An OpenMP 3.1 Validation Testsuite

Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman

University of Houston,
Computer Science Dept, Houston, Texas
{cwang35,sunita,chapman}@cs.uh.edu
http://www2.cs.uh.edu/~hpctools

Abstract. Parallel programming models are evolving so rapidly that it needs to be ensured that OpenMP can be used easily to program multicore devices. There is also effort involved in getting OpenMP to be accepted as a de facto standard in the embedded system community. However, in order to ensure correctness of OpenMP's implementation, there is a requirement of an up-to-date validation suite. In this paper, we present a portable and robust validation testsuite execution environment to validate the OpenMP implementation in several compilers. We cover all the directives and clauses of OpenMP until the latest release, OpenMP Version 3.1. Our primary focus is to determine and evaluate the correctness of the OpenMP implementation in our research compiler, OpenUH and few others such as Intel, Sun/Oracle and GNU.

We also aim to find the ambiguities in the OpenMP specification and help refine the same with the validation suite. Furthermore, we also include deeper tests such as cross tests and orphan tests in the testsuite.

Keywords: OpenMP, validation suite, task constructs, tests.

1 Introduction

OpenMP [5] has become the *de facto* standard in shared-memory parallel programming for C/C++ and Fortran. Defined by compiler directives, library routines and environment variables, the OpenMP API is currently supported by a variety of compilers from open source community to vendors (for e.g. GNU [18], Open64 [1], Intel [11], IBM [8]). OpenMP ARB ratified the version OpenMP 3.0 in 2008 and 3.1 in 2011. The main difference between versions 2.5 (released in 2005) and versions 3.0/3.1 is the introduction of the concept of tasks and the task construct. The task-based programming model enables the developers to create explicit asynchronous units of work to be scheduled dynamically by the runtime. This model and its capabilities address the previous difficulties in parallelizing applications employing recursive algorithms or pointers based data structures [2]. OpenMP version 3.1 was a minor release that offered corrections to the version 3.0. The main purpose of OpenMP version 3.1 is to improve efficiency for fine grain parallelism for tasks by adding `final` and `mergeable` clauses along with other extensions such as `taskyield`.

B.M. Chapman et al. (Eds.): IWOMP 2012, LNCS 7312, pp. 237–249, 2012.

The goal of the work in this paper is to build an efficient framework, i.e. a testing environment, that will be used to validate the OpenMP implementations in OpenMP compilers. OpenMP is evolving with the increase in the number of users, as a result, there is an absolute need to check for completeness and correctness of the OpenMP implementations. We need to create an effective testing environment in order to achieve this goal. In prior work, we collaborated with colleagues at University of Stuttgart, to create validation methodologies for OpenMP versions 2.0 and 2.5 reported in [16,17] respectively. We have built our current framework on top of the older one. We have improved the testing environment and now the OpenMP validation testsuite covers all tests for the directives and clauses in OpenMP 3.1. This testing interface is portable, flexible and offers an user-friendly framework that can be tailored to accommodate specific testing requirements. Tests could be easily added/removed adhering to the changes in the OpenMP specification in future. In our current work we have ensured that the bugs in the previous validation testsuite have been fixed.

The organization of this paper is as follows: In Section 2 we describe the design and execution environment of the OpenMP validation suite. Section 3 shows the implementations and basic ideas for each of the tests. Here, we mainly focus on the concept of OpenMP tasks. In Section 4 we evaluate the validation suite using several open-source and vendor compilers. We discuss the related work in Section 5. Finally, we present the conclusion and the future work in Section 6.

2 The Design of an OpenMP Validation Suite

The basic idea in the design of the OpenMP validation suite is to provide short unit tests wherever possible and check if the directive being tested has been implemented correctly. For instance, the `parallel` construct and its corresponding clauses such as `shared` are tested for correctness. A test will fail if the corresponding feature has not been implemented correctly. We refer to such typical tests as *normal tests*.

Basically a number of values are calculated using the directive being tested and we compare the result with a known reference value. There is one type of failure called performance failure, in this case, even if the implementation of a directive is incorrect, it is not directly related to the correctness factor but it would just degrade the performance, for e.g. the `untied` clause in `task` construct. So it is at times not quite enough to only rely on the result calculated, but would require carefully written tests to check for the correctness of the implementation in a given compiler.

In a given code base, there might be more than a few directives being used at a given time. However, it is a challenge to check for correctness for a particular directive of interest, for instance `loop`, among several others. To solve this issue, we perform another test methodology called *cross test*, to validate only the directive under consideration. If this directive is removed from the code base, the output of the code will be incorrect.

Besides, we also need to ensure that the directive is serving its purpose. For instance lets consider a variable declared as `shared`. We also know that the

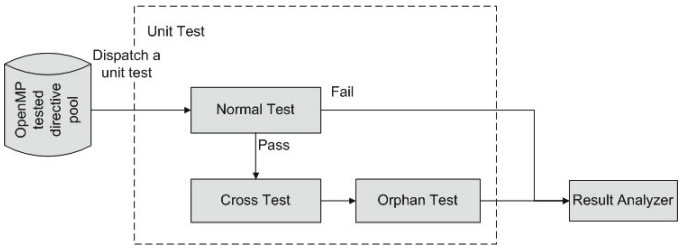

Fig. 1. The OpenMP Validation Suite Framework

variable is `shared` by default irrespective of explicitly declaring it as `shared`. Let us replace the `shared` with a `private` clause or any other clause which does not contain the functionality of the directive being tested, which in this case is `shared`. As a result, the *cross test* will check for the output result, which has to be incorrect because the variable is no longer being `shared`.

Moreover, in order to ensure that the directive being tested also gets executed correctly when "orphaned" from the main function, we creat a new test methodology named as *orphan test*. In the *orphan test*, the directive to be tested is placed into a children procedure which is called by the main function.

All test results will be statistically analyzed. Each test will be repeated multiple times. This is to ensure that the directive being tested functions correctly at all times. In order to estimate the probability that a test is passed accidentally we take the following approach: if n_f is the number of failed cross tests and M the total number of iterations, the probability of that test will fail is $p = \frac{n_f}{M}$. Thus the probability that an incorrect implementation passes the test is $p_a = (1-p)^M$, and the certainty of test is $p_c = 1 - p_a$, i.e. the probability that a directive is validated.

Currently the validation suite contains more than 70 unit tests covering all of the clauses in the OpenMP version 3.1 release. Each of the unit tests has three types of tests: *normal, cross*, and *orphan test*. One of the challenges is that if we implement each of these tests separately, the entire suite would be ad-hoc and error-prone. It would also be challenging to manage and analyze the results generated out of so many tests. So we created an execution environment that will manage these several tests methodically.

Figure 1 shows the proposed framework i.e. the execution environment of the OpenMP validation suite. In this framework, we create a test directive pool that will consist of templates for the unit tests for each of the OpenMP directives being tested. This framework has been developed mainly using the Perl scripting language. We use this framework to parse through the several templates that have been written for each of the OpenMP directives. Executing this framework will deliver the source code for the three types of tests, namely *normal*, *cross* and *orphan* tests. The *normal* tests will be the first test to be performed in this process. If this particular test fails then there is no need to perform the *cross* and

orphan tests. As a result, the corresponding source codes for *cross* and *orphan* tests will not be generated. This has been carefully crafted into our framework. If the *normal* tests pass successfully, the framework will automatically generate source codes for the other two tests. Note that we had to create only one template in order to generate source codes for all the three types of tests. As a result we emphasize that the framework adopts an automatic approach while creating the different kinds of tests necessary to check the correctness of the directives. There is very little manual labor involved in this process. Once these different tests have been created, our framework will compile and execute them as and when necessary. There is also a result analyzer component as part of the framework that will collect the results from each of the unit tests once all of them have completed execution. These results will be in the form of log files and the analyzer component will help in generating a complete report in a user-friendly manner.

The advantages that the execution environment offers are as follows:

- Creates one template for each test that is sufficient to automatically generate source codes for the three types of tests, i.e normal, cross and orphan tests.
- Creates bug reports that consist of adequate information about the compiler being used for testing purposes. The report will consist of version numbers of the compiler, build and configuration options, optimization flags used, and so on.
- Launches all the tests automatically, although individual tests will be performed only for those directives that are being tested.
- Generates reports that are easy to read and understand. These user-friendly reports will contain information about the bugs that have been identified. The details of the compilation and execution are also provided.

The framework is easy to use and maintain. It is quite flexible enough to accommodate changes as and when OpenMP specification gets updated with newer features.

3 Implementation

In this section, we discuss the unit tests for the OpenMP directives and clauses. We primarily discuss the unit tests for the directives and clauses in the latest versions of OpenMP (version 3.0 and 3.1). Previous publications [16,17] also discusses some of the ideas for the unit tests used to evaluate OpenMP Version 2.5 directives and clauses.

3.1 Directives and Clauses

`Task` is a new construct in OpenMP 3.0. It provides a mechanism to create explicit tasks. Tasks could be executed immediately or delayed by any assigned thread. Figure 2 shows the test for OpenMP `task` construct. The basic idea is to

generate a set of tasks by a single thread and execute them in a parallel region. The tasks should be executed on more than one threads. In the cross test, the `task` *pragma* is removed. As a result, every task is executed only by one thread since the tasks are in the `single` region hence delivering incorrect outputs.

```
int test_omp_task(){
 int i, result=0;
 int tids[NUM_TASKS];

 /*Initilization*/
 for (i=0; i<NUM_TASKS; i++)
 {
  tids[i]=0;
 }

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task
    {
     sleep (SLEEPTIME);
     tids[myi]=omp_get_thread_num();
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /*end of parallel */

 /*now check for results*/
 for (i = 1; i < NUM_TASKS; i++){
  if (tids[0] != tids[i])
   return (result = 1);
 }

 return result;
} /* end of test */
```

Fig. 2. Test for `task` construct

```
int test_omp_taskwait(){
 int i, result = 0;
 int array[NUM_TASKS];

 /*Initialization*/
 for(i=0;i<NUM_TASKS;i++)
 {
   array[i]=0;
 }

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task
     array[myi] = 1;
   } /* end of for */

   #pragma omp taskwait

   /*check for all tasks finish*/
   for (i = 0; i < NUM_TASKS; i++){
    if (array[i] == 0)
     result++;
   } /*end of for*/
  } /* end of single */
 } /*end of parallel */


 return (result == 0);
}/*end of test*/
```

Fig. 3. Test for `taskwait` construct

The `taskwait` construct specifies a synchronization point where the current task is suspended until all children tasks have completed. Figure 3 shows the code listing for testing the `taskwait` construct. A flag is set to each element of an array when a set of tasks are generated. If `taskwait` executes successfully, all elements in the array should be 1; otherwise, the elements should be zero. In the cross test, we remove the `taskwait` construct and check the value of elements in the array. Obviously, part of the values will be 0 while others will be 1 if there is no "barrier" at the completion of tasks. Consequently, it is able to validate the `taskwait` construct.

The `shared` clause defines a set of variables that could be shared by threads in `parallel` construct or shared by tasks in `task` construct. The basic idea to test this would be to update a shared variable i.e. say i by a set of tasks and

check whether it could be shared by all tasks. If this is the case, the value of the shared variable should be equal to number of tasks. In the cross test, we check if the result is wrong without the `shared` clause. As discussed in section 2, `shared` is replaced by the `firstprivate` clause, i.e., the attribute of i is changed to firstprivate. As a result, the value of i should be incorrect.

As opposed to `shared` clause, the `private` clause defines that variables are private to each task or thread. The idea of testing for the `private` clause is first to generate a set of tasks as before and each task to update a private variable, e.g., *local_sum*. We compare the value with the *known_sum* which is calculated in prior. In the cross test, we remove the `private` clause from `task` construct. Thus the private variable now becomes `shared` by default. As a result, we see that the value of `local_sum` should be incorrect.

The `firstprivate` clause is similar to `private` clause except that the new item list has been initialized prior to encountering the `task` construct. As a result, in contrast to `private` clause, we do not need to initialize variables declared as `firstprivate`. Consequently, test for the `firstprivate` is similar to the test for the `private` clause except that variable `local_sum` does not need to be initialized to zero in the `task` region. In the cross test, the `firstprivate` is removed and hence the variable `local_sum` becomes a `shared` variable once again.

The `default` clause determines the data-sharing attributes of variables implicitly. In C language, the variables declared as `default` is `shared`, while in Fortran from OpenMP 3.0, variables are declared as `private` or `firstprivate` by default. In addition, OpenMP 3.0 also allows variables to not have any predetermined data-sharing attribute declared as `none`. As a result, the idea of testing for `default` clause is actually the same as to testing for `shared` clause in C and `firstprivate, private` in Fortran.

The `if` clause controls the `task` implementation as shown in Figure 4. If the `if` is evaluated as false then the encountering task will be suspended and a new task is executed immediately. The suspended task will be resumed until the generated task is finished. The idea of testing the `if` clause is to generate a set tasks by a single thread and pause it immediately. The parent thread shall set a counter variable that the task which is paused, will consider when the thread wakes up. If the `if` clause is evaluated to false, the `task` region will be suspended and the counter variable *count* will be assigned to 1. When the `task` region resumes, we evaluate the value of the counter variable *count*. In the cross test, we removed `if` clause from the `task` construct, since `if` is evaluated to true by default, the `task` region will be executed immediately and the counter variable `count` will still be 0.

In OpenMP 3.0, task is executed by a thread of the team that generated it and is tied by default, i.e., tied tasks are executed by the same thread after the suspension. If it is the `untied` clause, any thread could resume the task after the suspension. The implementation of untied clause introduces newer kinds of compiler bugs and performance failures. But degradation in performance is unrelated to the implementation of the clause and its correctness. Testing such

```
int test_omp_task_if(){
 int count, result=0;
 int cond_false=0;

 #pragma omp parallel
 {
  #pragma omp single
  {
   #pragma omp task if (cond_false)
   {
    sleep (SLEEPTIME_LONG);
    result = (0 == count);
   } /* end of omp task */

   count = 1;
  } /* end of single */
 } /*end of parallel */

 return result;
} /*end of test*/
```

Fig. 4. Test for `if` clause

```
int omp_for_collapse(){
 int is_larger = 1;
 #pragma omp parallel
 {
  int i,j,my_islarger = 1;
  #pragma omp for schedule(static
       ,1) collapse(2) ordered
   for (i = 1; i < 100; i++)
    for (j =1; j <100; j++)
    {
     #pragma omp ordered
      my_islarger = my_islarger &&
        check_i_islarger(i);
    } /* end of for */
  #pragma omp critical
   is_larger=is_larger &&
        my_islarger;
 } /*end of parallel*/
 return (is_larger);
} /*end of test*/
```

Fig. 5. Test for `collapse` clause

features require the test codes to be very carefully created, since if the clause is not implemented correctly, it will not yield incorrect results but just degrade the performance and moreover the purpose of validating the feature will not be achieved.

We discuss the idea of testing the `untied` as shown in figure 6. First we create a set of tasks in parallel region and save the thread id executed by each task. Then we suspend all the tasks using `taskwait`. We send half of the threads into a busy loop so that at least half of the other idle threads could be rescheduled to the suspended tasks. We compare the thread number before and after the suspension. Since task is untied, tasks could be rescheduled by different threads after the suspension. In the cross test, the `untied` clause is removed so that the tasks are tied with the execution thread by default. As a result, the thread number before and after the task suspension should be the same delivering incorrect result.

Besides the `tasking` model, OpenMP 3.0 defines a new `collapse` clause for the `loop` construct that handles perfectly nested multi-dimensional loops. This clause collapses the loops, it is associated with, into one single loop,and controls the number of loops associated with one larger loop. The order of iterations in the collapsed loop is determined by the order of iterations in all loops before the collapse. If no `collapse` clause specified, the only loop that is associated with the `loop` construct is the one that immediately follows the construct.

Figure 5 shows the basic idea of testing the `collapse` clause that binds the two loops together. With the `ordered` clause, both i and j loops should be executed in order, thus the variable *my_islarger* should be TRUE. In the cross test, since the `collapse` clause is removed, the only loop that is associated with the `loop` construct is the i loop, the one that immediately follows the construct which should be executed in parallel and the only j loop will be executed in order. Consequently, the result will be incorrect.

```
int omp_task_untied(){
 int init_tid[NUM_TASKS];
 int curr_tid[NUM_TASKS];
 int i, count=0;

 /*Initialization*/
 for(i=0; i<NUM_TASKS; i++){
   init_tid[i]=0;
   curr_tid[i]=0;
 }

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task untied
    {
     init_tid[myi]=
          omp_get_thread_num();
     #pragma omp taskwait
     if ((init_tid[myi]%2) == 0){
      sleep (SLEEPTIME);
      curr_tid[myi]=
           omp_get_thread_num();
     } /*end of if*/
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /* end of parallel */

 for(i=0;i<NUM_TASKS;i++){
  if(curr_tid[i]!=init_tid[i])
   count++;
 } /*end of for*/
 return count;
} /*end of test*/
```

Fig. 6. Test for `untied` clause

```
int test_omp_task(){
 int tids[NUM_TASKS];
 int i, error=0;

 /*Initilization*/
 for(i=0; i<NUM_TASKS; i++)
 {
   tids[i]=0;
 }

 #pragma omp parallel
 {
  #pragma omp single
  {
   for (i = 0; i < NUM_TASKS; i++){
    int myi = i;
    #pragma omp task
            final(myi>=THRESH)
    {
     sleep (SLEEPTIME);
     tids[myi]=
        omp_get_thread_num();
    } /* end of omp task */
   } /* end of for */
  } /* end of single */
 } /*end of parallel */


 /*check tid beyond thresh*/
 for (i =THRESH;i < NUM_TASKS;i++)
 {
  if (tids[THRESH] != tids[i])
   error++;
 }

 /*check for if result is correct*/
 return (error==0);
} /* end of test */
```

Fig. 7. Test for `final` clause

3.2 Support for OpenMP 3.1

OpenMP version 3.1 was released in July 2011, a refined and extended version of OpenMP 3.0. The `taskyield` construct defines an explicit scheduling point, i.e. the current task is suspended and switched to a different task in the team. The test for the `taskyield` construct is similar to the test for `untied` clause, except for the `taskwait` begin replaced by `taskyield`.

The OpenMP 3.1 also provides a new features to reduce the task generation overhead by using `final` and `mergeable` clause. If the expression in `final` clause is evaluated to true, the task that is generated will be the final task and no further tasks will be generated. Consequently, it reduces the overheads of generating new tasks, especially in recursive computations such as in *Fibonacci* series when the *Fibonacci* numbers are too small. The test for the `final` clause is shown in Figure 7. The idea is to set a threshold and if the task number is larger to the threshold, that particular task will be the final task. We save the *task id* to check if the task larger to the threshold is executed by the same task.

In OpenMP 3.1, the `atomic` is refined to include the `read, write, update,` and `capture` clauses. The `read` along with the construct `atomic` guarantee an atomic read operation in the region. For instance `x` is read atomically if `v=x`. Similarly, the `write` forces an atomic write operation. It is much more lightweight using `read` or `write` separately than just using `critical`. The `update` clause forces an atomic update of an variable, such as *i++, i–*. If no clause is presented at the `atomic` construct, the semantics are equivalent to atomic update. The `capture` clause ensures an atomic update of an variable that also captures the intermediate or final value of the variable. For example, if `capture` clause is present then in `v = x++`, `x` is atomically updated while the value is captured by `v`.

OpenMP 3.1 also extends the `reduction` clause to add two more operators: `max` and `min`, that is to find the largest and smallest values in the reduction list respectively.

For our tests, we use a common search/sort algorithm to discuss the reduction clause and compare the results with a known reference value.

4 Evaluation

In this section, we use the OpenMP validation suite to evaluate the correctness of some of the open source an vendor compilers including OpenUH, GNU C, Intel and Oracle Studio compiler (suncc). The experiments were performed on a Quad dual-core Opteron-880 machine and we used eight threads to perform the evaluation.

To begin with, we use our in-house OpenUH compiler [14,13], to test the correctness of OpenMP implementation in the compiler. Currently, OpenUH supports OpenMP Version 3.0. OpenUH compiler is a branch of the open-source Open64 compiler suite for C, C++, Fortran 95/2003, with support for a variety of targets including x86 64, IA-64, and IA-32. It is able to translate OpenMP 3.0, Co-array Fortran, UPC, and also translates CUDA into PTX format. An OpenMP implementation translates OpenMP directives into corresponding POSIX thread code with the support of runtime libraries.

The versions of the other compilers that we have used for the experimental purposes are:

- GNU compiler is 4.6.2 (gcc)
- Intel C/C++ compiler 12.0 (icc)
- Oracle Studio 12.3 (suncc)

For the first round of experiments, we disable the optimization flags to avoid any potential uncertainties, e.g. code reconstruction while compiling the unit tests. And then we turn on the -O3 optimization as most of the time compiler optimizations are highly used by programmers.We did not find any differences with the turning off/on of the optimizations flags. Almost all the tests passed with 100% certainity.

Table 1 shows the experimental results of evaluating the directives on several compilers. For each sub-column, N is normal test, C is cross test, O is orphan

Table 1. Experimental results on several compilers

	OpenUH				Gnu				Intel				Oracle			
Directive	N	C	O	OC	N	C	O	OC	N	C	O	OC	N	C	O	OC
has_openmp	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_shared	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_private	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_firstpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_if	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_reduction	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_copyin	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_num_thres	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_default	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_private	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_firstpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_lastpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_reduction	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_sche_dynam	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_sche_static	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_sche_guided	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_collapse	100	100	100	100	0	-	0	-	100	100	100	100	100	100	100	100
for_ordered	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
for_nowait	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sec_private	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sec_firstpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sec_lastpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sec_reduction	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sec_nowait	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sing_priv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sing_firstpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sing_copypriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
sing_nowait	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_priv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_fpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_lpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_ordered	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_reduc	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_for_if	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_sec_fpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_sec_lpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_sec_priv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
para_sec_reduc	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task_private	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task_firstpriv	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task_if	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task_untied	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
task_default	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
master	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
critical	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
barrier	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
taskwait	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
atomic	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
flush	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
threadprivate	100	100	100	100	100	100	100	100	100	100	100	100	100	100	ce	-
get_wtick	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
get_wtime	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

test while OC is orphan cross test (the cross test within orphan test). Each row contains the directive to be tested. For instance, the `para_shared` is to test the `shared` clause in the `parallel` construct. Other terms such as `ce` means compile error and textttto means time out (reach the maximum execution time threshold we set in case of deep dead).

Using statistical analysis approach, the tests are repeated several times (the number of times to be repeated is configured at the beginning) and the number of times the tests pass/fail is calculated. Through this strategy, we can capture uncommon circumstances, where a compiler would still fail but pass under normal circumstances. From the experimental results, we see that most of the tests pass with 100% certainty. However, we could still see that the `collapse` implementation fails for the GNU C compiler, and the `threadprivate` implementation fails for the Oracle Studio compiler.

It is quite challenging to analyze the reason behind why a compiler would fail certain tests. But our validation suite still tries to provde as much detail as necessary to the compiler developers in order to assist them in improving the implemenation of the features in the compiler. Also we believe that the validation suite will help resolve ambiguities in the OpenMP specification and help refine the same if necessary.

5 Related Work

To the best of our knowledge, there is no similar public efforts reporting on the validation of OpenMP implemenations on compilers. Vendors have their own internal testsuites but this does not allow for open validation of implemented features which may be of great importance to application developers. As mentioned in Section 1, [16,17] report on the validation methodologies and testsuite for older OpenMP versions (2.0 and 2.5). Since there is no means with OpenMP specification by which an user can obtain dynamic feedback on the success or otherwise of a specific feature, open means for testing features' availability is a matter of concern. A path to extend OpenMP with error-handling capabilities was proposed in [19]. This effort was to address OpenMP's lack of any concept of errors (both OpenMP runtime and user code errors) or support to handle them. A number of works report on evaluation of peformance measurement using OpenMP, for e.g. EPCC [6],PARSEC [4],NAS [12],SPEC [3],BOTS [10] and SHOC [9].

A methodology called *randomized differential testing* [20] was developed that employs random code generator technique to detect compiler bugs. This is an hand-tailored program generator methodology that took about three years to complete, this work identifies compiler bugs that are not uncommon. Although this effort has helped find more than 325 bugs so far in common compilers, the execution environment is quite complex, this tool generates programs that are too large, consequently bug reports are hard to understand. In order to clean this up, manual intervention will be required, since automated approach would introduce unidentifiable undefined behavior. Also it requires voting heuristics to determine

which compiler implementation is wrong, this can be hard to determine at times. In our approach we use fine-grained unit tests for each of the OpenMP directives, this will help us determined the faults due to erroneous implementations very easily. Other approaches to detect bugs in compilers include [7,15]. Our approach is slightly different in a way that we designed an efficient, portable and flexible validation framework that will detect bugs in OpenMP implementations in various compilers.

6 Conclusion

The work in this paper presents a validation testsuite evaluating OpenMP implementations on several different compilers, both academic and commercial compilers. The validation suite basically validates OpenMP Version 3.1 specification. We developed a framework, that employs an automatic approach to run different types of tests such as *normal*, *cross* and *orphan* tests. The framework provides a flexible, portable and user-friendly testing environment.

Acknowledgements. We would like to acknowledge the use of shark/crill cluster machine that was provided by NSF infrastructure grant NSF-CNS-0958464. We used the cluster machine to perform experiments for this project.

References

1. The Open64 Compiler, `http://www.open64.net/`
2. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 Tasking Implementation in OpenUH. In: Open64 Workshop at CGO, vol. 2009 (2009)
3. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, pp. 72–81. ACM, New York (2008)
5. Board, O.A.R.: OpenMP Application Program Interface, Version 3.1 (July 2011)
6. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
7. Burgess, C., Saidi, M.: The Automatic Generation of Test Cases for Optimizing Fortran Compilers. Information and Software Technology 38(2), 111–119 (1996)
8. Cappello, F., Etiemble, D.: MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks. In: ACM/IEEE 2000 Conference on Supercomputing, p. 12. IEEE (2000)
9. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU 2010, pp. 63–74. ACM, New York (2010)

10. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in Openmp. In: Proceedings of the 2009 International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE Computer Society, Washington, DC (2009)
11. C. Intel and C. User. Reference guides. Available on the Intel Compiler Homepage (2008), `http://software.intel.com/en-us/intel-compilers`
12. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report (1999)
13. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. Concurrency and Computation: Practice and Experience 19(18), 2317–2332 (2007)
14. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. In: 12th Workshop on Compilers for Parallel Computers, p. 2006 (2006)
15. McKeeman, W.: Differential Testing For Software. Digital Technical Journal 10(1), 100–107 (1998)
16. Müller, M., Neytchev, P.: An OpenMP Validation Suite. In: Fifth European Workshop on OpenMP, Aachen University, Germany (2003)
17. Müller, M., Niethammer, C., Chapman, B., Wen, Y., Liu, Z.: Validating OpenMP 2.5 for Fortran and C/C
18. Stallman, R.M., GCC DeveloperCommunity: Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3. CreateSpace, Paramount, CA (2009)
19. Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., de Supinski, B.R., Churbanov, A.: Towards an Error Model for OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 70–82. Springer, Heidelberg (2010)
20. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and Understanding Bugs in C Compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 283–294. ACM, New York (2011)