

# An OpenMP Validation Suite

Matthias Müller and Pavel Neytchev

HLRS, University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany,  
mueller@hlrs.de

August, 2003

## Abstract

We present a collection of C programs with OpenMP directives that were designed to validate the correctness of an OpenMP implementation. The validation methodology and implemented tests are presented together with a discussion where the OpenMP standard is precise enough to allow a verification. To show the effectiveness of the suite a coverage analysis of a freely available OpenMP compiler is performed.

## 1 Introduction

In the last years OpenMP [7] has found wide spread acceptance as portable programming model. This is not only reflected by the use of OpenMP in many applications but also by the availability of a large number of compilers supporting OpenMP. In addition to commercial products [2, 4, 8] a number of research or open source projects exist [1, 3, 5, 6].

Currently the OpenMP 2.0 standard has about one hundred pages. For the C/C++ language binding it consists of ten different constructs, two directives and eleven clauses that can be applied to the constructs. Although not all clauses can be used together with all constructs this results in a sufficient complexity to make the implementation of an OpenMP compiler a difficult task. Unfortunately, to the best of our knowledge, there exists currently no freely available validation suite that could support the efforts to develop open source OpenMP compilers. The target of this work is to provide a starting point for such a suite.

In the next section the overall design is presented. In section 3 the content of the validation suite is described. In the following section we shortly describe our experiences with several compilers. To show that the suite actually tests a large portion of a compiler the results of a coverage analysis are presented in section 4.1.

## 2 Design of the Validation Suite

The idea of the suite is to have a subroutine for each OpenMP construct that returns true if the construct works as expected and false otherwise. Each subroutine will perform a calculation for which the correct result should depend on the correct functionality of the used OpenMP construct. In order to check the dependency of the result from the correct implementation we evaluate the result when the construct is missing. To increase the likelihood to catch a race condition we perform  $N$  repetitions of each test. Clearly, if a test fails once the construct is not working correctly. To estimate the likelihood that the test is passed accidentally we take the following approach: If  $n_f$  is the number of failed cross checks and  $M$  the total number of iterations, we estimate the probability of the test to fail with  $p = \frac{n_f}{M}$ . The probability that an incorrect implementation passes the test is  $p_a = (1 - p)^M$  and the certainty of the test  $p_c = 1 - p_a$ .

It is clear that a missing OpenMP directive will have different race conditions than a broken implementation, but the approach described above gives an estimate of the reliability of the test. The existence of race condition will also depend on the optimization level, e.g. if a variable is stored in a register a missing private clause may not lead to wrong results.

Of course a directive is not simply removed for the crosscheck, but replaced by a directive that just does not contain the functionality to be tested. E.g. when a `firstprivate` is removed for the cross checks it is not merely removed, but replaced by a `private` clause. Tab. 1 contains a list of directives and their replacements.

## 3 Content of the Validation Suite

Ideally the validation suite should cover the complete standard. Most of the directives can be combined with a number of clauses that modify the behavior of the construct. Tab. 2 gives an overview of the valid combinations. In

clause	substitute
if	
private	shared
firstprivate	private
lastprivate	private
ordered	
single	
copyprivate	private
copyin	
reduction	
num_threads	

Table 1: OpenMP directives and clauses and their substitutions in the cross checks.

parallel	if, private, firstprivate, default, shared, copyin, reduction, num_threads
for	private, firstprivate, lastprivate, reduction, ordered, schedule, nowait
sections	private, firstprivate, lastprivate, reduction, nowait
single	private, firstprivate, copyprivate, nowait
parallel for	if, private, firstprivate, default, shared, copyin, reduction, num_threads, reduction, ordered, schedule
parallel sections	if, private, firstprivate, default, shared, copyin, reduction, num_threads

Table 2: OpenMP directives and valid clauses.

In addition to the directives several runtime library calls are included in the standard.

Since most of the directives cannot be used isolated most of the tests will contain a combination of constructs. Currently the suite consists of 38 different tests. Instead of describing them all some examples are provided in this section to give an idea of the included tests.

### 3.1 Workshare Directives

There are several workshare directives in OpenMP. The major two are for parallel loops and for parallel sections. The `single` directive will limit the execution to one thread. Finally, the `workshare` directive is only available for the F90 binding. The major task of a validation suite is not to test the workshare directives, but the set of clauses that can be combined with each directive.

#### 3.1.1 Loops

The `reduction` clause is tested by calculating  $S = \sum_{i=1}^N i$  and comparing it with the known result  $S =$

$N * (N + 1) / 2$ . Both  $N$  and  $i$  are integers to avoid any problems with rounding errors. The test is currently limited to this single calculation, other operations or types of variables are not tested. In section 4.1 we will see how this limits the effectiveness of the test suite.

The `private` clause is tested by performing the same calculation with a manual implementation calculating local sums for each thread and adding them in a critical section. The local sums are stored in a private variable. For the `firstprivate` clause the initialization has to be performed by the compiler to get the correct result. For the `lastprivate` clause we introduce a private variable that stores the value of the loop count variable on each thread and verify that it has the value  $N$  after the loop.

The `ordered` clause is tested by calculating a reduction (sum) in a parallel for loop, where the calculation is performed in the section with the ordered directive. In addition we verify that the loop count variable is increased in a monotonic fashion across the threads.

```
#pragma omp for ordered
for (i=1; i<=N; i++)
{
#pragma omp ordered
{
my_islarger= islarger(i)
&& my_islarger;

sum=sum+i;
}
}
}
```

The function `islarger()` stores the last value of its parameter and checks whether the next value is larger:

```
static int last_i=0;
int islarger(int i){
int islarger;
islarger=(i>last_i);
last_i=i;
return (islarger);
}
```

#### 3.1.2 Sections

The section directive is again tested by calculating  $S = \sum_{i=1}^N i$ . Instead of calculating the sum in one single loop, several partial sums are calculated, one for each section.

```
#pragma omp parallel
{
#pragma omp sections reduction(+:sum)\
private(i)
{
```

```

#pragma omp section
{
for (i=1;i<N1;i++){
    sum += i;
}
}
#pragma omp section
{
for(i=N1;i<N;i++)
    sum += i;
} /* end of section reduction.*/
} /* end of parallel */

```

With this approach we can test the appropriate clauses (reduction, private, firstprivate, lastprivate) with tests that correspond to the tests used for the for directive.

### 3.1.3 Single Directive

For the single directive we verify that each block protected by the single directive is executed only by one thread. We do this by counting the number of executions of this block:

```

#pragma omp parallel private(i)
{
    for (i=0;i<N;i++)
    {
#pragma omp single
{
#pragma omp flush
    nr_threads_in_single++;
#pragma omp flush
    nr_iter++;
    nr_threads_in_single--;
    result=result+nr_threads_in_single;
} /* end of single*/
    } /* end of for */
} /* end of parallel */
return(result==0)&&(nr_iter==N);

```

Since there is an implicit barrier at the end of the single construct we can also check that there is only one thread active inside the block at any single point in time.

### 3.1.4 Combined Workshare Constructs

For the combined workshare constructs `parallel for` and `parallel section` we simply join the parallel directive and the workshare directive described in the tests above.

## 3.2 Master and Synchronization Constructs

For the master construct we check that the block with the master directive is executed only by one thread and that this thread is thread number zero.

```

#pragma omp parallel
{
#pragma omp master
{
#pragma omp critical
{
    nthreads++;
}
    exe_thread=omp_get_thread_num();
} /* end of master*/
}
return ((nthreads==1) &&
        (exe_thread==0));

```

### 3.2.1 Critical and atomic directive

For the critical directive we perform the usual reduction  $S = \sum_{i=1}^N i$ , where the statement  $S=S+i$  is inside a critical directive instead of using a reduction clause. In the same way we check the atomic clause.

### 3.2.2 Barrier Clause

IN order to test the barrier clause we have to bring the threads out of sync and test whether the threads wait for each other at the barrier directive. We do this by executing a sleep statement with thread one and verifying that the assignment made by this thread is visible to the thread zero after the barrier. In this way we also use the additional property of barrier to provide a consistent view to the memory afterwards.

### 3.2.3 Flush Directive

To test the effectiveness of the flush directive we verify that a value written by one thread is visible to a second thread. With an appropriate delay we make sure that the consumer calls flush after the producer has done so.

## 3.3 Data Environment

### 3.3.1 Threadprivate clause

The threadprivate clause is checked like the private clause, but the variable used to calculate the partial sums is a static variable. In addition we check that the value of a variable

that is declared `threadprivate` retains its value between parallel regions. Care has been taken that the dynamic threads mechanism is disabled for this test.

### 3.3.2 Copyin clause

The more interesting case is the `copyin` clause. We have to verify that the value owned by the master thread is communicated to the other threads at entry of the parallel section.

```
static int sum1=789;
#pragma omp threadprivate(sum1)

int check_omp_copyin()
{
    int sum=0;
    int known_sum;
    int i;
    sum1=0;
#pragma omp parallel copyin(sum1)
    {
        /*printf("sum1=%d\n",sum1);*/
#pragma omp for
        for (i=1;i<N;i++)
            {
                sum1=sum1+i;
            }
#pragma omp critical
            {
                sum= sum+sum1;
            }/*end of critical*/
    }/* end of parallel*/
    known_sum=((N-1)*N)/2;
    return (known_sum==sum);
}/* end of check_threadprivate*/
```

### 3.4 Runtime Library

Several tests are performed to test the existence and the functionality of the OpenMP runtime library calls. For the conditional compilation we simply check whether code contained in an `#ifdef _OPENMP` clause is compiled. For `omp_get_num_threads()` we count the number of threads in a parallel region and check whether the result is consistent with the value returned by the runtime function:

```
int nthreads=0;
int n_lib;
#pragma omp parallel
{
    #pragma omp critical
```

```
    {
        nthreads++;
    }
#pragma omp single
    {
        n_lib=omp_get_num_threads();
    }
} /* end of parallel */
return nthreads==n_lib;
```

The check for `omp_in_parallel` is whether it returns the correct value inside and outside a parallel region.

Checking the lock functions is more complicated. The basic idea is to make similar tests as in the case of critical sections. A call to `omp_set_lock` is performed on entry to the section. On exit `omp_unset_lock` is called. We replace `omp_set_lock` with a spin loop containing `omp_test_lock` to test this function.

For nested locks we currently perform the same tests as for the unnested locks.

### 3.5 Scheduling Clauses

As described above the validation suite contains different scheduling clauses. The main objective during the selection process was to increase the likelihood of race conditions. In this way the schedule clauses are tested together with other constructs. However, this does not include a validation of the specific properties of the different schedule types. We currently develop several procedures to identify the chunk sizes assigned to the threads and how they develop throughout the loop execution. Since an OpenMP implementation does not need to stick exactly to the description of the schedule types provided in the standard it is difficult to come up with an automatic test for scheduling. Currently this is done in a semi-automatic way.

### 3.6 OpenMP 2.0 features

The OpenMP 2.0 features are implemented in a separate compilation unit. This allows to skip this tests if the compiler only supports OpenMP 1.x. For the C binding of OpenMP the new features are the `num_threads` clause, the `copyprivate` clause and the `omp_get_wtime` and `omp_get_wtick` runtime library functions. Extended features are the possibility to separate clauses by comma, the use of C99 variable length arrays and the privatization of private variables.

For the `num_threads` clause we increase the number of threads from one to the value set by the `OMP_NUM_THREADS` environment variable. We

count the number of threads inside the parallel region with the `num_threads` clause and also verify that the number is consistent with the value given by `omp_get_num_threads`.

The time measured by `omp_get_num_threads` is compared with the value provided by `gettimeofday`. To check `copyprivate` we modify the value of a private variable inside a `single` directive and check that this value is distributed to all other threads.

The support of OpenMP 2.0 is currently limited, e.g. we don't check any of the extensions of existing clauses. With increasing number of compilers supporting V2.0 we plan to extend the validation suite.

File	gcc	OpenMP	gcc+OpenMP
C-compile-decl.c	72%	45%	72%
C-compile-expr.c	83%	34%	83%
C-expr-mem.c	96%	81%	96%
C-lex.c	73%	46%	74%
C-main.c	62%	41%	62%
C-mem.c	100%	84%	100%
C-omp-pragma.c	0%	56%	56%
C-opt-expr.c	0%	0%	0%
C-pragma.c	5%	24%	24%
C-tea-pragma.c	0%	0%	0%
X-decompile.c	0%	0%	0%
X-io.c	32%	28%	32%
machine-dep.c	89%	89%	89%

Table 3: Results of the coverage analysis.

## 4 Results

First of all you would like to verify that the validation suite itself is correct. In addition to a careful implementation the correct execution with different compiler is an indication that it works reliable. In addition we used the tool *assure* to verify that the program is correct, at least for the constructs that can be tested with *assure*.

We applied the validation suite to ten different compilers. For five of them problems were reported. In two cases we were able to identify a compiler bug. One of them was fixed with an upgrade to the latest release. The second was a problem with an incorrect implementation of the `single copyprivate` construct. It was fixed after the problem was escalated to the appropriate support level.

Another interesting point when looking at the results of different compilers is the effectiveness of the implemented cross checks. Tab. 4 shows the results of ten different platforms. For the tests the number of iterations were set to  $M = 5$ . Most of the tests achieve a failure rate of 100%. Only some tests don't fail when the OpenMP clause is replaced. Some tests show the existence of a race condition with a failure rate between 20% and 80%. Depending on the platform between 26 and 33 of the 35 tests are successfully cross checked. We conclude that most of the tests are well designed for testing the underlying OpenMP infrastructure. This was achieved by writing the tests in a way that increase the requirements on the synchronization that needs to be added by the OpenMP compiler. E.g. instead of the default schedule that results in one big chunk for every thread with only little need for synchronization we have chosen small chunk sizes and different schedules for the tests wherever appropriate.

### 4.1 Coverage Analysis

The design of this verification suite was made from the users point of view. They were made by reading the OpenMP standard and creating tests that seemed to be appropriate to test whether the functionality provided by a certain feature of the standard works as expected. An implementor of an OpenMP compiler and runtime library would probably have a different view and more insight into the required tests. In order to test the usefulness of this suite we performed a coverage analysis with the freely available Omni compiler[6].

A compiler for a programming language like C is a large piece of code and only a small fraction of it will handling the OpenMP specific compilation. This is also true for a front-end compiler like Omni that uses a back-end compiler for the actual code generation. If we used the OpenMP validation suite alone most of the uncovered parts would be responsible for tasks related to plain C. For an easier identification of uncovered OpenMP relevant code inside the compiler we combined the validation suite that comes with the GNU gcc compiler with our suite. The idea was that the majority of the code that is not covered by the gcc suite should be responsible for OpenMP. Tab. 3 shows the coverage for different files of the Omni compiler. The two columns indicate the results of the suites alone and the combination of both.

After applying both suites we could analyze the remaining uncovered parts to find shortcomings in the OpenMP validation suite. Parts of it were due to unused flags from the compiler and internal error conditions that were not triggered. The compiler also supports some language extensions like `long long` that were not used by the suites. The largest fraction with more than 23 different locations is responsible to handle syntax errors in OpenMP con-

structs.

The remaining items identified some real OpenMP constructs that the validation suite did not cover.

- The first was the `default` clause. Both were left out intentionally, because `default(shared)` is the default and `default(none)` will not change the runtime behavior. Therefore it cannot be tested with a suite like this.
- Several types of schedules were not included in the suite: `guided`, `runtime`, `affinity`. `Affinity` is an extension to standard OpenMP. The other schedules were not included due to the difficulties to verify the scheduling algorithms.
- The `nowait` clause was the first item that accidentally was not included.
- For the reduction clause we only checked one operator. The operators `*`, `-`, `&`, `^`, `|`, `&&` and `||` are currently not tested.

## 5 Summary and Conclusion

In this paper we presented an OpenMP validation suite. It consists of a number of routines to test the functionality of OpenMP constructs. By careful cross checking we test whether the correct functionality of the constructs under examination is actually required in order to deliver the correct result. We applied the validation suite to several compilers and found that it is capable to identify bugs and problems with several compilers. The effectiveness of the tests is also demonstrated with an efficient cross checking. In addition we performed a coverage analysis with one open source compiler to verify that for the compilation of the validation suite all OpenMP relevant part of the compilers are required. We believe that the validation suite is useful for the development of compilers and tools for OpenMP and hope that it will be widely applied by the OpenMP community.

Future work will be the extensions of the suite to cover the missing items that have been identified by the coverage analysis or omitted in the first release. Another focus will be the improvement of existing tests to increase the probability that an incorrect implementation will fail on the test. An integration in an external framework will offer the possibility to include things that are currently not addressed by this suite. Examples are different runtime schedules or the sensitivity to optimization levels.

## Acknowledgments

We would like to thank Sharat Maddineni for the help in creating the validation suite.

## References

- [1] Eduard Ayguade, Marc Gonzalez, Jesus Labarta, Xavier Martorell, Nacho Navarro, and Jose Oliver. Nanoscompiler: A research platform for openmp extensions. In *EWOMP'99 - First European Workshop on OpenMP*, Lund University, Lund, Sweden, Sept. 1999.
- [2] <http://www.intel.com/software/products/compilers/>.
- [3] Roy Ju, Sun Chan, Fred Chow, and Xiaobing Feng. Open research compiler (orc): Beyond version 1.0. In *PACX'02 - The Eleventh International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, September 22-25 2002.
- [4] <http://www.kai.com>.
- [5] Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eigenmann. Portable compilers for openmp. In R. Eigenmann and M.J. Voss, editors, *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2001*, number 2104 in LNCS. Springer, West Lafayette, IN, USA, July 2001.
- [6] OMNI OpenMP compiler, <http://pdplab.trc.rwcp.or.jp/Omni>.
- [7] OpenMP Architecture Review Board. *OpenMP Specifications*. <http://www.openmp.org/specs>.
- [8] <http://www.pgroup.com>.

Platform	1	2	3	4	5	6	7	8	9	10
check_has_omp	100	100	100	100	100	100	100	100	100	100
check_omp_get_num_threads	100	100	100	100	100	100	100	100	100	100
check_omp_in_parallel	100	100	100	100	100	100	100	100	100	100
for ORDERED	100	100	100	100	100	100	100		100	100
for REDUCTION	100	0	100	100	100	100	100	100	100	100
for PRIVATE	100	0	100	100	100	100		100	0	100
for FIRSTPRIVATE	0	100	100	100	40	100	0	100	0	100
for LASTPRIVATE	100	100	60	20	0	0	100	40	100	20
section REDUCTION	0	0	100	100	100	100	100	100	100	100
section PRIVATE	80	0	100	100	100	100	0	100	100	100
section FIRSTPRIVATE	100	100	100	100	100	100	100	100	100	100
section LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
SINGLE	100	100	100	100	100	100	100	100	100	100
single PRIVATE	0	60	60	0	0	0	60	0	0	0
SINGLE NOWAIT	100	100	100	100	100	100	100	100	100	100
parallel for ORDERED	100	100	100	100	100	100	100		100	100
parallel for REDUCTION	0	100	100	100	0	100	100	0	100	100
parallel for PRIVATE	0	0	0	0	0	0	0	0	0	0
parallel for FIRSTPRIVATE	100	100	100	100	100	100	100	100	100	100
parallel for LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
parallel section REDUCTION	20	100	100	100	100	100	100	100	100	100
parallel section PRIVATE	0	100	100	100	100	100	0	100	100	100
parallel section FIRSTPRIVATE	100	100	100	100	100	100	100	100	100	100
parallel section LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
omp MASTER	100	100	100	100	100	100	100	100	100	100
omp CRITICAL	0	100	100	100	100	100	60	100	100	100
omp ATOMIC	0	100	100	100	100	100		100	100	100
omp BARRIER	100	100	100	100	100	100	100	100	100	100
omp FLUSH	0	0	0	0	0	0	0	0	0	
omp THREADPRIVATE	100	100	100		100	100	100			
omp COPYIN	100	100	100	100	80	100	100	100		100
omp LOCK	100	100	100	100	100	100		100	100	100
omp TESTLOCK	100	100	100	100	100	100		100	100	100
omp NEST_LOCK	100	100	100	100	100	100		100	100	100
omp NEST_TESTLOCK	100	100	100	100	100	100		100	100	100

Table 4: Results of the cross checks.