Introduction to Parallel & Distributed Programming

Lec 06—OpenMP

Subodh Sharma | Jan 16, 2026

RECAP: 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

RECAP: 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.

```
#pragma omp atomic
  sum += sumLocal;
```

- Example: **Sum of array elements**

Work Sharing Constructs: FOR

Scheduling for iteration space

- We have already discussed the: `#pragma omp for`
- Specify the chunk-size — with static the assignment is in **round-robin** mode

```
#pragma omp parallel for schedule (static, chunk-size)
{
  for (i = 0; i < N; i++)
    do_stuff();  // a[i] += b[i]
}
```

Best for uniform work per iteration and low overhead.

- Dynamic Scheduling: On demand, thread requests chunk-size one finished with its allotted task

```
#pragma omp parallel for schedule (dynamic, chunk-size)
{
  for (i = 0; i < N; i++)
    do_stuff();  // a[i] += b[i]
}
```

Higher overhead but better load balance

Work Sharing Constructs: FOR

Reductions

- Reductions **avoid races**
- Supported operators:
  - +, *, min, max
  - &, |, &&, ||
  - ^

```
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) sum += a[i];
```

Work Sharing Constructs: Sections

- Each thread executes the region within the section
- Each section is expected only **ONCE**
- **Good use cases:**
  - Pipelined tasks
  - Overlapping I/O and compute
  - Multimodal streaming tasks
  - Divide-and-conquer

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { compute_A(); }

    #pragma omp section
    { compute_B(); }

    #pragma omp section
    { compute_C(); }
  } // implicit barrier at end of sections (unless nowait)
}
```