CISC 372: Parallel Computing
OpenMP, Part 4

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Synchronization constructs

These constructs control synchronization among threads.

- `barrier`
- `ordered`
- `critical`
- `atomic`
- `master`

Note:

- except for `barrier`, these do not impose barriers

Synchronization constructs: `barrier`

```
#pragma omp barrier
```

- a stand-alone construct: does not modify subsequent block
- all threads in a team block until every thread in team has reached the barrier
- all threads in a team must encounter the same sequence of worksharing and barrier constructs
- note most constructs already impose a barrier at end
  - so explicit barrier is usually unnecessary
- main use: control accesses to shared variables to avoid data races
  - e.g., one thread writes, `barrier`, other threads read

Synchronization constructs: `ordered`

```
#pragma omp ordered
S
```

- must occur inside an omp `for` loop using clause `ordered`
- the block S will be executed in iteration order
  - write $S[i]$ for execution of S in i-th iteration
  - $S[i]$ will complete before $S[i+1]$ begins $(i = 0, 1, \ldots)$
- this essentially forces iterations to execute sequentially
  - except there can be some overlap of the non-ordered code
- typical use cases: print statements, debugging

Synchronization constructs: `critical`

```
#pragma omp critical [ ( name ) ]
S
```

- declares S to be a critical section
- at any time: at most one thread can be executing inside a critical region named *name*
- in order for a thread to start executing S:
  - no other thread can be inside a critical region with same name as S
- other threads may execute concurrently
  - as long as they are not in a critical region with same name as S
- *name* is optional
  - all critical regions with no name are considered to have the same name, distinct from all named critical regions
- common uses: printing, computation of max or min, ...
  - complex modifications to shared data
    - don't want any other thread to "see" the data in an intermediate state
    - all threads access the data through critical regions with the same name
    - very similar to use of locks or Java's `synchronized`

Synchronization constructs: `atomic`

```
#pragma omp atomic
S
```

- following statement executes in one atomic step
- no other threads can intervene
- S must be a simple assignment statement of a certain form (see OpenMP 4.0 Sec. 2.12.6)
- acceptable examples
  - `x++`, `x--`, `++x`, or `--x`
  - `x = x` *binop* *expr*
  - `x` *binop*`=` *expr*
  - `x =` *expr* *binop* `x`
- binary operators: `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`
- no function calls or other kinds of expressions
- can be more efficient than `critical`
  - can take advantage of low-level atomic operations

Synchronization constructs: `master`

```
#pragma omp master
S
```

- the associated block is executed by only the master thread of the team
- no barrier
- similer to `single`, but recall:
  - `single` can choose any thread (not just master)
  - `single` has a barrier at end by default

OpenMP locks

- a type and functions to lock/unlock, similar to Pthread's mutexes
- considered "lower-level" primitives than the directive-based constructs
- type: `omp_lock_t`
- functions
  - `void omp_init_lock(omp_lock_t *lock);`
  - `void omp_destroy_lock(omp_lock_t *lock);`
  - `void omp_set_lock(omp_lock_t *lock);`
  - `void omp_unset_lock(omp_lock_t *lock);`

The threadprivate directive

- consider the program semiprivate.c. What is the output?

```
#include <stdio.h>
#include <omp.h>
int x = 99;
void f() {
  x=omp_get_thread_num();
}
int main() {
#pragma omp parallel private(x) num_threads(5)
  {
    int tid = omp_get_thread_num();
    f();
    printf("Thread %d: x = %d\n", tid, x);
  }
  printf("Final x = %d\n", x);
}
```

`semiprivate.c`: output

```
omp$ gcc-mp-4.8 -fopenmp semiprivate.c
omp$ ./a.out
Thread 1: x = -348111896
Thread 2: x = -348111896
Thread 3: x = -348111896
Thread 4: x = 19907219
Thread 0: x = 0
Final x = 0
omp$
```

Why?

- the `private` clause affects only references to the variable inside the construct (the static extent), not the region (dynamic extent).
- if you want `x` to be `private` everywhere, you need to use the `threadprivate` directive.

threadprivate.c

```
#include <stdio.h>
#include <omp.h>
int x;
#pragma omp threadprivate(x)
void f() {
  // this updates the private copy of x...
  x=omp_get_thread_num();
}
int main() {
#pragma omp parallel num_threads(5)
  {
    int tid = omp_get_thread_num();
    f();
    printf("Thread %d: x = %d\n", tid, x);
  }
  printf("Final x = %d\n", x);
}
```

`threadprivate.c`: output

```
omp$ ./a.out
Thread 1: x = 1
Thread 2: x = 2
Thread 0: x = 0
Thread 3: x = 3
Thread 4: x = 4
Final x = 0
omp$
```

- use this when you have a global variable you wish to share between functions
  - and you want it private
- you can even make the shared variable persist between parallel regions
  - certain requirements must be met
  - in particular, all the parallel regions in which variable is used must have same number of threads
- note the variable must be initialized inside a parallel region before it is used

MPI/OpenMP hybrid programs

- for clusters of multicore nodes, you may
  - use MPI everywhere: one MPI process per core, or
  - use an MPI+threads "hybrid" model
    - one MPI process per node
    - threads within a node map to cores
    - threads may be specified by Pthreads, OpenMP, or some other thread API
- advantages of "MPI everywhere"
  - simpler
  - re-use all your old MPI programs with no changes!
  - performance often pretty good
    - sends and receives within a node implemented using `memcpy` or similar
- advantages of MPI+threads
  - might get better time performance
  - often uses less memory
    - in MPI everywhere, common data structures must be duplicated on every process, i.e., core
    - in MPI+threads, need only one copy of data structure on each node

MPI with threads

- MPI does not have threads, but it does specify how MPI may interact with threads
- an MPI implementation may or may not be thread-compliant
- a process can request a certain level of thread support from MPI
- MPI will respond with the best thread support it can provide for that request
- different processes can request (and receive) different levels of support
- the interfaces for messages, etc., are the same whether or not there are multiple threads
  - hence a message sent by one thread on process p looks exactly the same as a message sent by another thread on p
  - there is no way for another process to tell which thread it came from
  - a message sent by p to another process q cannot target a particular thread on q
  - to participate in a collective routine, only one thread in p should call the collective functions

Four levels of thread support are specified

1. MPI_THREAD_SINGLE: only one thread will execute
2. MPI_THREAD_FUNNELED
   - multiple threads may execute, but only the master thread will call MPI functions
3. MPI_THREAD_SERIALIZED
   - multiple threads may execute and call MPI functions, but at any time only one thread will be calling MPI
   - user needs to synchronize threads properly to ensure this
4. MPI_THREAD_MULTIPLE
   - multiples threads may call MPI functions at the same time
   - the implementation will ensure these calls are sequentialized

The following function should be called instead of MPI_Init:

```
int MPI_Init_thread(int *argc, char ***argv,
                    int required, int *provided)
```

Thread queries

```
int MPI_Query_thread(int *provided);
```

- returns provided level of thread support

```
int MPI_Is_thread_main(int *flag)
```

- true if calling thread is main thread, false otherwise