2006

Testing Programs That Contain OpenMP Directives

Bob Barnhart
Grand Valley State University

Testing Loops that Have Been Parallelized with OpenMP

A Masters Project for Grand Valley State University

Presented on December 14th, 2006

Student: Robert Barnhart

Advisor: Christian Trefftz

OpenMP is an API made up of compiler directives, run-time library routines, and environment variables that allows for parallel processing on shared memory computer systems. This brings about data dependencies that, if not caught and handled correctly, can cause faults in a program. Standard testing plans, both of the structural and functional type, are not suitable to be used for programs which make use of OpenMP because of the complications introduced by the use of multiple threads processing the same code at the same time. While much research has been done to help a programmer detect, classify, and remove these data dependencies, not much has been done to help test a program that has been parallelized. My goal for this project was to experiment with testing to see what might be the best way to discover improperly parallelized loops.

Background

The two most widely used testing methods - functional (black box) and structural (white box) testing do not do much good when it comes to OpenMP – in fact, they are worse than in serial programs. It is definitely possible for a perfectly valid set of functional test cases to be used on both a correct serial version and an incorrectly parallelized version of a program to produce good results, even though the parallel version is incorrect – this is because the data dependencies produced by parallelizing a program are not likely to become apparent every time the program is tested.

One method to test a parallelized program is to first un-parallellize the program by removing the OpenMP directives. Test the serialized program with whatever method is desired. This will show that the program is working correctly before parallelization is brought into the picture. After that, the tester can then focus on the loops that have been parallelized.

The reason that we have to be concerned with data dependencies in loops is because JC Huang's rule is no longer valid when it comes to parallel processing. Huang was a testing researcher who proved that every loop involves a decision, and in order to test the loop, we need only to test both outcomes of the decision: one outcome is to traverse the loop (only traversing it one time is sufficient), and the other is to exit (or not enter) the loop (Jorgensen 113-114). For problems introduced by a parallelized loop, however, Huang's idea does us no good. In fact, it would do even worse than that by giving us a false sense of security, the reason being that problems caused by parallelization will NOT surface until there are AT LEAST two iterations of the loop.

So now that we know that one traversal of a parallelized loop is not good enough, we need to decide how many traversals is enough to show that the loop is correct. This is the key question which I attempted to address with my project. Ideally, a tester needs to know how many times an incorrectly parallelized loop must be run before the programming error results in improper output. If this number can be found, then we will have an understanding of how much testing needs to be done on parallelized loops.

Test Strategy

The number of times required to test a parallelized loop is most likely based on a number of factors: the computer architecture involved, the number of processors being used, the number of iterations in the loop, and the amount of processing done in the loop. By controlling these variables and repeatedly executing parallelized loops, I was able to see how the different factors affected the number of tests required to discover problems.

The following steps comprised my test strategy:

1. Take different kinds of data dependencies.
2. Test each dependency in multiple environments. I had access to two shared memory environments. The first computer used in testing was my own laptop – an IBM Thinkpad with a 1.8 GHz processor, 512 MB of RAM, and Windows XP. I also had access to the 'winserv' computer at Grand Valley State University's campus. It had a 2.8 GHz processor, 6 GB of RAM, and Windows XP. I used Visual Studio 2005 on both machines to compile and execute the tests.
3. Vary the number of processors used, from two to ten, and all the way to 62 in one test.
4. Vary the number of iterations in the loop from 10 to 100 million.
5. For different variations of factors above, find out how many tests it will takes to uncover the error.
6. Run each test 50 times, and take the average – this average is what can be found in the table values of my test results

I used four different loops to test my strategy, all of which had obvious programming errors related to the parallelization process. By using three different loops, I was able to vary the type of data dependency, as well as the number of work being done within the loops.

Loop 1 (Reduction)

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
        x = x + 1;
```

This loop simply increments x by 1 with each iteration (this process is also known as reduction). Since each iteration of the loop reads and writes from x, there is a clear data dependence on x.

Loop 2 (Array Shift)

```
#pragma omp parallel for
for(int i = 0; i < (n-1);i++)
        a[i] = a[i+1];
```

This loop shifts the elements of an array. For each value of i, a[i] is read by iteration i, and written by iteration i-1.

Loop 3 (Reduction Plus)

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
        x = x + 1;
        for(int z = 0; z < 100 ; z++)
                d[z] = z;
}
```

This loop is similar to loop 1, but it adds a simple inner loop, which effectively adds 100 statements to the work being done in the parallelized loop.

Loop 4 (Array Shift Plus)

```
#pragma omp parallel for
for(int i = 0; i < (n-1);i++) {
        a[i] = a[i+1];
        for(int z = 0; z < 100 ; z++)
                d[z] = z;
}
```

This loop is similar to loop 2, with a simple inner loop to increase the amount of work being done within the parallelized loop.

Test Results

The following tables contain the results of my tests. The different rows show how the results changed with the size (number of iterations) of the loop. The columns show how the results changed with the number of threads that were used to process the loop. The actual table cell values show how many times (on average) the loop had to be executed under the given circumstances to result in an error. The tester wants this number to be small – the goal is to find the minimum number of times the loop must be executed in order to be certain that the loop is correct.

Loop 1 (Reduction) Test Results on IBM Thinkpad

Processors/ Iterations 2 4 6 8 10					
100	232391	47617	62195	45124	56529
1000	40507	2106	1554	855	854
10000	4351	469	85	77	62
100000	373	48	8	8	7
1000000	21	3	1	1	1
10000000	1	1	1	1	1
100000000	1	1	1	1	1

Loop 1 (Reduction) Test Results on Winserv

Processors/	**2 4**				

Iterations	**6 8 10**				
100	1283401	10	1	3	4
1000	27548	1	1	5	1
10000	1	1	1	1	1
100000	1	1	1	1	1
1000000	1	1	1	1	1
10000000	1	1	1	1	1
100000000	1	1	1	1	1

Loop 2 (Array Shift) Test Results on IBM Thinkpad

Processors/ Iterations	**2 4 6 8 10**				
10	8200	30	10	15	13
100	5794	16	18	10	11
1000	1764	18	20	20	6
10000	344	14	16	20	15
100000	34	9	8	5	6
1000000	1	2	3	3	3
10000000	1	1	1	1	1

Loop 2 (Array Shift) Test Results on Winserv

Processors/ Iterations	**2 4 6 8 10**				
10	37571	1	1	1	1
100	24247	1	1	1	1
1000	13113	1	2	2	2
10000	1	2	4	6	7
100000	1	3	5	7	8
1000000	1	3	5	7	8
1000000	1	3	5	7	9

0					

Loop 3 (Reduction Plus) Test Results on IBM Thinkpad

Processors/ Iterations	**2 4 6 8 10**				
100	907995	531101	177467	363371	235740
1000	29819	43268	20027	20352	25968
10000	3468	3189	2068	2193	4603
100000	1032	391	136	224	376
1000000	63	65	12	13	42
10000000	5	4	1	2	5
100000000	1	1	1	1	1

Loop 3 (Reduction Plus) Test Results on Winserv

Processors/ Iterations	**2 4 6 8 10**				
100	6	1	1	1	1
1000	1	1	1	1	1
10000	1	1	1	1	1
100000	1	1	1	1	1
1000000	1	1	1	1	1
10000000	1	1	1	1	1
100000000	1	1	1	1	1

Loop 4 (Array Shift Plus) Test Results on IBM Thinkpad

Processors/ Iterations	**2 4 6 8 10**				
10	1284	27	19	17	9
100	295	15	9	15	20
1000	28	8	7	7	7
10000	3	1	1	2	2
100000	1	1	1	1	1
1000000	1	3	5	7	8
10000000	1	3	5	7	9

Loop 4 (Array Shift Plus) Test Results on Winserv

Processors/ Iterations	**2 4 6 8 10**				
10	52214 1				

	1 1				
100	1 2 5 6 6				
1000	1 3 5 7 8				
10000	1 3 5 7 8				
100000	1 3 5 7 8				
1000000	1 3 5 7 8				
10000000	1 3 5 7 9				

Observations

There are several aspects of the test results which I believe would be useful to a tester. First, the computing environment significantly affected the amount of testing that must be done in order to reveal an improperly parallelized loop. In most situations, winserv took fewer tests to discover the problem than my laptop, and the difference was often quite large. The only real exception is Loop 2, but in that case the laptop was only slightly better in just a few situations. Of course, it is important to note that the environment is only important as a matter of efficiency and convenience. Using winserv makes it easier to find the errors, but it is still entirely possible to do so on my laptop – it just takes more iterations, or more tests.

Another observation from my experiments is that as you increase the size (in iterations) of a loop, it gets easier to uncover any problems. By the time you get to a million iterations, the number of tests required is very manageable. By 100 million

iterations, that number is down to 1 in most cases. This is a big improvement, especially considering that when the table shows that it took thousands or even hundreds of thousands of tests with 100 iterations, we are only talking about the AVERAGE number of tests. Sometimes it might be much less, but sometimes it could be much more. So a tester would not be able to rely on the large numbers to know how many times to run a test. But as the numbers get smaller, it is easier for the tester to know that if he runs the test the proper number of times, then it is sufficient.

A third significant observation from my test results is that the number of processors used to execute the loop also significantly affects the ability for a tester to uncover a problem when the loop has a small number of iterations. Increasing the number of processors from two to four has a major positive impact on testing; but using any more than six processors does not have any advantages; in fact, in some situations it even makes things worse. Also, as the number of iterations increases, the advantage of using more processors decreases. Again, Loop 2 on winserv is a slight exception to the rule, but it is very minor. If it takes one test with two processors, but three tests with four processors, there is not much of a difference there.

All of these observations are interesting, but they are not very significant if there is not a good way to apply them to test a program. However, I believe that they could be used as a guideline. For example, if the tester can control both the number of iterations and the number of processors used, the best situation, regardless of the environment, would be to use four processors, and to force the loop to iterate ten million times. Based on my experiments, this situation would most likely uncover an improperly parallelized loop. Running that same test multiple times would increase the tester's level of confidence. But the important thing is that the tester would know that it's not necessary to run the test a large number of times. Over all the four hundred times I ran this combination of processors and iterations across all of the examples, it took on average two tests to find the problem.

Sometimes, however, the tester can't control all of the variables in the testing environment. All that this would do would affect the number of tests that must be run. Say, for example, that due to some property of a loop, it can have at most one thousand iterations. To the tester, this would mean that a larger number of tests is required. Based on my results, the highest average with four processors and 1,000 loops was Loop 3 on my laptop, which required on average a little over 3,000 tests. Because this is only the average, running the test 3,000 times would not suffice. But running it 10,000 to 15,000 times would be enough to give the tester some confidence. A more interesting number in cases like this would be the maximum number of tests that it took.

Conclusion

While experimenting with OpenMP, I was able to test out 4 different loops on two different computers. This is by no means exhaustive, but I was able to observe several trends that were consistent across all of my tests. Based on my experimenting, I don't believe there is a "magic number" of tests required to reveal improperly parallelized loops. I do believe, however, that if a tester knows his computing environment, it is possible to determine a number of tests that would be required to be confident that the parallelization was done correctly. It seems that the key concept here is that of confidence. I don't believe that the approach that I have taken can be used to be 100

percent sure that nothing is wrong. But by using a reasonable number of tests, a good level of confidence can be reached. As the number of tests done is increased, regardless of the situation, so is the level of confidence.

Resources

Chandra, Rohit et al. *Parallel Programming in OpenMP.* San Francisco, CA: Morgan Kauffman Publishers, 2001

Jorgensen, Paul. *Software Testing: A Craftsman's Approach.* Boca Raton, FL: CRC Press, 1995