Introduction to Parallel & Distributed Programming

Lec 05—Shared Memory Programming

Subodh Sharma | Jan 13, 2026

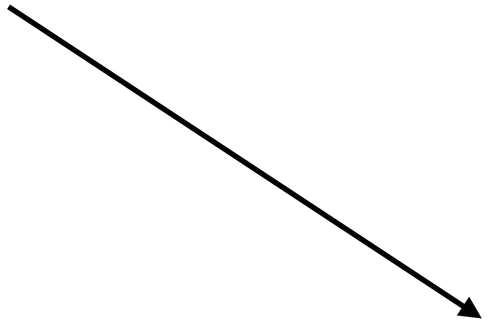
Recap

- **OpenMP primitives - parallel, shared, private**

More on OpenMP

Thread control divergence

- Introducing thread-id specific control flow



```
int main (){

int numt, tid ;

#pragma omp parallel private(tid) {

tid = omp_get_thread_num();

if (tid == 0)
  numt = omp_get_num_threads();

}
```

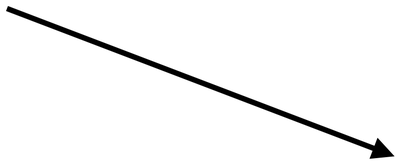
Alternate way:

```
#pragma omp single nowait
{
   numt = omp_get_num_threads();
} // implicit barrier
```

More on OpenMP

Use of explicit barriers as synchronisation

- **Barrier** — synchronisation primitive for data consistency and ordered execution
- **Semantics**: No thread can advance beyond the barrier point until all have arrived



```
int main (){

int numt, tid ;

#pragma omp parallel shared (numt) private(tid)
{
  tid = omp_get_thread_num();

   if(tid == 0) numt = omp_get_num_threads();

#pragma omp barrier

  printf("hello world %d of %d\n", tid, numt);

}

}
```

Other Synchronisations: Locks

Help achieve orderly access — Mutual Exclusion

- `omp_init_lock(omp_lock_t *)` - Nestable locks are declared with the type
  - omp_nest_lock_t
- `omp_set_lock()` –- acquires lock
- `omp_unset_lock()` – releases lock
- `omp_destroy_lock()` – free memory
- `omp_test_lock ()` – set the lock if available, else return w/o blocking

```
while(!flag) {
    flag = omp_test_lock();
}
```

Locks help in achieving race-freedom in accessing critical region

What is Mutual Exclusion?

And how is it difference from atomicity?

- Mutual Exclusion: Only **one thread** accessing a **critical section** at any moment
- How is it different from atomicity?
  - Atomicity guarantees that a threads actions appear as a single indivisible step to other threads
- Food for thought — **Does atomicity imply mutual exclusion? Or the other way around**?

```
Shared balances (vault)
         |
         | guarded by LOCK
         v
+--------------------------+
|  critical section        |
|  balance0 -= 1;          |
|  balance1 += 1;          |
+--------------------------+


T0: enters  ---------------------- exits
T1:         waits...... enters ------ exits
T2:                     waits........ enters ...
```

Other Synchronisations: Critical

- **Critical directive**: makes the runtime execute the associated region of code only by one thread at a time.

```
    #pragma omp critical (update_sum) // name of the block
       sum += sumLocal;
}
```

- Example: **Sum of array elements**

Other Synchronisations: Atomic

- **Atomic directive**: runtime ensures that the memory location is updated atomically by only one thread at a time.

- Example: **Sum of array elements**