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

Lecture 7: September 15

Lecturer: Vijay Garg *Scribe: Yichi Zhang*

7.1 Introduction

In this lecture, we will first review the Lamport's Fast Mutex Algorithm and then discuss a couple of basic synchronized constructs of openMP. Finally, we will solve the puzzle mentioned in the first class with best solution balancing the complexity of $Work$ and $Time$.

Outline is lied as follows:
- Lamport's Fast Mutex Algorithm (review)
- Discuss basic knowledge of OpenMP explained with examples
- Solve the previous puzzle (and come up with a new one)

7.2 Lamport's Fast Mutex Algorithm

Code of Fast Mutex Algorithm on next page. Fast Mutex Algorithm is using Splitter method and can be drawn like Figure 7.1.

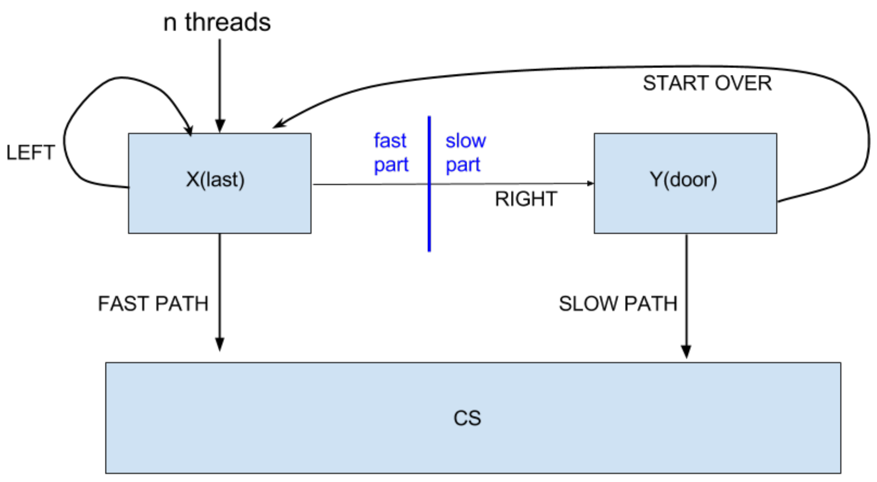

Figure 7.1: Fast Mutex Algorithm Model

Explanation for this algorithm:

There are two shared registers, X and Y, that each processor and read and write. For X, it holds the last process that acquires permission to get into the critical session and is initially set to -1. For Y, it simulates the door, $Y = -1$ indicates the door is open and $Y = i$ means door closed and process P_i is the

```
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) {                          // splitters left
        flag[i] := down;
        waitUntil(Y == -1);
        continue;
    }
    else {
        Y := i;
        if (X == i)                         // success with splitter
            return;                         // fast path
        else {                              splitter's right
            flag[i] := down;
            forall j:
                waitUntil(flag[j] == down);
            if (Y == i) return;             // slow path
            else {
                waitUntil(Y == -1);
                continue;
            }
        }
    }

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

Lamport's Fast Mutex Algorithm

last one that get entered the door. Also, there are n-length array $flag[n]$ for each process, $flag[i]$ could be *up* or *down*, *up* means P_i is contending for mutex using the fast path, *down* means other cases.

As can see from Figure 7.1, at first, process P_i goes into block X, set $X = i$, $flag[i] = up$ and check if the door is open($Y = -1$). If not open($Y >= 0$), it would go to the **left** route and try acquiring again some later. If open($Y = -1$), it would close the door by setting $Y = i$ and check if the last acquiring process is still itself ($X = i$), if so, it would go into CS through the **fast** route, if not, go to the Y block through the **right** route and set $flag[i] = down$. The last chance to get into CS in this iteration is to wait all flags to be down(which means that no process is in the CS) and if so, at this moment, if the last door-closing process is still P_i, then it will enter CS through the **slow** route, otherwise it would start over and try acquiring again.

To sum, Lamports Fast Mutex Algorithm can guarantee deadlock-free but can not guarantee starvation-free. Fast Mutex Algorithm improves Splitter method that it expands Splitter model as two part, fast part and slow part. For fast part, the **down** route in fast part is similar as the **down** route in Splitter, can guarantee that the |**Down**|, the number of threads through down route, is no more than 1, but could be 0. When this case happens, there is nothing in CS, so the slow part could help this algorithm to pick one thread enter CS through slow path, to ensure there is no deadlock. But unlike PetersonN Algorithm, Fast Mutex Algorithm is not first-come-first-serve algorithm, which means the first-come thread might have to wait infinitely. Thus it cannot guarantee starvation-free.

7.3 Basic synchronized constructs of OpenMP

OpenMP(Open Multi-Processing)[1] is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. By using openMP, it is easier to convert a serial code into parallel and also can run the same code as serial code. Figure 7.2 shows that openMP can run parallel code as serial flow, it can control when to distribute main thread into parallel processes and when to merge all parts as a real sequential program.

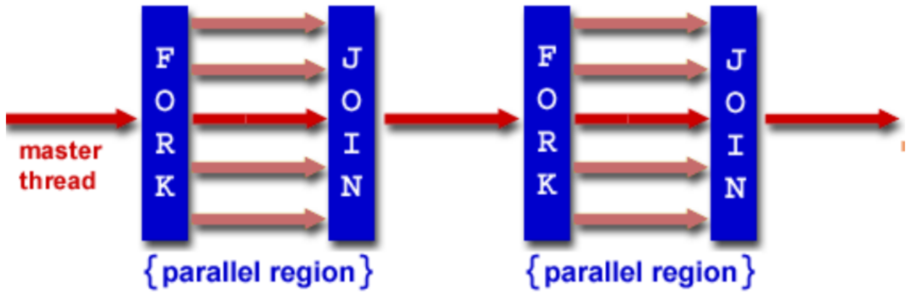

Figure 7.2: Simple Fork-Join Parallelism[2]

Most of the constructs in OpenMP are compiler directives. For example:

$$\#pragma\ omp\ parallel\ num\_threads(4)$$

is to set the number of threads as 4 (there is always one master thread as the main thread so to create 4 threads means to create 3 more threads besides the master thread).

7.3.1 OpenMP variables

Now we will introduce several variables of OpenMP with examples and explanation[3].

7.3.1.1 Critical

Variable *critical* indicates Critical Session, in the code, we have n processors and can execute all iterations in the for-loop at the same time but only one thread would run $Line10$ at a time.

7.3.1.2 Atomic

Variable *atomic* specifies that a memory location that will be updated atomically. For the code, all $foo(i)$ can be executed by n threads at the same time, but r will be updated atomically.

```
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);
    }
}
```

Variable Critical

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

Variable Atomic

7.3.1.3 Reduction

```
double sum=0.0, ave, A[MAX]; int i;
#pragma omp parallel for reduction (+ : sum)
for(i=0; i < MAX; i++) {
    sum += A[i];
}
ave = sum/MAX;
```

Variable Reduction

Variable *reduction* specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. For the above code, local copies of variable *sum* would be created and then combined.

7.3.1.4 Barrier

Variable *barrier* synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

7.3.1.5 Master construct

The target region will be executed only by the *master* thread.

7.3.1.6 Single construct

```
1  #pragma omp parallel
2  {
3     do_many_things();
4     #pragma omp single
5     {exchange_boundaries();}    // target region
6     #pragma omp barrier
7     do_many_other_things();
8  }
```

Variable Single construct

The target region will be executed by just one thread (any thread, not necessary the master). In this case, *Line* 5 would be executed by one thread, if the construct here is *master* instead of *single*, *Line* 5 would only be executed by master thread. And after *Line* 5, there is a barrier at *Line* 6, which makes other threads waiting for the working threads finish their work and then go to *Line* 7 at the same time.

7.3.1.7 Ordered

Locations updated in any order but printed in sequential order.

7.3.1.8 Lock

- omp_init_lock() – Initializes a simple lock.
- omp_set_lock() – Blocks thread execution until a lock is available.
- omp_unset_lock() – Releases a lock.
- omp_test_lock() – Attempts to set a lock but does not block thread execution.
- omp_destroy_lock() – Uninitializes a lock.

7.3.2 OpenMP data attributes

- *Shared* variable is shared among threads.
- *Private* variable(*var*) creates a new local copy of *var* for each thread.
- *Firstprivate* variable initializes each private copy with the corresponding value from the **master** thread.
- *Lastprivate* variable passes the value of a private from the **last** iteration to a global variable.

See the example on next page. We can pick one line from those three: *Line* 4, *Line* 5, *Line* 6. *Line* 4 indicates that the value(*tmp*) is uninitialized and undefined after the region. *Line* 5 indicates that all copies have value of *tmp* initialized as 0. *Line* 6 indicates that not only all copies have value of *tmp* initialized as 0, but after the *for* loop, the variable *tmp* has the value from the last iteration (i.e. j=99).

7.3.3 Dijkstra Algorithm

Dijkstra code example with OpenMP implementation can be seen from [4].

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

OpenMP data attributes

7.4 Solve the puzzle

From previous lectures, we have three ways to solve the find-maximum-from-N-numbers puzzle, listed as Table 7.1

Algorithm	Work	Time
Sequential	O(N)	O(N)
Binary	O(N)	O($log(N)$)
All-pair Algorithm	O(N^2)	O(1)

Table 7.1: Previous result of solving the puzzle

Now we will discuss two new methods to solve the puzzle with better performance.

7.4.1 DoublyLog Algorithm

In All-pair Algorithm, we have the idea that we can divide the N numbers into $\sqrt{N}$ groups with each group has $\sqrt{N}$ numbers. To expand that, we can more deeply divide each $\sqrt{N}$ groups into $\sqrt{\sqrt{N}}$ sub-groups and keep dividing until the size of each group is only 1 or 2. This idea is shown in Figure 7.3.

The height of this tree is k, which is $log(log(N))$, if we have enough processors, the time complexity will be the number of layers, which is $O(log(log(N)))$, and for each layer, the work we will be $O(N)$, so the total work will be $O(Nlog(log(N)))$.

7.4.2 Cascaded Algorithm

To better improve the performance, we try to cascade two different methods to get benefit from both of them. In Cascaded Algorithm, we synthesize Sequential Algorithm and DoublyLog Algorithm.

We divide N numbers into $log(log(N))$-length groups, which means there are $N/log(log(N))$ groups. With sequential algorithm, we can get $N/log(log(N))$ maximum candidates, and then we get the final maximum by compare these candidates using DoublyLog Algorithm. This idea is shown in Figure 7.4.

For the first part, work complexity for sequential is $O(N)$ and time complexity is size of each group,

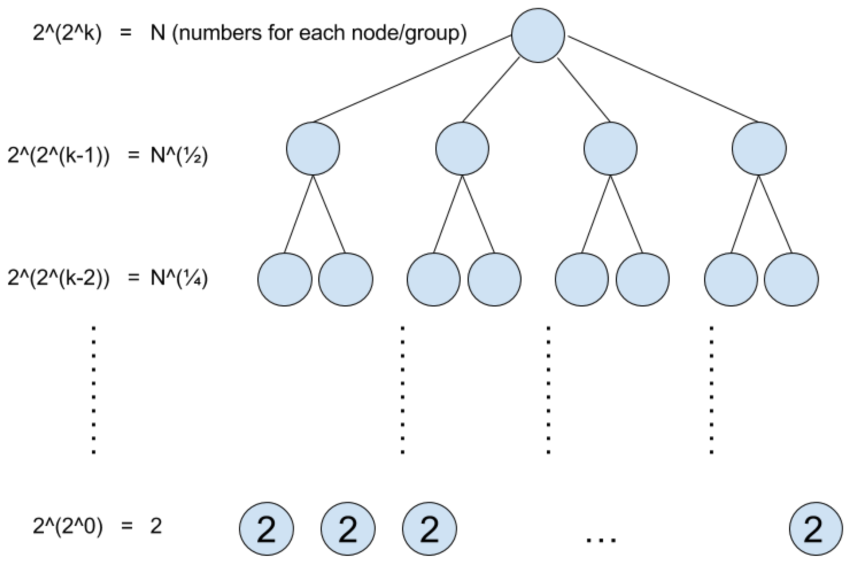

Figure 7.3: DoublyLog Algorithm Model

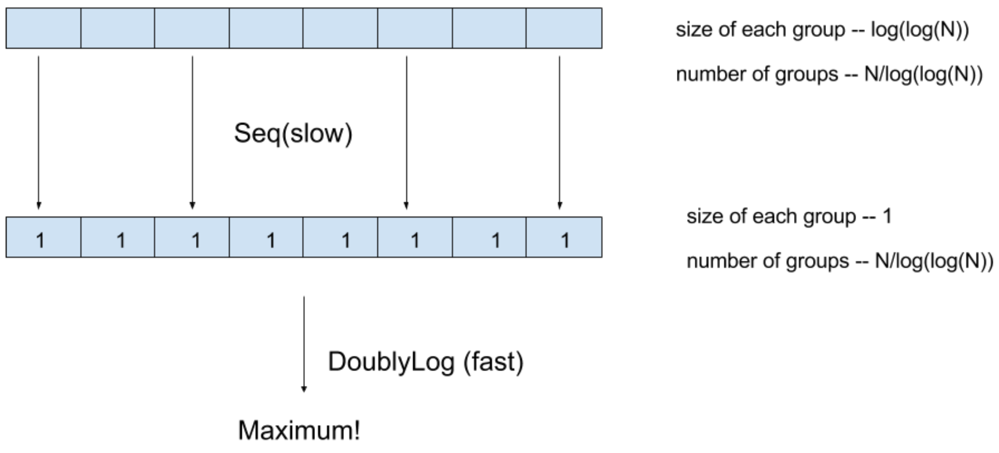
size of each group -- log(log(N))
number of groups -- N/log(log(N))
Seq(slow)
1
size of each group -- 1
number of groups -- N/log(log(N))
DoublyLog (fast)
Maximum!

Figure 7.4: Cascaded Algorithm Model

which is $O(log(log(N)))$. For the second part, according to DoublyLog ALgorithm, work complexity is $N/(log(log(N))) * (log(log(N))) = O(N)$, and time complexity is still $O(log(log(N)))$. Now we can expand Table 7.1 as Table 7.2.

Algorithm	Work	Time
Sequential	O(N)	O(N)
Binary	O(N)	O($log(N)$)
All-pair Algorithm	O(N^2)	O(1)
DoublyLog Algorithm	O($N * log(log(N))$)	O($log(log(N))$)
Cascaded Algorithm	O(N)	O($log(log(N))$)

Table 7.2: Current result of solving the puzzle

7.4.3 New puzzle

We have two sorted arrays, say $A[m]$ and $B[n]$, how could we merge them into a larger array, say $C[m+n]$, in parallel?

References

[1] https://en.wikipedia.org/wiki/OpenMP

[2] http://www.llnl.gov/computing/tutorials/openMP

[3] https://msdn.microsoft.com

[4] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/openMP/Dijkstra.c