IBM XL C/C++ Compiler for AIX

Exploring new OMP3.1 constructs with the IBM XL C/C++ Compiler for AIX

By: Salma Elshatanoufy,
Lu Yang

Level: Intermediate

May 2012

Contents

Before you start

About this series

Walk through this scenario and others online as part of the IBM XL C/C++ Compiler for AIX.

About this Tutorial

This demo explains how different OMP3.1 atomic constructs are used in multi-threaded programs. User scenarios that utilize different XLC/C++ Compiler OMP3.1 atomic pragmas show how atomic read, write and capture constructs are used to control access to shared data regions in multi-threaded environments. In addition, a user scenario that utilizes task final demonstrates the usage of the final clause to control task switching between threads. Finally, a user scenario demonstrating min/max reduction demonstrates newly added reduction functionality in OMP3.1.

Objectives

- Using IBM XL C/C++ Compiler for AIX to exploit OMP3.1 atomic read, write and capture
- Using IBM XL C/C++ Compiler for AIX to exploit OMP3.1 task final
- Using IBM XL C/C++ Compiler for AIX to exploit OMP3.1 min/max reduction

- Total time: 45 minutes

Prerequisites

- Basic Unix skills
- Basic command line compilation experience
- Background in OMP3.0 parallel constructs
- IBM XL C/C++ Compiler trial/ GA version installed

System Requirements

http://www.ibm.com/software/awdtools/xlcpp/aix/sysreq

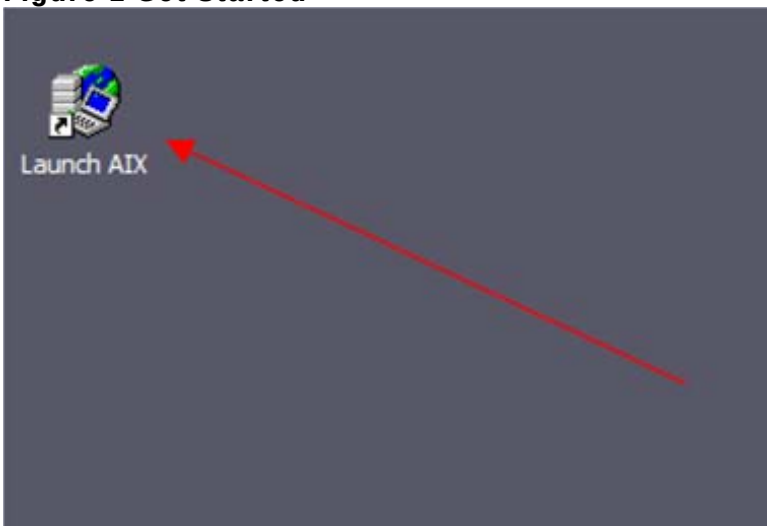
Glossary

IBM XL C/C++ Compiler: IBM® XL C and C++ compilers offer advanced compiler and optimization technologies and are built on a common code base for easier porting of your applications between platforms. They comply with the latest C/C++ international standards and industry specifications and support 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 OpenMP3.1

Successful login will result with user presented with a menu of demo hosted on the server. Type 17 and press Enter to select the “C/C++ Exploring new OMP3.1 constructs “ demo.

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 (See Figure 2 Demo Prepared). 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. Your home directory contains necessary source code to perform the tutorial. Type ls command to see the directory content (See Figure 3 Contents).

Command:

```
ls
```

Figure 3 Contents

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

       Thank you for waiting, setup is now ready

       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.

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


$ bash
bash-3.2$ ls
minmax.c      read.cpp      task.cpp      update_1.cpp  update_2.cpp  write.cpp
bash-3.2$
```

Setup needed environment variables

Figure 4 export OMP_NUM_THREADS environment variable

In your terminal type 'export OMP_NUM_THREADS=16'

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 16.

Figure 5 Export Environment Variables

```
bash-4.1$ export OMP_NUM_THREADS=16
bash-4.1$
```

Get started with the multi-threaded scenarios

Part 1: Using OMP3.1 atomic capture

Steps:

The atomic update is used to increment/decrement a variable atomically. The program shown in Figure 7 demonstrates how an atomic update is used in a parallel for loop to control access to the shared variable 'index'. The update pragma ensures that all accesses made to 'index' by any thread are done atomically. Without this control it is possible for race conditions to occur whenever 'index' is updated.

1. Create a new working directory call it update_1 and cd into that directory

Figure 6 Create the working directory update_1

```
bash-4.1$ mkdir update_1
bash-4.1$ cd update_1/
bash-4.1$
```

2. Save the source code of update_1.cpp locally in your home directory, update_1

Figure 7 Program Source Code
update_1.cpp:

```
/*Licensed Materials - Property of IBM
update_1.cpp
Copyright IBM Corp. 2012.
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.
This sample source code file ("update_1.cpp") is owned by International Business
Machines Corporation or one of its subsidiaries ("IBM") and is copyrighted
and licensed, not sold. You may use this Sample for  internal use only and
and only for the purpose of testing and evaluating IBM Rational products.
You may not alter or delete any copyright information or notices contained
in this Sample.  IBM provides this Sample without obligation of support and
"AS IS", WITH NO WARRANTY OF ANY KIND, EITHER EXPRESS OR
IMPLIED, INCLUDING THE WARRANTY OF TITLE, NON-INFRINGEMENT
OR NON-INTERFERENCE AND THE IMPLIED WARRANTIES AND
CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.
*/
#include <stdio.h>
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
        char a;
        char index= 'a';
        char verify='v';
        int i;
        int rc = 55;
        int num_threads=0;
        int thread_id=0;
        int thread_sum=0;

        #pragma omp parallel for  shared (index)
                for (i=0; i<21; i++)
                {
                        thread_id = omp_get_thread_num();
                        num_threads = omp_get_num_threads();
                        #pragma omp atomic update
                                index++;

                }



        if (index != verify) rc++;
        cout<<"index is "<<index<<" verify is "<<verify<<endl;

return rc;
}
~
```

Note: 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. Your home directory contains necessary source code to perform the tutorial.

3. Compile the source to produce one executable update_1

Command:
`xlC_r -qsmp=omp -o update_1 update_1.cpp`

Figure 8 build the executable

```
bash-4.1$ xlC_r -qsmp=omp -o update_1 update_1.cpp
```

4. List the files in update_1 dir, you should be able to see the update_1 executable

Figure 9 list compiled files 1

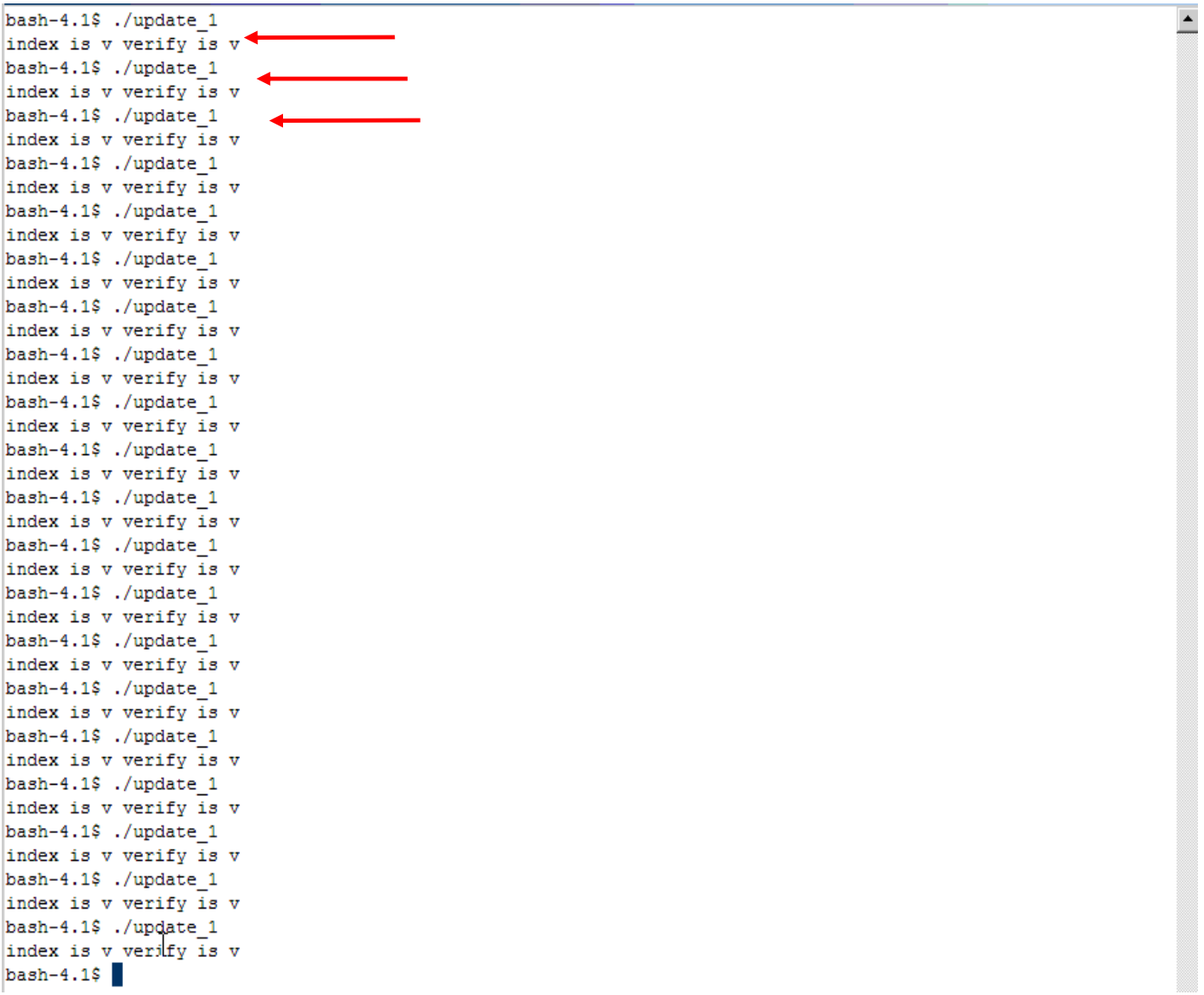
```
bash-4.1$ ls
update_1      update_1.cpp
bash-4.1$
```

5. Run the program repeatedly and observe the result

Command:

```
./update_1
```

Figure 10 Run 1



The values printed for ‘index’ and ‘verify’ are the same, character ‘v’.

6. Change to your home directory and create a new working directory, update_2

Command:

```
cd ..
mkdir update_2
cd update_2
```

Figure 6 Change Dir 1

```
bash-4.1$ mkdir update_2
bash-4.1$ cd update_2/
bash-4.1$
```

7. Save the source code of update_2.cpp locally in your home directory, update_2

Figure 7 Program source code
update_2.cpp:

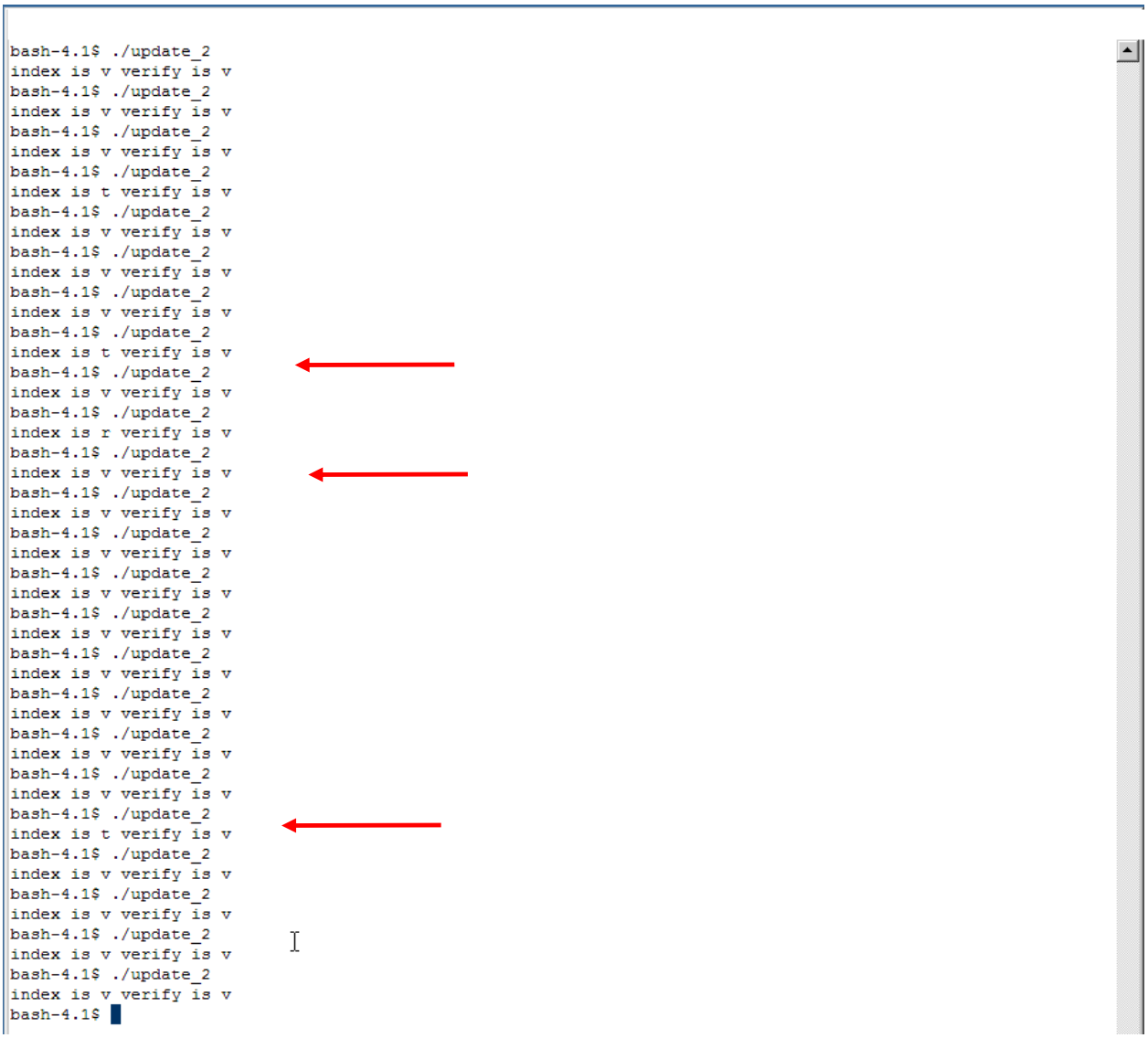
```
/*Licensed Materials - Property of IBM
update_2.cpp
Copyright IBM Corp. 2012.
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.
This sample source code file ("update_2.cpp") is owned by International Business
Machines Corporation or one of its subsidiaries ("IBM") and is copyrighted
and licensed, not sold. You may use this Sample for  internal use only and
and only for the purpose of testing and evaluating IBM Rational products.
You may not alter or delete any copyright information or notices contained
in this Sample.  IBM provides this Sample without obligation of support and
"AS IS", WITH NO WARRANTY OF ANY KIND, EITHER EXPRESS OR
IMPLIED, INCLUDING THE WARRANTY OF TITLE, NON-INFRINGEMENT
OR NON-INTERFERENCE AND THE IMPLIED WARRANTIES AND
CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.
*/
#include <stdio.h>
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
        char a;
        char index= 'a';
        char verify='v';
        int i;
        int rc = 55;
        int num_threads=0;
        int thread_id=0;
        int thread_sum=0;

        #pragma omp parallel for  shared (index)
                for (i=0; i<21; i++)
                {
                        thread_id = omp_get_thread_num();
                        num_threads = omp_get_num_threads();
                                index++;

                }


        if (index != verify) rc++;
        cout<<"index is "<<index<<" verify is "<<verify<<endl;

return rc;
}
~
```

Note: Notice how update_2.cpp is identical to update_1.cpp, except the atomic update pragma is removed inside the parallel for construct.

8. Repeat steps 3 through 5, building executable update_2 and running it multiple times

9. Notice the difference in the output of the program each time it is run. This should demonstrate the benefit of the locking mechanism created by the atomic update.

Figure 8 Run



Notice how the value of index printed as t instead of the correct expected value ‘v’.This concludes this tutorial segment for using OMP3.1 atomic capture for C/C++.

Part 2: Using OMP3.1 atomic write

Steps:

The atomic write is used to atomically update a target memory location with an rvalue expression. The evaluation of the right hand expression itself is not evaluated atomically. The write to the target memory location (lvalue) on the other hand is executed atomically. When used in conjunction with atomic read, users can essentially create their own locking mechanisms such as mutex locks and semaphores.

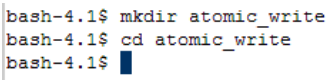
In the following program, the array ‘a’ is written to by multiple threads. Since each element holds a long long which is bigger than the native machine word size in 32-bit, atomic write is needed to ensure the stores to the target element proceed without interruptions from the

other threads. The expression ‘id>>32+id’ is not evaluated atomically. On the other hand, the expression ‘a[i]’ is in fact evaluated atomically, and the write to the destination proceeds with no interference from other threads.

1. Change to your home directory and create a new working directory, atomic_write.

   Command:
   ```
   mkdir atomic_write
   cd    atomic_write
   ```

Figure 9 Change and create new working directory



2. Save the source of write.cpp locally to your working directory, atomic_write.

Figure 10 Program Source Code
write.cpp:

```
Copyright IBM Corp. 2012.
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.
This sample source code file ("write.cpp") is owned by International Business
Machines Corporation or one of its subsidiaries ("IBM") and is copyrighted
and licensed, not sold. You may use this Sample for  internal use only and
and only for the purpose of testing and evaluating IBM Rational products.
You may not alter or delete any copyright information or notices contained
in this Sample.  IBM provides this Sample without obligation of support and
"AS IS", WITH NO WARRANTY OF ANY KIND, EITHER EXPRESS OR
IMPLIED, INCLUDING THE WARRANTY OF TITLE, NON-INFRINGEMENT
OR NON-INTERFERENCE AND THE IMPLIED WARRANTIES AND
CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.
*/
#include <stdio.h>
#include <omp.h>
#define Limit 16
#include <iostream>
using namespace std;
int main ()
{
        long long a [Limit];
        int i=1;
        int rc = 55;
        int temp[Limit];

        for (i=0; i<Limit; i++)
        {
                a[i]= 0;
                temp[i]=0;
        }


       #pragma omp parallel shared (a)
       {
                long long temp_val=0;
                int upper_half=0;
                int lower_half=0;
                long long id = omp_get_thread_num();

                for(int i=0 ;i<Limit;i++)
                {
                        #pragma omp atomic write
                        a[i]=(id<<32)+id;

                        temp[i]*=(temp[i]+id)-i;

                        #pragma omp atomic read
                        temp_val=a[i];

                        lower_half= temp_val & 0x000000000000FFFF;
                        upper_half= temp_val >>32;

                        if (upper_half != lower_half)
                        {
                                cout<<"upper half is "<<upper_half<<" lower half is "<<lower_half<<endl;
                                rc++;
                        }

                }


        }  /*end of parallel region*/


        return rc;
  }
```

The terminal window is now ready for commands. Your home directory contains necessary source code to perform the tutorial.

3. Compile the source to produce one executable, write

Command:
```
xlC_r –qsmp=omp –o write write.cpp
```

Figure 11 build the executable

4. List the files in write dir, you should be able to see the write executable

Figure 12 list compiled files 1

5. Run the program repeatedly and observe the result

Command:
```
for ((i=0;i<1000;i++)); do echo $i; ./write; if [[ $? -ne 55 ]]; then break;
fi ; done;
```
Here is the sample command to run the program 1000 consecutive times:

Figure 13 Run 1000 repeats

As long as the upper half of each element of a[i] is equal to the word in the lower half, the if statement check is not triggered, and hence nothing is printed from the program. As an exercise to the user, remove the atomic write pragma, re-compile the code and run repeatedly(at least 1000 consecutive times). The user should eventually see output from the program where the upper and lower words of the same element are unequal.

```
bash-4.1$ for ((i=0;i<1000;i++)); do echo $i; ./write; if [[ $? -ne 55 ]]; then break; fi ; done;
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
```

This concludes this tutorial segment for using OMP3.1 atomic write for C/C++.

Part 3: Using OMP3.1 atomic read

Steps:

The atomic read is used to read a value from a target memory location atomically. It prohibits other threads from updating the target memory location while the read is in progress. As mentioned, an atomic read is best used in conjunction with an atomic capture or an atomic write when re-creating a program specific locking mechanism.

In the following program, the atomic read is used to “peek” into the locking variable ‘semaphore’. If it is nonzero, then another thread currently owns the lock. If it is zero, then the thread goes ahead and obtains the lock through an atomic capture.

1. Change to your home directory and create a new working directory, atomic_read.

   Command:
   ```
   mkdir atomic_read
   cd    atomic_read
   ```

Figure 14 Change and create new working directory

2. Save the source of read.cpp locally to your working directory, atomic_read.

Figure 15 Program Source Code

```
        for (i=0; i<Limit; i++)
        {
                a[i]= 0;
                verify[i] = 1;
        }

omp_set_dynamic(0);
omp_set_num_threads(16);

i=0;

#pragma omp parallel
  {
        check_lock();
        a[i]=a[i]+1;
        printf("a is %d i is %d\n",a[i],i);
        i=i+1;
        release_lock();
}


        for (i=1; i<Limit; i++)
                if (a[i]!=1)    rc++;

        return rc;
}

int check_lock()
{
        int lock=0;
        /*most of the threads will get stuck here while the semaphore is locked*/

        while(1)
        {
                do
                {
                #pragma omp atomic read
                lock = semaphore;
                }
                /* while (lock == 1); here, "lock == 1" may result in infinite recursive calls */
                while (lock >= 1);

                /*once you kick out of the loop acquire the lock again to avoid race conditions*/
                #pragma omp atomic capture
                lock = ++semaphore;

                if (lock >1 || lock <1)
                {
                        #pragma omp atomic update
                        semaphore -= 1;
                        continue;
                }

                break;
        }
        int thread_id=omp_get_thread_num();
        printf("thread number %d just acquired the lock\n",thread_id);
        return semaphore;
}


int release_lock()
{
        int thread_id=omp_get_thread_num();
        printf("thread number %d just released the lock\n",thread_id);
        #pragma omp atomic update
        semaphore -= 1;

}

70
71
```

The terminal window is now ready for commands. Your home directory contains necessary source code to perform the tutorial.

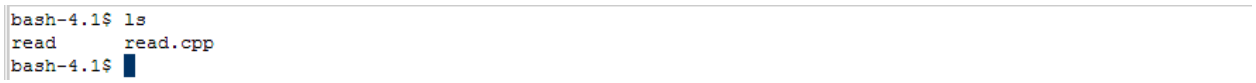
3. Compile the source to produce one executable read

   Command:
   `xlC_r –qsmp=omp –o read read.cpp`

Figure 16 build the executable

4. List the files in write dir, you should be able to see the read executable

Figure 17 list compiled files 1



5. Run the program repeatedly and observe the result

   Command:
   `./read`

Figure 18 Run 1

```
bash-4.1$ ./read
thread number 15 just acquired the lock
a is 1 i is 0
thread number 15 just released the lock
thread number 7 just acquired the lock
a is 1 i is 1
thread number 7 just released the lock
thread number 4 just acquired the lock
a is 1 i is 2
thread number 4 just released the lock
thread number 2 just acquired the lock
a is 1 i is 3
thread number 2 just released the lock
thread number 13 just acquired the lock
a is 1 i is 4
thread number 13 just released the lock
thread number 8 just acquired the lock
a is 1 i is 5
thread number 8 just released the lock
thread number 14 just acquired the lock
a is 1 i is 6
thread number 14 just released the lock
thread number 12 just acquired the lock
a is 1 i is 7
thread number 12 just released the lock
thread number 6 just acquired the lock
a is 1 i is 8
thread number 6 just released the lock
thread number 10 just acquired the lock
a is 1 i is 9
thread number 10 just released the lock
thread number 3 just acquired the lock
a is 1 i is 10
thread number 3 just released the lock
thread number 0 just acquired the lock
a is 1 i is 11
thread number 0 just released the lock
thread number 11 just acquired the lock
a is 1 i is 12
thread number 11 just released the lock
thread number 9 just acquired the lock
a is 1 i is 13
thread number 9 just released the lock
thread number 5 just acquired the lock
a is 1 i is 14
thread number 5 just released the lock
thread number 1 just acquired the lock
a is 1 i is 15
thread number 1 just released the lock
bash-4.1$
```

```
                        continue;
                }

                break;
        }
        int thread_id=omp_get_thread_num();
        printf("thread number %d just acquired the lock\n",thread_id);
        return semaphore;
}


int release_lock()
{
        int thread_id=omp_get_thread_num();
        printf("thread number %d just released the lock\n",thread_id);
        #pragma omp atomic update
        semaphore -= 1;

}

70
71
```

The values printed for each element of ‘a[i]’ correspond to 1. Since each element of the array is updated only once, by one of the threads from the parallel region. The print statement shows which thread id acquired the lock and displays when the lock is released. This concludes this tutorial segment for using OMP3.1 atomic capture for C/C++.

Part 4: Using OMP3.1 final tasks

Steps:

The final task is a construct introduced into OMP3.1 to decrease the overhead of task switching between threads. If the condition on the final clause is true, then the encountering thread executes the associated task region and nested task regions. No other thread from the same team is allowed to execute a task region that is final and owned by another thread.

In the following program, the final task conditional is set to true, ensuring that the nested task is executed by the same thread without being deferred.

1. Save the program task.cpp to your home directory.

Figure 19 Program source code
task.cpp

```
/*Licensed Materials - Property of IBM
task.cpp
Copyright IBM Corp. 2012.
US Government Users Restricted Rights - Use, duplication or disclosure
restricted by GSA ADP Schedule Contract with IBM Corp.
This sample source code file "task.cpp" is owned by International Business
Machines Corporation or one of its subsidiaries ("IBM") and is copyrighted
and licensed, not sold. You may use this Sample for  internal use only and
and only for the purpose of testing and evaluating IBM Rational products.
You may not alter or delete any copyright information or notices contained
in this Sample.  IBM provides this Sample without obligation of support and
"AS IS", WITH NO WARRANTY OF ANY KIND, EITHER EXPRESS OR
IMPLIED, INCLUDING THE WARRANTY OF TITLE, NON-INFRINGEMENT
OR NON-INTERFERENCE AND THE IMPLIED WARRANTIES AND
CONDITIONS OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.*/

#include <stdio.h>
#include <omp.h>
#define Limit 16

int main()
{
        int a=0;
        int rc = 55;
        int thread_id=0;
        int counter=0;

        #pragma omp parallel shared(a) firstprivate(thread_id)
        {

                thread_id=omp_get_thread_num();

                omp_set_num_threads(2);
                #pragma omp task final(thread_id == thread_id) //true condition, task is final
                {
                        thread_id=omp_get_thread_num();
                        int val=omp_get_max_threads();
                        #pragma omp task untied
                        {
                                int other_id=omp_get_thread_num();
                                #pragma omp atomic update
                                a++;

                                if (omp_in_final()!=1 || other_id != thread_id)
                                {
                                        #pragma omp atomic update
                                        counter++;
                                }
                        }

                }
                #pragma omp barrier
                #pragma omp master
                {
                        if (a != omp_get_num_threads()) //make sure final tasks are executed before other task
s
                        {
                                #pragma omp atomic update
                                counter++;
                        }
                }

        }


        if (counter !=0) rc++;

        return rc;
}
```

2. Compile the source code, producing the binary ‘task’

Figure 20 build the executable, ‘task’

```
bash-4.1$ xlC_r -qsmp=omp -o task task.cpp
```

3. Run the binary and observe the result. The omp_in_final runtime routine helps identify whether a task is final or not, in this case it is final, since it is nested inside a final task. Also, ‘other_id’ and ‘thread_id’ are the same, since the encountering thread for the final outer task is the same one that executes the inner nested tasks.

Figure 21 Run

```
bash-4.1$ ./task
bash-4.1$ echo $?
55
bash-4.1$
```

This concludes this tutorial segment for using OMP3.1 atomic capture for C/C++.

Part 5: Using OMP3.1 min/max reduction

Steps:

OpenMP has a concept of reduction to efficiently compute operations that can be done in parallel. OpenMP3.1 extended reductions to include the min/max operation in C/C++ to find the minimal and maximum value of a list.

The reduction clause specifies an operation to perform on all items in the list. Each thread performs the operation on a unique subset of the list and stores the result in a private copy. At the end of the reduction, a single thread performs the operation on all private copies to compute the final result.

The following program uses the min/max reduction to calculate the minmal and maximum value in an integer array.

1. Create a working directory call it minmax and cd into that directory

   Command:

   ```
   mkdir minmax
   cd minmax
   ls
   ```

Figure 22 Create the working directory min / max

```
bash-4.1$ mkdir minmax
bash-4.1$ cd minmax
```

2. Save the source code of minmax.c locally in your working directory minmax

Figure 23 Program source code
minmax.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MIN(num1, num2) (num1 < num2) ? num1:num2
#define MAX(num1, num2) (num1 > num2) ? num1:num2

#define N 20
#define NUM_THREAD 4

int main ()  {
  int i, a[N], XMin, XMax;

  unsigned int iseed = (unsigned int)time(NULL);
  srand (iseed);

  for (i=0; i<N; i++) {
    a[i] = rand();
    printf("a[%d] = %d\n", i, a[i]);
  }

  XMin = INT_MAX;
  XMax = INT_MIN;

  omp_set_num_threads(NUM_THREAD);

  #pragma omp parallel for private(i) reduction(min:XMin) reduction(max:XMax)
    for (i=0; i<N; i++) {
        fprintf(stdout, "Thread %d out of %d pool\n",omp_get_thread_num(),omp_g
et_num_threads());
        XMin = MIN(XMin, a[i]);
        XMax = MAX(XMax, a[i]);
    }

  printf("min: %d\n", XMin);
  printf("max: %d\n", XMax);

  return 0;
}
```

3. Compile the source code, producing the binary ‘minmax’

   Command:

   ```
   xlc_r -qsmp=omp -o minmax minmax.c
   ```

Figure 24 Build the executable, ‘minmax’

```
bash-4.1$ xlc_r -qsmp=omp -o minmax minmax.c
```

4. Run the binary and observe the result. The min/max reduction correctly calculated the minimal and maximum value in the array using 4 threads.

Command:

```
./minmax
```

Figure 25 Run

```
a[0] = 14758
a[1] = 25356
a[2] = 19629
a[3] = 23389
a[4] = 20398
a[5] = 16910
a[6] = 30841
a[7] = 1862
a[8] = 1187
a[9] = 968
a[10] = 30622
a[11] = 472
a[12] = 19383
a[13] = 8473
a[14] = 26242
a[15] = 13840
a[16] = 20258
a[17] = 16592
a[18] = 26727
a[19] = 28413
Thread 0 out of 4 pool
Thread 3 out of 4 pool
Thread 1 out of 4 pool
Thread 0 out of 4 pool
Thread 0 out of 4 pool
Thread 3 out of 4 pool
Thread 3 out of 4 pool
Thread 0 out of 4 pool
Thread 0 out of 4 pool
Thread 2 out of 4 pool
Thread 1 out of 4 pool
Thread 3 out of 4 pool
Thread 3 out of 4 pool
Thread 1 out of 4 pool
Thread 2 out of 4 pool
Thread 1 out of 4 pool
Thread 2 out of 4 pool
Thread 1 out of 4 pool
Thread 2 out of 4 pool
Thread 2 out of 4 pool
min: 472
max: 30841
```

You can modify the size of the array (N) and number of threads (NUM_THREADS), rebuild, rerun the executable and observe the results.

This works with xlC_r for C++ source code in exactly the same way.

This concludes this tutorial segment for using OMP3.1 min/max reduction for C/C++.

What you have learned

In this exercise you learned how to:

- Use IBM XL C/C++ compiler for AIX to build source code.
- Use OpenMP atomic mechanisms to control atomic access to special regions

Conclusion

This tutorial demonstrated how to utilize OpenMP 3.1 atomics to exploit synchronization mechanisms with the XL C/C++ AIX compilers. You have also learned how the final construct reduces the overhead of task switching between threads in a team. Demonstration also presents how you can use OpenMP atomics to create your own locking mechanisms. In addition, the new min/max reduction was demonstrated in a typical user scenario.

Trademarks

Resources

XL C/C++ for AIX Product Library
http://www.ibm.com/software/awdtools/xlcpp/aix/library/

Community Cafe
http://www.ibm.com/software/rational/cafe/community/ccpp