OpenMP Synchronization (5A)

Young Won Lim
7/24/24

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

This document was produced by using LibreOffice.

Young Won Lim
7/24/24

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.

- two 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.

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

Synchronization (3)

- synchronization imposes order constraints

- used to protect access to shared data

High level synchronization:

- critical

- atomic

- barrier

- ordered

Low level synchronization:

- flush

- locks (both simple and nested)

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

Critical (1)

Mutual exclusion: only <u>one</u> thread <u>at a time</u> 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);
        }
    }
}
```

Threads wait here: only one thread
at a time calls consume().
So this is a piece of sequential code
Inside the for loop.

Young Won Lim
7/24/24

Critical (2)

```
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 --- */
```

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

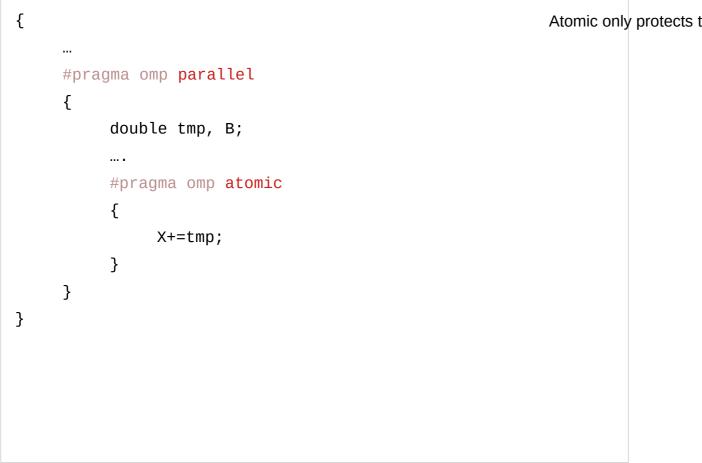https://www3.nd.edu/~zxu2/acms60212-40212-S12/Lec-11-02.pdf

Atomic (1-1)

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.

Atomic (1-2)

```
{

    …

    #pragma omp parallel

    {

        double tmp, B;

        ….

        #pragma omp atomic

        {

            X+=tmp;

        }

    }

}
```

Atomic only protects the update of X.

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.

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

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

Atomic only protects the update of X.

```
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[].

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.

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

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

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

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

Synchronization

Barrier Synchronization is "each thread should wait until all threads arrive"

Mutual exclusion: "only one thread can access x resource"

Constructs:

Note that pragmas are always applicable to the thing directly below. So use

blocks if you want to effect multiple things.

Critical

Only one resource may access at a time:

```
int sum = 0;
#pragma omp parallel
{
    int id = omp_get_thread_num();
#pragma omp critical
    sum += id;
}
```

Atomic

#pragma omp atomic

#pragma omp atomic read|write|update|capture

"If the low-level, high performance constructs for mutual exclusion exist on this hardware, use them. Otherwise act like this is a critical section."

Is there any benefit to critical sections in this case? Perhaps critical sections allow for function calls, where atomic only refers to a scalar set operation? Yes - video just said this is just available for simple binary operations to update values.

https://dev.to/winstonpuckett/openmp-notes-1cfa

Barrier

Wait until all threads process to this point before moving on:

```
#pragma omp parallel

{

    int id = omp_get_thread_num();

#pragma omp barrier

    printf("%d", id);

}
```

Flush (1)

Compilers are really good at optimizing where reads and writes occur. The order that you place operations in may not be the same order things happen if they are deemed to have equivalent results. This holds true for OpenMP. If you need to make reads and writes consistent, you need to use a Flush.

Creates a synchronization point that says, "you are guaranteed to have a consistent view of memory with the flush set." The flush set is the list of variables inside parenthesis passed to the flush pragma. When you leave off the flush set, everything must be consistent.

https://dev.to/winstonpuckett/openmp-notes-1cfa

Flush (2)

All reads and writes before the flush must resolve to memory before and reads or writes to memory after the flush set.

Flushes with overlapping flush sets may not be reordered with respect to each other.

For all intents and purposes, flush is equivalent to a fence in compiler terminology.

Flushes are hard to get right, so OpenMP provides implicit flushes at:

entering/exiting parallel regions
implicit/explicit barriers
entry/exit to critical sections
set/unset of a lock

https://dev.to/winstonpuckett/openmp-notes-1cfa

Flush (3)

Flush makes variables available to other threads.

If you spin lock on a variable, you also need to put a flush in the body of the loop. That forces the compiler to read the value every time not from a cache.

#pragma omp flush

#pragma omp flush(variableOne, variableTwo)

https://dev.to/winstonpuckett/openmp-notes-1cfa

Master

#pragma omp master schedules the next block on the main thread. For most use cases of master, you usually want a barrier on the next statement.

Young Won Lim
7/24/24

Critical (1)

Recall that critical sections are introduced in OpenMP with the critical directive:

#pragma omp critical

{

  /* critical section here */

}

This is an anonymous critical section. OpenMP will only allow one thread into this critical section at one time.

There is another usage of OpenMP critical sections wherein we have multiple critical sections that all must be preserved.

Critical (2)

Clearly, we do not want one thread to increment and another to decrement at the same time. The following approach will not work:

```
int global_data;

...

/* write in one location */
#pragma omp critical
global_data++;

...

/* write in another location */
#pragma omp critical
global_data--;
```

Critical (3)

For example, if we have some data that is written in multiple places in our program:

int global_data;

...

/* write in one location */
global_data++;

...

/* write in another location */
global_data--;

Critical (4)

We can link the two critical sections with a named critical section:

int global_data;

...

/* write in one location */

#pragma omp critical (global_data_lock)

global_data++;

...

/* write in another location */

#pragma omp critical (global_data_lock)

global_data--;

This causes OpenMP to enforce the rule that only one thread can be in either critical section at a time.

Atomic operation (1)

If, as in the example above, our critical section is a single assignment, OpenMP

provides a potentially more efficient way of protecting this.

OpenMP provides an atomic directive which, like critical, specifies the next

statement must be done by one thread at a time:

#pragma omp atomic

global_data++;

Atomic operation (2)

Unlike a critical directive:

    The statement under the directive can only be a single C assignment statement.

    It can be of the form: x++, ++x, x-- or --x.

    It can also be of the form x OP= expression where OP is some binary operator.

    No other statement is allowed.

The motivation for the atomic directive is that some processors provide single instructions for operations such as x++. These are called Fetch-and-add instructions.

As a rule, if your critical section can be done in an atomic directive, it should. It will not be slower, and might be faster.

References

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

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

Young Won Lim
7/24/24