also has much more control

  - pthread_create
  - pthread_join
  - pthread_exit
  - pthread_mutex_lock
  - pthread_mutex_unlock

Creating a thread

- Threads need to be created and given the code to execute

```
int pthread_create(pthread_t *tid, const pthread_attr_t
    *attr, void * (*func)(void *), void *arg);
```

- The first argument holds the resulting thread ID
- The second argument holds the thread attributes
- The third argument is a pointer to the function
- The fourth argument is a pointer to an argument

Working inside the thread

- Who are we?
  pthread_t pthread_self()

- Once we're done, how do we finish?
  void pthread_exit(void *status);

- The pointer to status needs to point to a global variable, since all local variables disappear when the thread exits

Waiting for a thread to finish

- Once we spawn off a thread, we may want to wait for it to finish

```
int pthread_join(pthread_t tid, void **status);
```

- This blocks until the thread finishes, so don't use it unless you really need it

Critical sections again

- We can get race conditions with POSIX threads when we try and read/write from global variables
- It's the programmer's responsibility to deal with it

```
int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
```

- If you need to initialize/reinitialize a mutex

```
pthread_mutex_t mptr =
    PTHREAD_MUTEX_INITIALIZER
int pthread_mutex_init(pthread_mutex_t *mptr,
    pthread_mutexattr_t *attr);
```