EE 382C/361C: Multicore Computing	**Fall 2016**

Lecture 7: September 15

Lecturer: Vijay Garg — *Scribe: Tong Zhou*

7.1 Outline

The lecture covered the following topics:

- Review of Lamport's Fast Mutex Algorithm
- Introduction to OpenMP
- Two new ways to find maximum from N numbers

7.2 Lamport's Fast Mutex Algorithm

Lamport's fast mutex algorithm uses two shared variables X and Y and a shared single-writer-multiple-reader registers $flag[i]$. A process P_i can enter the critical section either along a *fast* path or along a *slow* path. The algorithm is shown below.

The variable $flag[i]$ is set to value *up* if P_i is actively contending for mutual exclusion using the fast path. When Y is -1, the door is open. A process P_i closes the door by updating Y with i. Lamport's Fast Mutex Algorithm is deadlock-free but allows starvation of individual processes.

In the algorithm, process i first sets X to i, and then checks the value of Y. When it finds $Y = -1$, it sets Y to i and then checks the value of X. If $X = i$ process i enters its critical section along the *fast* path. At most one process can enter its critical section along *fast* path. If a process finds $X \neq i$ then it delays itself by looping until it sees that all the values in the array *flag* are *down*. Checking these n values in array *flag* plays two roles:

1. Say that process i enters its critical section along *slow* path, if it finds $Y = i$ after exiting the for-loop. A consequence of having observed all the values in the array *flag* to be *down* is that the value of Y will not be changed thereafter until process i leaves the critical section. This follows because every other contending process either reads $Y \neq -1$ and waits at $waitUntil(Y == -1)$, or reads $Y = 0$ and then finished the assignment $Y := i$ before setting $flag[i]$ to *down*. Hence, once process i finds $Y = i$ after the loop, no other process can change the value of Y until process i sets Y to -1 in its exit code. It follows that at most one process can enter along *slow* path.

2. The for-loop ensures that if a process enters the critical section along *fast* path, then any other process is prevented from entering along *slow* path. To see this, observe that when a process i enters its critical section along *fast* path, its $flag[i]$ remains *up*. Thus, if another process tries to enter along *slow* path it will find that $flag[i] == up$ and will have to wait until process i exits its critical section and set $flag[i]$ to *down*.

Algorithm 1 Lamport's Fast Mutex Algorithm

```
var
    X, Y: int initially -1;
    flag: array[1..n] of {down, up};

acquire(int i)
{
    while(true):
        flag[i] = up;
        X = i;
        if(Y != -1): // splitter's left
            flag[i] = down;
            waitUntil(Y == -1):
            continue;
        else:
            Y = i;
            if(X == i): return; // fast path
            else: // splitter's right
                flag[i] = down;
                waitUntil(∀j : flag[j] == down);
                if(Y == i): return; // slow path
                else:
                    waitUntil(Y == -1);
                    continue;
}

release(int i)
{
    Y = -1;
    flag[i] = down;
}
```

7.3 OpenMP

OpenMP(Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, processor architectures and operating systems.

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met. The sequential program evolves into a parallel program.

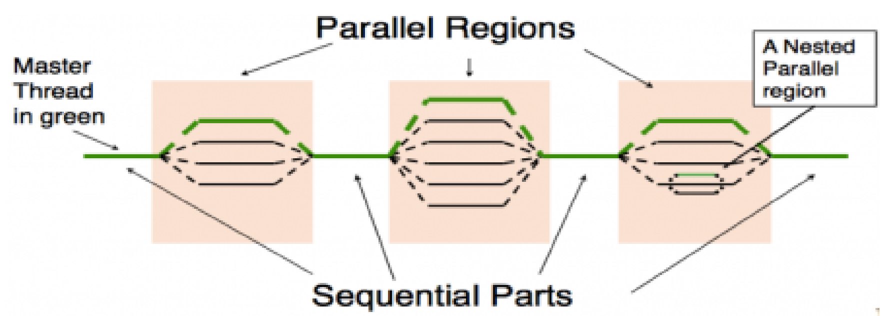

Figure 7.1: Fork-Join Parallelism

We create threads in OpenMP with the parallel construct. For example, to create a 4 thread Parallel region. Each thread calls $foo(ID, A)$ for $ID = 0$ to 3.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

Thread creation: parallel regions

7.3.1 Synchronization

Synchronization is used to impose order constraints and to protect access to shared data. In OpenMP, we have 4 high level synchronization:

- critical
- atomic
- barrier
- ordered

7.3.1.1 Synchronization: critical

Mutual exclusion: only one thread at a time can enter a *critical* region. Example code:

```
float result;
#pragma omp parallel
{
    float B; int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for(i=id; i<N; i = i+nthrds){
        B = foo(i); // expensive computation
    #pragma omp critical
        consume(B, result)
    }
}
```

7.3.1.2 Synchronization: atomic

Atomic provides mutual exclusion but only applies to the update of a memory location. In the following example code, $foo's$ are run in parallel but r updated atomically.

```
int n,r;
#pragma omp parallel shared(n,r)
{
    for(i=0; i<n; i = i++){
    #pragma omp atomic
        r += foo(i); // expensive computation
    }
}
```

7.3.1.3 Synchronization: barrier

Barrier: Each thread waits until all threads arrive. Example code:

```
#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);
}
```

7.3.1.4 Synchronization: ordered

The *ordered* region executes in the sequential. In the following example code, array *a* updated in any order but printed in sequential order

```
#pragma omp parallel for
{
  for(i=0; i<n; i = i++){
    tid = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", tid, i);
    a[i] += i;
    #pragma omp ordered
    {
      printf("Thread %d prints value of a[%d]=%d\n",
                tid, i, a[i]);
    }
  }
}
```

7.3.1.5 Synchronization: locks

We also have low level synchronization *locks* (both simple and nested).

- 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()
- 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()

7.3.2 Data Environment and Data Attributes

All variables declared outside parallel for pragma are shared by default, except for loop index. *for* index variable is private. One can selectively change storage attributes for constructs using the following clauses.

- shared
- private
- firstprivate

7.3.2.1 Private Clausse

private(var) creates a new local copy of var for each thread. The value is uninitialized and is undefined after the region.

7.3.2.2 firstprivate Clausse

firstprivate is a special case of private. Initializes each private copy with the corresponding value from the master thread. In the following example, All copies have value of tmp initialized as 0.

```
void Foo()
{
    int tmp = 0;
    #pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j) {
        tmp += j;
    }
}
```

7.3.2.3 lastprivate Clausse

lastprivate passes the value of a private from the last iteration to a global variable. In the following example, All copies have value of tmp initialized as 0. After the for loop, the variable tmp has the value from the last iteration (i.e. j=99).

```
void Foo()
{
    int tmp = 0;
    #pragma omp for firstprivate(tmp) lastprivate(tmp)
    for (int j = 0; j < 100; ++j)
        tmp += j;
         printf("%d\n", tmp);
}
```

7.4 Find Maximum from N Numbers

In the first lecture, we mentioned four ways to find maximum from N numbers.

	time complexity	space complexity
Sequential	$O(N)$	$O(N)$
Binary Tree	$O(log(N))$	$O(N)$
All-pair	$O(1)$	$O(N^2)$
Comparison	$O(1)$	$O(N^{3/2})$

7.4.1 DoublyLog Algorithm

As we mentioned in the All-pair Algorithm, N numbers can be divided into $\sqrt{N}$ groups. In each group, there are $\sqrt{N}$ numbers. In DoublyLog Algorithm, we further divide every $\sqrt{N}$ group into $\sqrt{\sqrt{N}}$ sub-groups. We keep doing so until the size of every group is 1 or 2. The height of this tree structure is $log(log(N))$. For each layer, the computing time complexity is $O(N)$. The total time complexity for DoublyLog Algorithm is $O(Nlog(log(N)))$.

7.4.2 Cascaded Algorithm

In Cascaded Algorithm, we combine Sequential Algorithm and DoublyLog Algorithm to reduce the number of processors that we need.

Divide N numbers into $N/log(log(N))$ groups. Then, in every group, there are $log(log(N))$ numbers. We use Sequential Algorithm to get $N/log(log(N))$ maximum candidates in every group. For these maximum candidates, we use DoublyLog Algorithm.

For Sequential Algorithm, the work complexity is $O(N)$ and time complexity is $log(log(N))$. To select maximum from candidates with DoublyLog Algorithm, the time complexity is $O(log(log(N)))$ and work complexity is $O(N)$ (There are$N/(log(log(N)))$ groups. Every group has $log(log(N))$ time complexity).

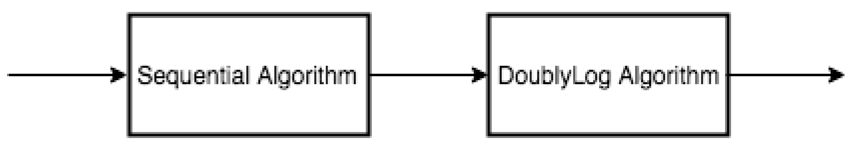

Figure 7.2: Cascaded Algorithm Model

References

[1] Leslie Lamport, A Fast Mutual Exclusion Algorithm(1986).

[2] Michael Merritt, Gadi Taubenfeld, Speeding Lamport?s Fast Mutual Exclusion Algorithm(1991).

[3] Tim Mattson, A "Hands-on" Introduction to OpenMP (2008).

[4] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore