OpenMP Synchronization (5A)

Young Won Lim
6/30/24

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
6/30/24

Based on

https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf

Synchronization (1)

threads communicate through shared variables.

- uncoordinated access of these variables

  can lead to undesired effects.

- <u>two</u> threads update (write) a shared variable

  in the same step of execution,

  the result is dependent on the way this variable is accessed.

  a race condition.

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf

Synchronization (2)

- to prevent race condition,

  the <u>access</u> to shared variables must be synchronized.

- synchronization can be time consuming.

- the barrier directive is set to synchronize all threads.

- <u>all</u> threads wait at the barrier

  until <u>all</u> of them have arrived.

Synchronization (3)

Synchronization II

• Synchronization imposes order constraints and is

used to protect access to shared data

• High level synchronization:

– critical

– atomic

– barrier

– ordered

• Low level synchronization

– flush

– locks (both simple and nested)

Critical (1)

Synchronization: critical

• Mutual exclusion: only one thread at a time can enter a critical

region.

```
{
double res;
#pragma omp parallel
{
double B;
int i, id, nthrds;
id = omp_get_thread_num();
nthrds = omp_get_num_threads();
for(i=id; i<niters; i+=nthrds){
B = some_work(i);
#pragma omp critical
consume(B,res);
}
}
} https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf
```

Critical (2)

Threads wait here: only one thread

at a time calls consume(). So this is

a piece of sequential code inside

the for loop.

Critical (3)

```
Sum = 0;

#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)

{

        TID = omp_get_thread_num();

        sumLocal = 0;

        #pragma omp for

                for (i=0; I<n; i++)

                        sumLocal += a[i];

        #pragma omp critical (update_sum)

        {

                sum += sumLocal;

                printf("TID=%d: sumLocal=%d sum=%d\n", TID, sumLocal, sum)

        }
} /* --- End of parallel region --- */
```

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf

Critical (4)

```
{
...
#pragma omp parallel
{
#pragma omp for nowait shared(best_cost)
for(i=0; i<N; i++){
int my_cost;
my_cost = estimate(i);
#pragma omp critical
{
if(best_cost < my_cost)
best_cost = my_cost;
}
}
}
}
```

Only one thread at a time executes if() statement. This ensures mutual exclusion when accessing shared data. Without critical, this will set up a race condition, in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable

Young Won Lim
6/30/24

Atomic (1)

atomic provides mutual exclusion but only applies to the

load/update of a memory location.

• This is a lightweight, special form of a critical section.

• It is applied only to the (single) assignment statement that

immediately follows it.

Atomic only protects the update of X.

```
26
{
…
#pragma omp parallel
{
double tmp, B;
….
#pragma omp atomic
{
X+=tmp;
}
}
}
```
https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf

Atomic (2)

Int ic, I, n;

Ic = 0;

#pragma omp parallel shared(n,ic) private(i)

     for (i=0; i++, I<n)

     {

          #pragma omp atomic

               ic = ic + 1;

     }

Atomic only protects the update of X.

"ic" is a counter. The atomic construct ensures that no updates
are lost when multiple threads are updating a counter value.

Atomic (3)

• Atomic construct may only be used together with an expression Atomic only protects the update of X.

statement with one of operations: +, *, -, /, &, ^, |, <<, >>

Int ic, I, n ;

Ic=0;

#pragma omp parallel shared(n,ic) private(i)

    for (i=0; i++, I<n)

      {

          #pragma omp atomic

            ic = ic + bigfunc();

      }

The atomic construct does not prevent multiple threads

from executing the function bigfunc() at the same time.

Barrier (1)

Suppose each of the following two loops are run in parallel

over i, this may give a wrong answer.

29

for(i= 0; i<N; i++)

a[i] = b[i] + c[i];

for(i= 0; i<N; i++)

d[i] = a[i] + b[i];

There could be a data race in a[].

Atomic only protects the update of X.

Barrier (2)

for(i= 0; i<N; i++)

a[i] = b[i] + c[i];

for(i= 0; i<N; i++)

d[i] = a[i] + b[i];

wait

barrier

To avoid race condition:

• NEED: All threads wait at the barrier point and only continue

when all threads have reached the barrier point.

Barrier syntax:

• #pragma omp barrier

Atomic only protects the update of X.

https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf

Barrier (3)

```
barrier: each threads waits until all threads arrive
31
#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(i,C);}
A[id]=big_calc4(id);
}
```

Implicit barrier at
the end of for
Construct

No implicit barrier
due to nowait

Implicit barrier at the end of
a parallel region

Barrier (4)

When to Use Barriers

• If data is updated asynchronously and data
integrity is at risk

• Examples:

– Between parts in the code that read and write the
same section of memory

– After one timestep/iteration in a numerical solver

• Barriers are expensive and also may not scale to a
large number of processors

Implicit barrier at
the end of for
Construct

No implicit barrier
due to nowait

Implicit barrier at the end of
a parallel region

References

[1]   ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2]   https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf