IBM XL Fortran for AIX

New and Enhanced Atomic Constructs in OpenMP 3.1

By: Dorra Bouchiha

Level: Intermediate

May 2012

Contents

Before you start

About this Tutorial

This demo illustrates the use of the new and enhanced OpenMP atomic construct in the IBM XL Fortran compiler. The principles demonstrated here also apply to C and C++ programs using the IBM XL C/C++ compilers. The extension allows 3 new types of atomic access: *read*, *write* and *capture*. The new keyword *update* was also introduced to differentiate the behavior of the atomic construct in OpenMP 3.0 and earlier from the new behavior.

Objectives

- Use IBM XL Fortran Compiler to illustrate the use of the new atomic update, read write and capture constructs
- Total time: 45 minutes

Prerequisites

- Basic Unix skills
- Basic Source code compile and build experience
- Basic Fortran knowledge
- Basic OpenMp knowledge

System Requirements

http://www.ibm.com/software/awdtools/fortran/xlfortran/aix/sysreq

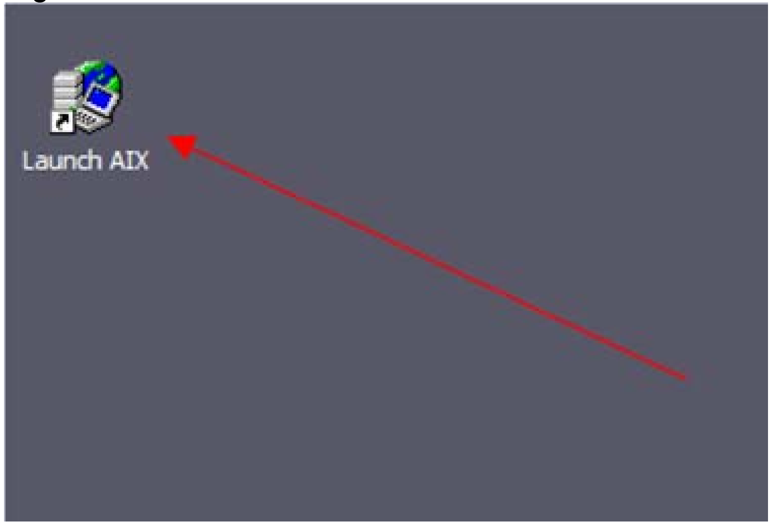
Glossary

IBM XL Fortran Compiler: The IBM® XL Fortran compiler offers advanced compiler and optimization technologies and is built on a common code base for easier porting of your applications between platforms. It complies with the latest Fortran international standards and industry specifications and supports a large array of common language features.

Getting Started

Start the Terminal Emulator to AIX System

Figure 1 Get Started

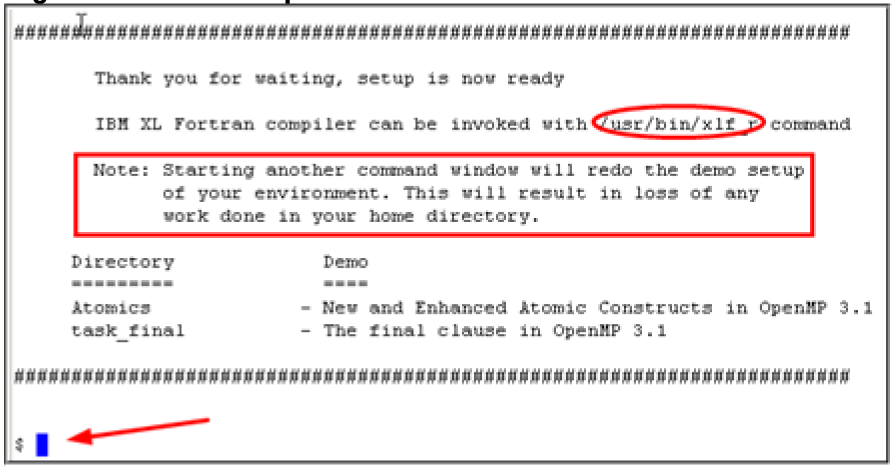


Double click the “Launch AIX” icon on the desktop (See Figure 1) to start the character terminal to AIX system.

Get Started with IBM XL Fortran Compiler

A successful login will result with the user presented with a menu of the demos hosted on the server. Type 19 and press Enter to select the “Fortran Exploring new OMP3.1 constructs” demo.

Figure 2 Demo Prepared



```
##########################################################################

       Thank you for waiting, setup is now ready

       IBM XL Fortran compiler can be invoked with /usr/bin/xlf_r command

       Note: Starting another command window will redo the demo setup
             of your environment. This will result in loss of any
             work done in your home directory.

     Directory                 Demo
     =========                 ====
     Atomics                 - New and Enhanced Atomic Constructs in OpenMP 3.1
     task_final              - The final clause in OpenMP 3.1

##########################################################################

$ 
```

On the terminal window you will see important information and the directory path to the compiler invocation command (See Figure 2 Demo Prepared *oval red*).

Note: Starting another command window will start the demonstration setup of your environment. This will result in loss of any work done in your home directory. This will impact any progress you have made on demo steps going forward.

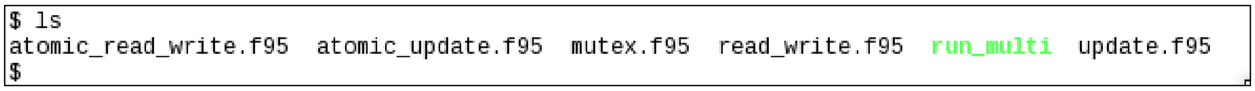
This demo does not require more than one terminal window. However, if you prefer more than one terminal window then you may open them before going forward.

The terminal window is now ready for commands. The Atomics demo resides in OMP31_F_demo/Atomics directory. Type the ls command to see the directory content (See Figure 3 Contents).

Command:

```
ls
```

Figure 3 Contents



```
$ ls
atomic_read_write.f95  atomic_update.f95  mutex.f95  read_write.f95  run_multi  update.f95
$
```

The Fortran programs are written in Fortran 95 and are compliant with OpenMP 3.1 specifications.

The demo consists of 5 samples. The first 2 (*update.f95, atomic_update.f95*) are used to illustrate the use of the **update clause** of the **atomic directive**. The variable X is updated by incrementing its old value by one. The update is done N times within the loop. On exiting the parallel loop, X is expected to be equal to N.

The next two demo samples (*read_write.f95* and *atomic_read_write.f95*) demonstrate the use of the **read** and **write** clauses to perform exclusive reads and/or writes to a variable. In these programs the variable being read/written to is of type double complex. Double complex uses a 16-byte representation and requires two hardware instructions to be written or read. The atomic directives ensure that each thread will read / write the variable as a whole and prevent other threads from reading/writing to it at the same time.

The last demo sample, *mutex.f95*, illustrates how atomic capture and atomic read can be used to implement a locking mechanism and avoid any race conditions.

Five Fortran programs are provided:

update.f95	Illustrates the race condition created when multiple threads try to simultaneously update the same storage location.
atomic_update.f95	The above race condition is avoided by using the **atomic update** directive.
read_write.f95	A variable is being read and written to concurrently by multiple threads. The program behaves unpredictably and may yield wrong results.
atomic_read_write.f95	The above race condition is avoided using the **read** and **write** clauses of the atomic construct.

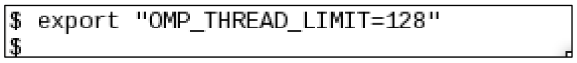
mutex.f95	Using **atomic capture** and **atomic read** to implement mutual exclusion

Setup necessary environment variables

Command:

```
export "OMP_THREAD_LIMIT=128"
```

Figure 4 OMP_NUM_THREADS **environment variable set-up**



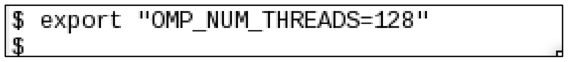
```
$ export "OMP_THREAD_LIMIT=128"
$
```

In this demo, we will use a large number of threads to better expose the race conditions when applicable. The environment variable OMP_THREAD_LIMIT sets the maximum number of threads to use for the whole program. Its default value is the number of available processors.

Command:

```
export "OMP_NUM_THREADS=128"
```

Figure 5 OMP_NUM_THREADS **environment variable set-up**



```
$ export "OMP_NUM_THREADS=128"
$
```

The OMP_NUM_THREADS environment variable controls the number of threads that will be created when the first parallel construct is encountered. In this case, the default number of threads created will be 128. It is easier to expose race conditions with a relatively large number of threads. As the machine used for this demo supports a smaller number of hardware threads, this may result in the program running slowly because of the overhead associated with thread scheduling and thread switching.

Get Started with the OpenMP Atomics:

Part 1: Using OMP3.1 atomic update

Steps:

First, let's start with the demo sample update.f95. In this program, the value of the variable X is incremented N times. On exiting the parallel region, X should be equal to N. If X is not equal to N, a STOP statement is executed and “STOP 10” is printed out to standard output.

The test illustrates a race condition on the update of the variable X. A race condition occurs when 2 or more threads try to read and/or write to a shared memory location at the same time. The result can be unpredictable as it depends on the thread scheduling algorithm. In the next sample, we will use the **atomic update** statement to ensure that each update is done atomically.

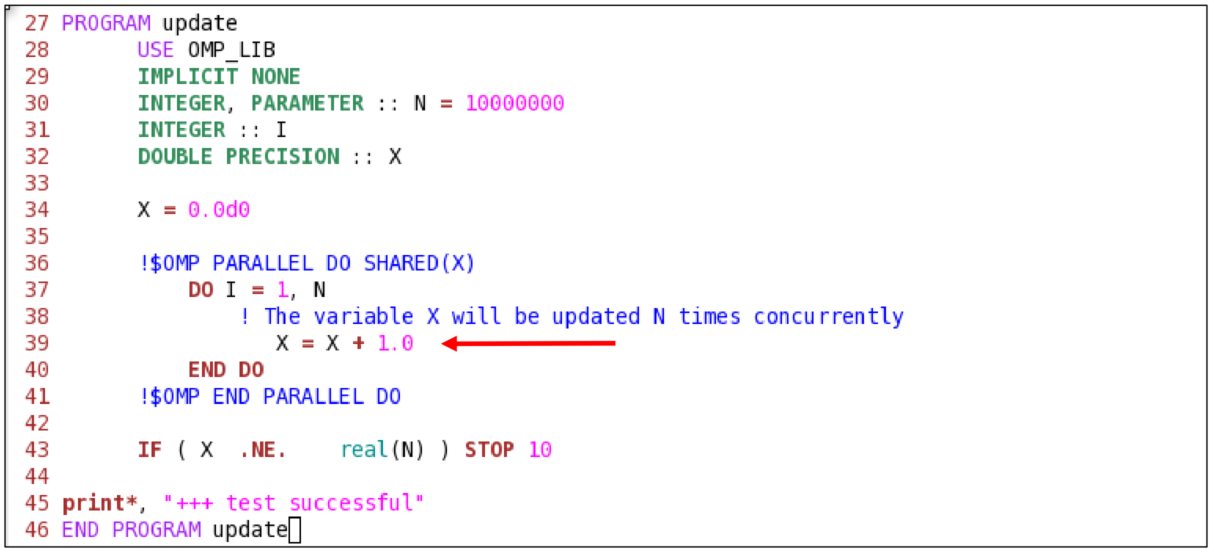
1. Compile and run update.f95

   View the source code of update.f95 (See figure 6)

   Command:

   ```
   vi update.f95
   ```

   Figure 6 Program Source Code



```
27 PROGRAM update
28       USE OMP_LIB
29       IMPLICIT NONE
30       INTEGER, PARAMETER :: N = 10000000
31       INTEGER :: I
32       DOUBLE PRECISION :: X
33
34       X = 0.0d0
35
36       !$OMP PARALLEL DO SHARED(X)
37           DO I = 1, N
38               ! The variable X will be updated N times concurrently
39                  X = X + 1.0
40           END DO
41       !$OMP END PARALLEL DO
42
43       IF ( X  .NE.    real(N) ) STOP 10
44
45 print*, "+++ test successful"
46 END PROGRAM update
```

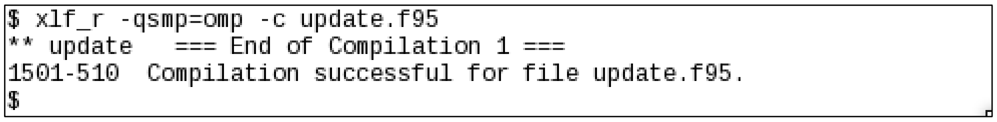
In line 39 of the program shown above, the memory location referred to as “X” is updated by each thread executing the loop. Any two threads can be updating X simultaneously.

Compile update.f95 (See figure 7)

Command:

```
xlf_r -qsmp=omp -c update.f95
```

Figure 7 Compile



```
$ xlf_r -qsmp=omp -c update.f95
** update   === End of Compilation 1 ===
1501-510  Compilation successful for file update.f95.
$
```

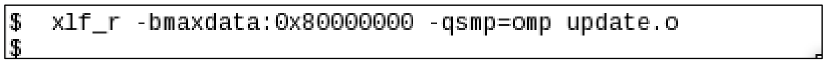
Link update.f95 (See figure 8)

Use the -bmaxdata linker option to increase the size of the static data and the heap on AIX to allow the creation of 128 threads.

Command:

```
xlf_r -bmaxdata:0x80000000 -qsmp=omp update.o
```

Figure 8 Link



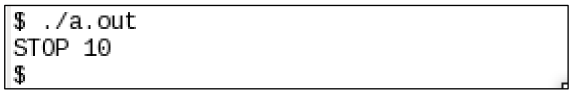
```
$  xlf_r -bmaxdata:0x80000000 -qsmp=omp update.o
$
```

Run update.f95 (See Figure 9).

Command:

```
./a.out
```

Figure 9 Run



```
$ ./a.out
STOP 10
$
```

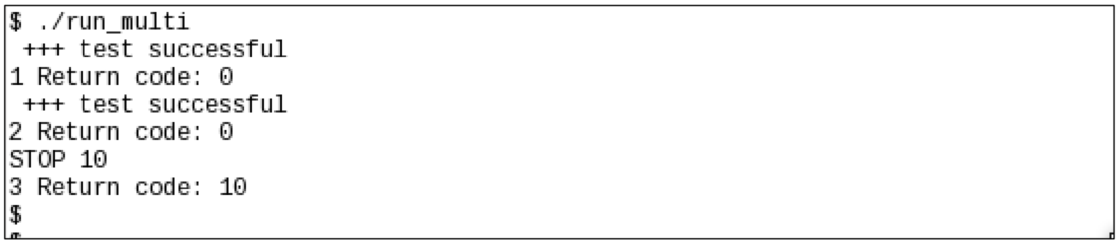
The program will terminate with a STOP statement because X does not have the expected value. Often multiple execution are needed to expose a race condition. The program may give the illusion that its correct by running successfully from time to time.

The script run_multi is provided to execute the output multiple times (See Figure 10). After a few iterations, the program will fail.

Command:

```
./run_multi
```

Figure 10 Multiple Runs



```
$ ./run_multi
 +++ test successful
1 Return code: 0
 +++ test successful
2 Return code: 0
STOP 10
3 Return code: 10
$
```

In the next sample, we will see how the atomic update construct allows us to write a thread safe program.

2. Compile and run atomic_update.f95

The following line was added to update.f to create atomic_update.f ensuring that the update of the memory location, designated by X, is performed atomically.

Program atomic_update.f

```
 ...
  39        !$OMP ATOMIC UPDATE
 ...
```

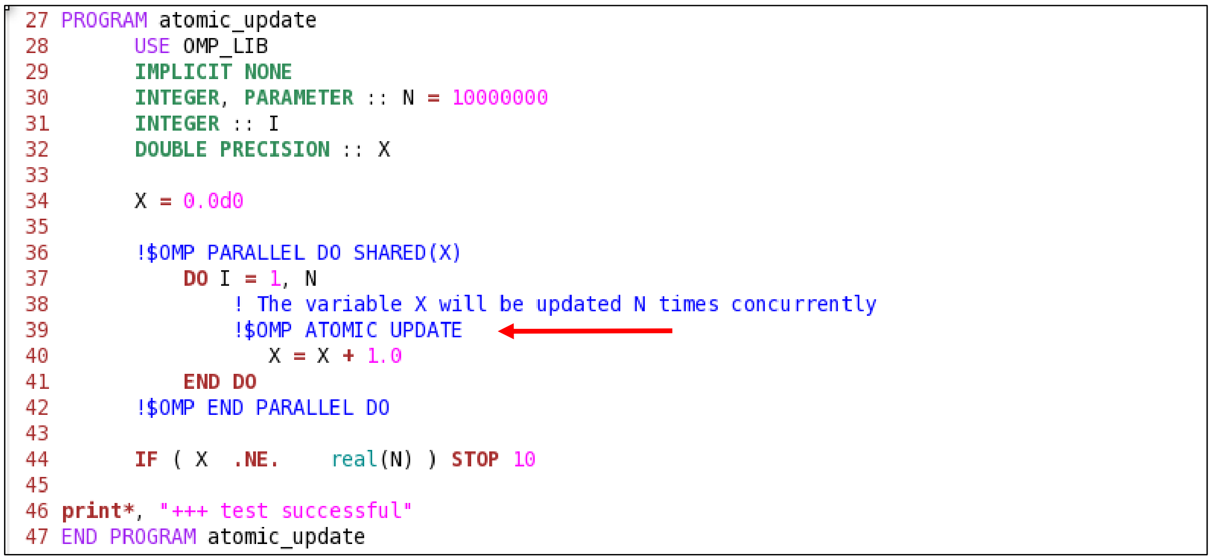
The atomic update is used to increment/decrement a variable atomically. The atomic update construct ensures that all accesses of X by any thread are performed atomically. It is ensuring that no race condition can occur when the variable X is updated N times within the loop.

View the source code of atomic_update.f95 (See figure 11)

Command:

```
vi atomic_update.f95
```

Figure 11 Program Source Code



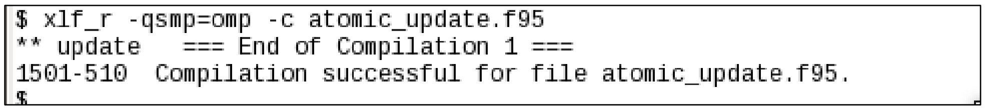
```
27 PROGRAM atomic_update
28       USE OMP_LIB
29       IMPLICIT NONE
30       INTEGER, PARAMETER :: N = 10000000
31       INTEGER :: I
32       DOUBLE PRECISION :: X
33
34       X = 0.0d0
35
36       !$OMP PARALLEL DO SHARED(X)
37           DO I = 1, N
38               ! The variable X will be updated N times concurrently
39               !$OMP ATOMIC UPDATE
40                  X = X + 1.0
41           END DO
42       !$OMP END PARALLEL DO
43
44       IF ( X  .NE.    real(N) ) STOP 10
45
46 print*, "+++ test successful"
47 END PROGRAM atomic_update
```

Compile atomic_update.f95 (See figure 12)

Command:

```
xlf_r -qsmp=omp -c atomic_update.f95
```

Figure 12 Compile



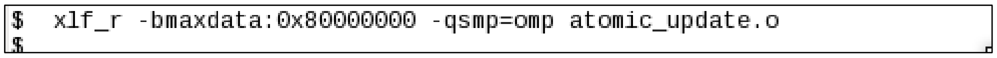
```
$ xlf_r -qsmp=omp -c atomic_update.f95
** update   === End of Compilation 1 ===
1501-510  Compilation successful for file atomic_update.f95.
$
```

Link atomic_update.f95 (See figure 13)

Command:

```
xlf_r -bmaxdata:0x80000000 -qsmp=omp atomic_update.o
```

Figure 13 Compile



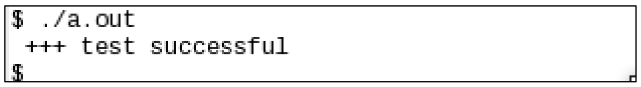
```
$  xlf_r -bmaxdata:0x80000000 -qsmp=omp atomic_update.o
$
```

Run atomic_update.f95 (See Figure 14).

Command:

```
./a.out
```

Figure 14 Run



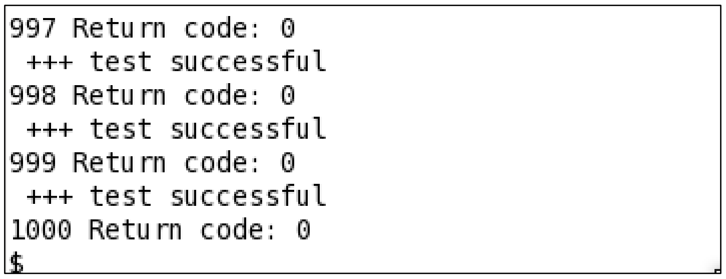
```
$ ./a.out
 +++ test successful
$
```

Even when running the program a very large number of times, the program terminates successfully and the race condition is avoided.

Command:

```
./run_multi
```

Figure 15 Multiple Runs



```
997 Return code: 0
 +++ test successful
998 Return code: 0
 +++ test successful
999 Return code: 0
 +++ test successful
1000 Return code: 0
$
```

In the first two samples we have demonstrated how the use of the atomic update directive can ensure a thread safe program when multiple threads are updating the same memory location concurrently. The OpenMP atomic update directive specifies that a memory location must be updated atomically. The update consists of reading the value stored in the memory location and writing it back atomically. It is important to note that the atomic update prevents multiple threads from writing to the same memory location at the same time. The same goal can be achieved by using a critical section but it can be significantly slower.

This concludes the tutorial segment for using OMP3.1 atomic update directive for Fortran. The next section will focus on the use of the atomic read and the atomic write directives.

Part 2: Using OMP3.1 atomic read and write

Steps:

In the next demo sample, c1 is an array of size N. The loop within the parallel region updates only the indices 1 to M of c1 (M < N). The way the loop is set-up, multiple threads can be simultaneously updating the same element of c1. Moreover c1 elements are of double complex type which uses a 16-byte representation in memory and requires two instructions to be written or read. As a consequence, one thread can write to part of an element while another thread is writing to the other part. The variable c1 can then have intermediate (wrong) results and the program might have an undefined behavior.

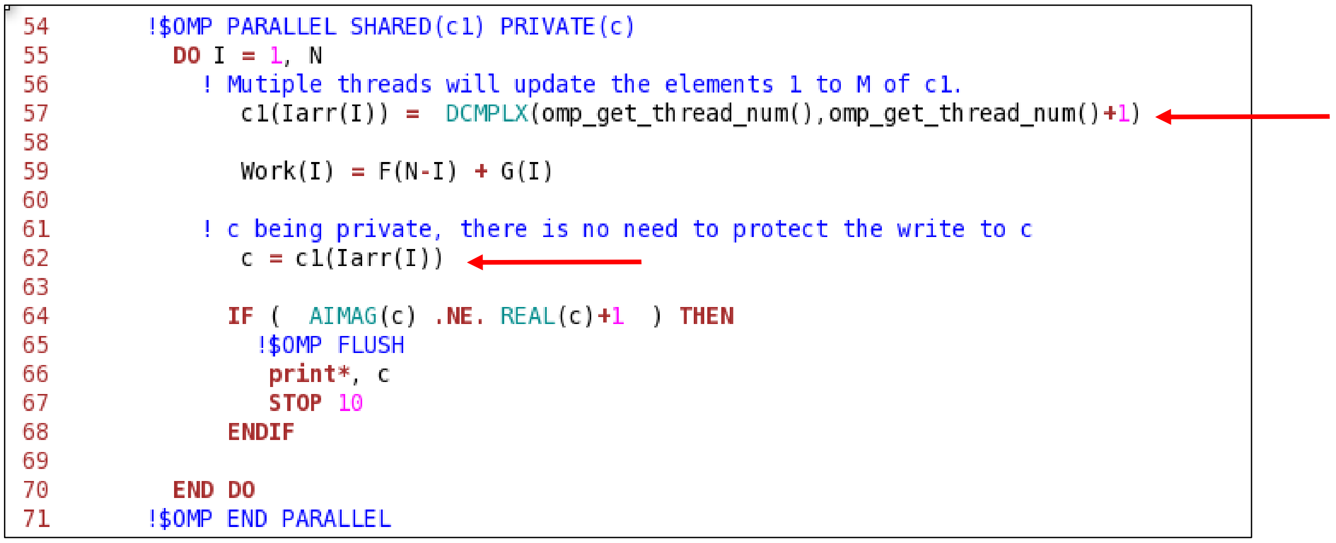
Again, to detect the race condition more easily, we will use a large number of threads: 128. The resource contention will result in the program running slowly but this demo's focus is not on performance.

3. Compile and run read_write.f95 program.

   Command:

```
vi read_write.f95
```

Figure 16 Program source code



```
!$OMP PARALLEL SHARED(c1) PRIVATE(c)
  DO I = 1, N
    ! Mutiple threads will update the elements 1 to M of c1.
       c1(Iarr(I)) =  DCMPLX(omp_get_thread_num(),omp_get_thread_num()+1)

       Work(I) = F(N-I) + G(I)

    ! c being private, there is no need to protect the write to c
       c = c1(Iarr(I))

      IF (  AIMAG(c) .NE. REAL(c)+1  ) THEN
        !$OMP FLUSH
         print*, c
         STOP 10
      ENDIF

  END DO
!$OMP END PARALLEL
```

The value assigned to the first M elements of c1 depends on the number id of the thread performing the work. The real part will be set to the thread number and the complex part to the thread number + 1. The thread number can be obtained with the omp_get_thread_num routine. Within the same parallel region, the value of each of the first M elements of c1 is read and then copied into c.

The first red arrow points at the statement where the values of the elements of the array c1 are being written to. At the same time, these values are read and stored in c (second red arrow). The elements of c1 being of type double complex, each read (load) and write (store) requires 2 hardware instructions. While one thread is reading one element nothing is preventing another thread from updating part of the same element. The write to c is thread safe because each thread is writing to its own private copy.

At line 67 of the program, we verify that c has an expected value by comparing its real and imaginary parts.

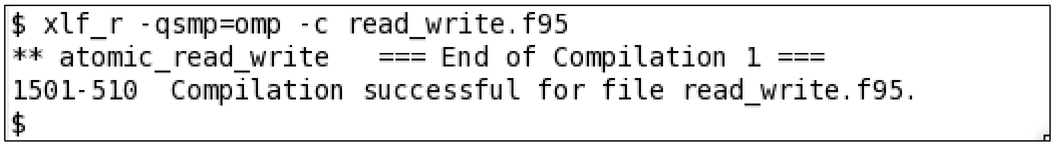
Now let's run the program and see what happens.

Compile read_write.f95 (See figure 17)

Command:

```
xlf_r -qsmp=omp -c read_write.f95
```

Figure 17 Compile



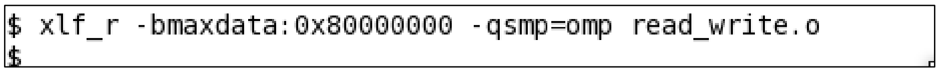
```
$ xlf_r -qsmp=omp -c read_write.f95
** atomic_read_write   === End of Compilation 1 ===
1501-510  Compilation successful for file read_write.f95.
$
```

Link read_write.f95 (See figure 18)

Command:

```
xlf_r -bmaxdata:0x80000000 -qsmp=omp read_write.o
```

Figure 18 Compile



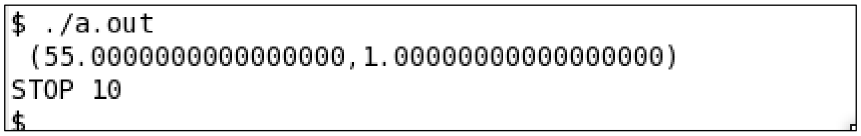
```
$ xlf_r -bmaxdata:0x80000000 -qsmp=omp read_write.o
$
```

Run read_write.f95 (See Figure 19).

Command:

```
./a.out
```

Figure 19 Run



```
$ ./a.out
 (55.0000000000000000,1.00000000000000000)
STOP 10
$
```

Again, you may need multiple executions to expose the race condition. You can use the provided script: run_multi

As you can see the variable “c” can have the wrong value. A program relying on the values of c or c1 exhibit undefined behavior.

4. Compile and run atomic_read_write.f

   The following lines were added to read_write.f to create atomic_read_write.f ensuring that the read and write of the first M elements of c1 are performed atomically and are mutually exclusive.

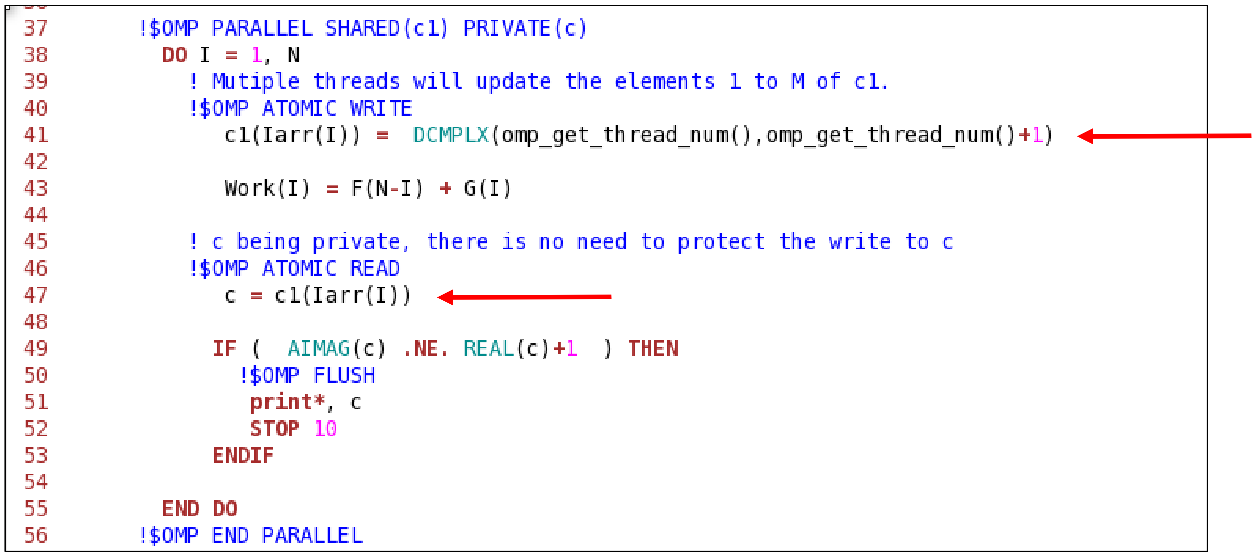
   Program atomic_read_write.f

```
 ...
   40        !$OMP ATOMIC WRITE
 …
   46        !$OMP ATOMIC READ
 …
```

   View the source code of atomic_read_write.f95 (see figure 20)
   Command:

   vi atomic_read_write.f95

   Figure 20 Program source code



```
      !$OMP PARALLEL SHARED(c1) PRIVATE(c)
        DO I = 1, N
          ! Mutiple threads will update the elements 1 to M of c1.
          !$OMP ATOMIC WRITE
             c1(Iarr(I)) =  DCMPLX(omp_get_thread_num(),omp_get_thread_num()+1)

             Work(I) = F(N-I) + G(I)

          ! c being private, there is no need to protect the write to c
          !$OMP ATOMIC READ
             c = c1(Iarr(I))

            IF (  AIMAG(c) .NE. REAL(c)+1  ) THEN
              !$OMP FLUSH
               print*, c
               STOP 10
            ENDIF

        END DO
      !$OMP END PARALLEL
```

The atomic write is used to ensure that all writes to a target memory location are performed atomically. The evaluation of the right hand expression is not done atomically but the write to the target memory location (the elements of c1) is executed atomically.

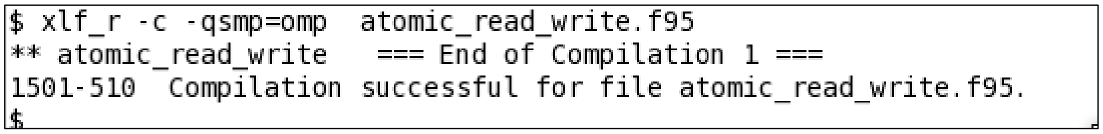
The atomic read is used to ensure that all reads of a target memory location are performed atomically. The read of the value stored in the target memory location (the elements of c1) is performed atomically but the write (into c) is not performed atomically. This is not an issue as c is private to each thread.

Compile atomic_read_write.f95 (See figure 21)

Command:

```
xlf_r -qsmp=omp -c  atomic_read_write.f95
```

Figure 21 Compile



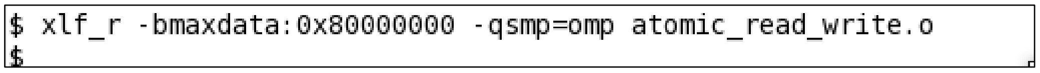
```
$ xlf_r -c -qsmp=omp  atomic_read_write.f95
** atomic_read_write   === End of Compilation 1 ===
1501-510  Compilation successful for file atomic_read_write.f95.
$
```

Link atomic_read_write.f95 (See figure 22)

Command:

```
xlf_r -bmaxdata:0x80000000 -qsmp=omp atomic_read_write.o
```

Figure 22 Link



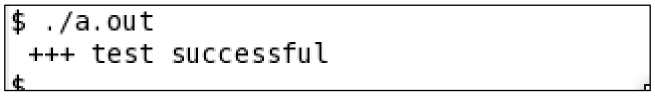
```
$ xlf_r -bmaxdata:0x80000000 -qsmp=omp atomic_read_write.o
$
```

Run atomic_read_write.f (See Figure 23).

Command:

```
./a.out
```

Figure 23 Run



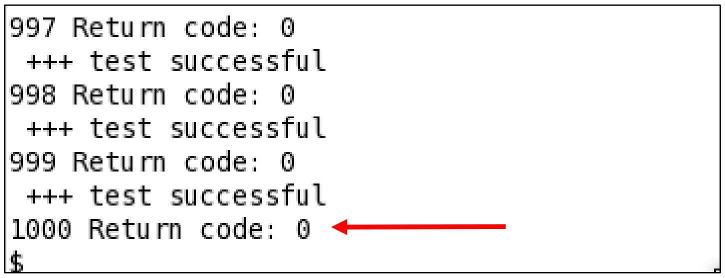
```
$ ./a.out
 +++ test successful
$
```

Even when running the program a very large number of times, the program terminates successfully and the race condition is avoided.

Command:

```
./run_multi
```

Figure 24 Multiple Runs



```
997 Return code: 0
 +++ test successful
998 Return code: 0
 +++ test successful
999 Return code: 0
 +++ test successful
1000 Return code: 0
$
```

In these two samples, we have demonstrated how the use of the atomic read and atomic write directives can ensure that a variable is being read/written by a thread as a whole without any other thread interfering. This is especially important to guarantee atomicity of reads and writes of variables larger than the size of the native machine word.

This concludes the tutorial segment for using the atomic read and the atomic write directives.

Part 3: Using OMP3.1 atomic capture

Steps:

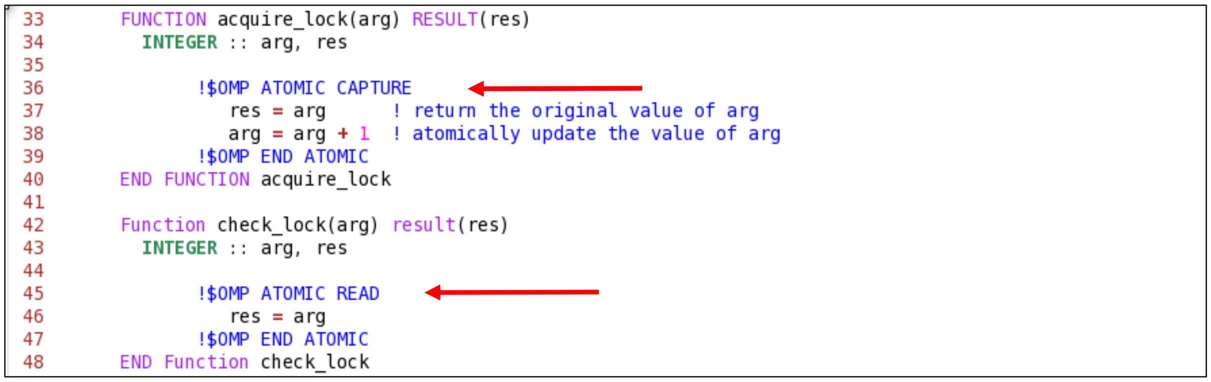
In the last demo sample, we will use the capture clause of the atomic directive to implement mutual exclusion. The atomic capture is used to acquire the lock (in the function acquire_lock) and the atomic read is used to verify that the lock is free (in the check_lock function).

Within the parallel region, once a thread acquires the lock, it can safely perform any work on a shared variable without another thread updating the whole variable or part of it.

In the sample, we use the implemented locks to perform a reduction operation on the variable “sum”. Obviously, this is not the best way to perform a reduction in OpenMP. It is only intended for the purpose of illustrating the use of the atomic capture directive. Later in the same sample we use the reduction directive to compute the expected value and test the robustness of our locking mechanism.

5. Compile and run mutex.f95.

Figure 25 Program source code



```
      FUNCTION acquire_lock(arg) RESULT(res)
        INTEGER :: arg, res

             !$OMP ATOMIC CAPTURE
                res = arg       ! return the original value of arg
                arg = arg + 1   ! atomically update the value of arg
             !$OMP END ATOMIC
      END FUNCTION acquire_lock

      Function check_lock(arg) result(res)
        INTEGER :: arg, res

             !$OMP ATOMIC READ
                res = arg
             !$OMP END ATOMIC
      END Function check_lock
```

The argument passed to check_lock will be read atomically. The argument to acquire_lock is first captured and then it is incremented. Both operations occur atomically. acquire_lock returns the old value of the effective argument.

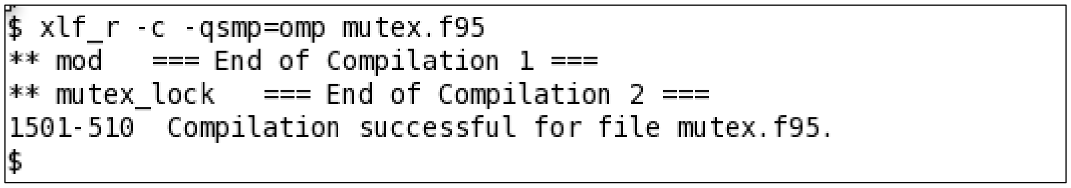
The lock is acquired when the variables lock and test_lock have different values.

Compile mutex.f95 (See figure 26)

Command:

```
xlf_r -qsmp=omp -c mutex.f95
```

Figure 26 Compile



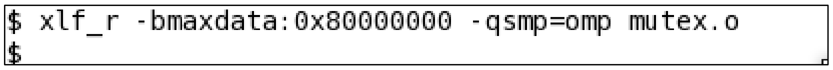
```
$ xlf_r -c -qsmp=omp mutex.f95
** mod    === End of Compilation 1 ===
** mutex_lock    === End of Compilation 2 ===
1501-510  Compilation successful for file mutex.f95.
$
```

Link mutex.f95 (See figure 27)

Command:

```
xlf_r -bmaxdata:0x80000000 -qsmp=omp mutex.o
```

Figure 27 Link



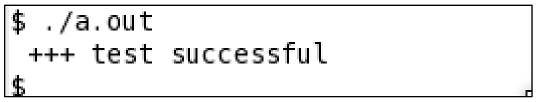
```
$ xlf_r -bmaxdata:0x80000000 -qsmp=omp mutex.o
$
```

Run mutex.f95 (See Figure 28).

Command:

```
./a.out
```

Figure 28 Run

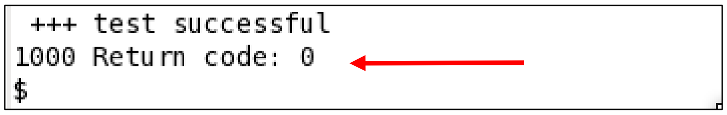


```
$ ./a.out
 +++ test successful
$
```

The order in which the threads will acquire the lock, perform the work and release the lock is unpredictable. The most important aspect of a locking system is that it ensures that there is no interleaving of threads – no thread should acquire the lock before another thread has released it.

The first test in the sample relies on the fact that we use 128 threads. We verify that when exiting the parallel region, the variable I was incremented 128 times. A second test verifies the computed sum. In the second parallel region, we use the OpenMP reduction clause to compute the same sum using the same number of threads as in our reduction (using the locks). Both sums should match.

The test is successful when both of the above tests are successful. Running the program multiple time should always yield a successful test.



```
 +++ test successful
1000 Return code: 0
$
```

In the last sample we saw how to implement mutual exclusion using 2 atomic directives in conjunction. It illustrates the most common use of the atomic directives.

This concludes the tutorial segment for using the atomic constructs.

What you have learned

In this exercise you learned how to:

- Use the IBM XL Fortran for AIX compiler to compile programs.
- Use atomic update, read, write and capture.

Conclusion

This concludes the tutorial on using the IBM XL Fortran compiler. This tutorial has shown how to use the update, read, write and capture clauses of the atomic construct.

Trademarks

Resources

Learn

Overview of the IBM XL C/C++ and XL Fortran Compiler Family
http://www.ibm.com/support/docview.wss?uid=swg27005175

Getting Started with XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024216

Compiler Reference - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024215

Language Reference - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024218

Optimization:

Optimization and Programming Guide - XL Fortran for AIX, V14.1
http://www.ibm.com/support/docview.wss?uid=swg27024219

Code Optimization with the IBM XL Compilers
http://www.ibm.com/support/docview.wss?uid=swg27005174