Validating OpenMP 2.5 for Fortran and C/C++

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

Barbara Chapman, Yi Wen, Zhenying Liu
University of Houston

Abstract

We present a collection of C/C++ and Fortran programs with OpenMP directives that were designed to validate the correctness of an OpenMP implementation. The validation methodology and implemented tests are demonstrated. We also discuss the differences between the Fortran and C validation suite and extensions made possible by the clarifications introduced by OpenMP 2.5.

1. Introduction

In the last years OpenMP [3] has found wide spread acceptance as a portable programming model. This is not only reflected by the use of OpenMP in many applications but also by the constant development of the standard. Since its first release in 1997, the OpenMP Architecture Review Board has released several versions of the standard. The OpenMP 2.5 standard is currently being finalized. It will not only merge the Fortran and C/C++ language binding, but it also contains a number of clarifications or more detailed specifications. In [2] a first proposal for an open and portable validation suite for the C language binding was presented. In this paper we discuss its extension to Fortran, improvements made during the last year and new tests that were made possible due to the clarifications introduced with the upcoming OpenMP 2.5.

This paper is organized as follows. In the next section we describe the basic design of the validation suite. In Section 3 we describe the improvements of the suite since last year and explain how it has been extended. Section 4 shows some results with current compilers. We draw a conclusion in Section 5.

2. Design

The idea of the suite is to have a subroutine for each OpenMP construct that returns the value *true* if the construct works as expected and *false* otherwise. The target is to completely cover all constructs and clauses of OpenMP (see Fig. 1).

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 of catching a race condition we perform N repetitions of each test, currently N is fixed and set to 20. 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 lead to different race conditions than a broken implementation, but the approach described above gives an estimate of the reliability of the test. The existence of a 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 does not contain the functionality to be tested. For example, when a `firstprivate` is removed for the crosschecks, it is not merely removed, but replaced by a `private` clause which will change the initial values of the variables to be undefined. Tab. 1 contains a list of directives and their replacements.

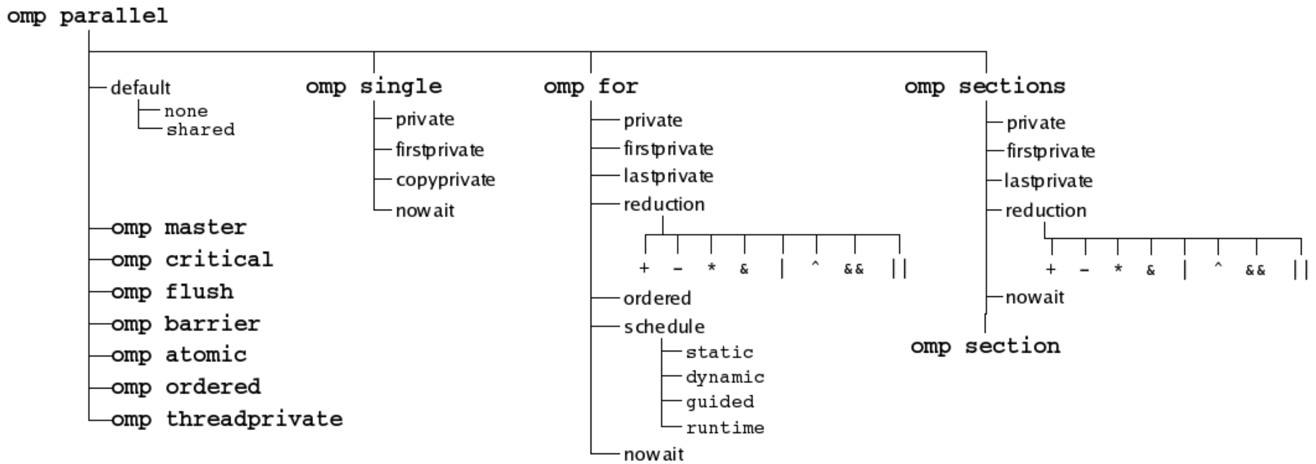

Figure 1. An overview of OpenMP directives and clauses.

Clause	Substitution
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.

3. Extensions of the Validation Suite

An early C version of the validation suite was introduced in [2], and we extend it in this section.

3.1. Extension to Fortran

We realized the Fortran OpenMP validation suite by converting the original C version [2] to Fortran. In almost all cases, this translation was straightforward. However, we did have to modify several variable names that were not accepted by some OpenMP compilers. Fortran 90 was used as it gave us the convenience of long procedure names and enabled us to give these names that are similar to their C counterparts [2]. Furthermore, dynamic arrays and the derived type of Fortran 90 are used in accordance with C. We sometimes force the lower bound of an array to be 0 for convenience although by default it is 1 in Fortran. As a result, the suite currently does not work with Fortran 77; however, it is easy to produce equivalent Fortran 77 and this will also be provided. One exception is the strategy used to test OpenMP locks, which employs the Fortran 90 KIND construct. A lock variable with the `KIND` of `OMP_LOCK_KIND` and/or `OMP_NESTED_LOCK_KIND`, declared in *omp_lib.h*, must be specified for the current test.

```
    USE OMP_LIB
    INTEGER (KIND=OMP_LOCK_KIND) :: LCK
    CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK)
    ...
    CALL OMP_SET_LOCK(LCK)
    ...
    CALL OMP_UNSET_LOCK(LCK)
!$OMP END PARALLEL
  ...
    CALL OMP_DESTROY_LOCK(LCK)
```

At this time, our validation suite has been used to test all of the Fortran OpenMP 1.0 features for Sun Forte 7 Fortran 95 compiler, Intel 8.0 compiler and the Open64 compiler [1]. We are working on the complete Fortran OpenMP 2.0 and 2.5 specification, including `WORKSHARE` directive.

3.2. Reductions

In the previous version of our tests, the `reduction` clause was tested only by calculating $S = \sum_{i=1}^{N} i$ and comparing it with the known result $S = N * (N + 1)/2$. Both N and i were integers to avoid any problems with rounding errors. The test was limited to this single calculation; other operations or types of variables were not tested. In the meantime we have implemented tests for all

reduction operations (`+,-,*,,&,|,^,&&,||` in C and `+,*,-,.AND.,.OR.,.EQV.,or.NEQV.` in Fortran), both for integer as well as floating point variables. The input data sets for the boolean operations are constructed to detect any omissions of a single value. E.g. for the logical `or` all values of the input array are false with only one value set to true.

For the reduction operation of floating point variables we use the calculation of $E = \sum_{i=1}^{N} \frac{1}{k^i}$ with $k = 3$ and compare it with the analytical result.

3.3. Schedule clauses

One of the areas of clarifications introduced with OpenMP 2.5 are the scheduling clauses. Now, several tests are possible with respect to the schedule clause used in conjunction with a parallel loop. For `static` as well as `dynamic` scheduling we check whether the chunks have the requested size, with the legal exception of the last chunk. For the `guided` clause we verify that the chunk size is decreasing following the algorithm outlined in the OpenMP specification. For `schedule(runtime)` the tests of the respective clause apply, and the execution environment (see Sec. 3.5) has to set the correct value for `OMP_SCHEDULE`.

The basic idea for identifying the chunks is simple. We simply perform the loop

```
#pragma omp for schedule(runtime)
    for(i=0;i<MAX_SIZE;++i)
    {
        a[i]=thread_id;
    } /*end omp for*/
```

However, in the dynamic case we must avoid assigning two successive chunks to the same thread, because this will appear like one larger chunk in the later analysis. The execution flow of the parallel loop is under the control of the compiler and runtime environment. Control is handed to the user program only after the chunk is already assigned to a specific thread. During execution only loop iterations are visible to the threads, so there is no way to decide if a single iteration is at the beginning or end of a chunk.

The algorithm we implemented is based on the idea that each thread has to wait until at least one different thread has been assigned the next chunk. The highest iteration count that is reached by the threads is saved in the shared variable *maxiter*. Each thread waits before it continues to iterate until

1. another thread has reached a higher iteration count
2. a timeout occurs
3. the first thread has left the parallel loop

The first condition makes sure that two successive chunks cannot be assigned to the same thread. The second guarantees progress, e.g. when the last chunk is assigned to the threads. The last condition combined with a `nowait` clause improves the execution time, because no timeout has to be used at the end of the loop.

With this test we can now not only test the standard conformance of the compilers, but also what decision the implementors have made regarding implementation defined behavior. Unfortunately the tests only worked with seven different compilers. One did reject the source code, and with two it did not work correctly. This failure was expected in once case, because the flush test also did not work correctly.

The first implementation defined behavior is the default schedule. All compiler used `static` as their default schedule. Another option is the distribution of loop iterations if the total iteration count is not a multiple of the chunksize. One option is to reduce the last chunk, another option is to reduce the last chunk assigned to each thread. Five compiler take the first option and two compilers the last. For the guided scheduling the OpenMP 2.5 standard requires that the chunksize is proportional to the number of remaining items divided by the number of threads. The linear factor is undefined. We found that the compilers either choose 1.0 or 0.5 for the factor. One compiler did not implement guided correctly, all chunks had the same size.

3.4. Test Output

As the version before the new validation suite shows a brief overview of the test results during execution. New to this version is the creation of a logfile, which contains detailled information on the tests. This logfile contains the result of all tests that helps to find out much better what has been going wrong. For example the test for the reduction clause contains several smaller tests of its options. So if only one test within the reductiontest failed you see in the overview that the whole test failed but in the logfile you can see exactly which option didn't pass it's test and what was the reason for this.

3.5. Execution Environment

We have begun to create a flexible environment for building and executing the tests in the validation suite with the goal of ensuring that neither compilation nor execution of the validation suite will be terminated by the occurrence of an error; an additional goal is to permit users to add or remove individual tests, and set compiler options, as desired. Furthermore, we intend to provide the test results in a user friendly way. The environment needs components to control both the compilation and execution of the

OpenMP compiler tests. In the current version of the validation suite, if a compiler error is encountered in the translation of an OpenMP construct, directive or library function, then the compilation process will stop and the executables for the validation suite will not be created. In order to solve this problem, we intend to use a script that will enable the compiler to continue to translate tests for the remaining OpenMP language features and library routines in the presence of such errors. We use a Perl script to accomplish this functionality as Perl is an interpreted language that is convenient for reading, extracting and writing text files. Perl is also suitable for executing system commands, such as invoking compilation, and handling the return information from them. Hence Perl will enable us to recover from compiler errors and generate a report in this situation. The Perl script acts as a driver for the OpenMP compiler being tested, invoking it to compile the individual tests that are part of the suite; while doing so, it keeps a record of any compiler errors that are encountered. The generated code will be executed under the control of a Perl script at runtime in order to complete the execution of all the desired tests. Thus deadlocks may be detected and reported on if a particular test lasts very much longer than expected. The portability of this environment will be a strong consideration in our implementation of the Perl scripts.

The environment will enable the validation suite to be used to test a particular language feature of OpenMP if desired. Users will be permitted to input the name of a certain directive which is to be tested individually. In particular, after users have obtained a complete report of their OpenMP compiler, they may want to retest a specific language feature that was not passed by the validation suite. On the other hand, we intend to group some similar tests together and give them a name so that users may specify, for example, *worksharing* as a means to request the environment to build and run precisely those tests that evaluate the implementation of worksharing constructs in OpenMP, including DO, SECTIONS, SINGLE and WORKSHARE. We also plan to allow users to select OpenMP 1.0 or OpenMP 2.0 so all the features for the given version of the API can be tested accordingly.

We use different files to store information internally and output results in a user friendly format. The main purpose is naturally to report which directives pass or fail the test. We need a configuration file to store the executable path for the compiler and the compiler flags specified by the user. Note that different compiler flags may result in different outputs, and it might be interesting or necessary to evaluate an OpenMP compiler using different sets of flags. A log file is necessary to save information when a compiler error occurs. Once a particular directive or set of directives is selected for testing, we will look up an *information file* that contains details of the separate tests and test sequences. This file includes a table with directive names, function names and file names. *Function names* refer to the functions that are implemented to test the directives, and *file name* refers to the files that include these functions. This information file may also contain addition information, especially for human understanding. After compilation, each test is executed independently and the outputs are written to a file. We will include a discussion of this environment in our presentation at EWOMP 2004.

4. Results

Table 2 shows a summary of the results with ten different compilers. A -1 indicates that the compiler failed the test. A positive number indicates the effectiveness of the cross checks. If the result is 40 it means that all test passed and 40 % of the cross checks failed. For this test we have chosen 20 iteration for the tests and cross checks.

The results show that the validation suite is very portable. The only exception is one compiler which refuses parts of the code. The compiler claims that the ‘Loop is not countable’, due to the constructs used to control the execution, like described in Sec. 3.3. In Table 2 the affected tests are indicated as failed.

5. Conclusion

In this paper we presented an OpenMP validation suite suitable to test an OpenMP implementation for the C and Fortran language binding. Compared to the previous version the suite was not only extended to cover Fortran, but a large number of tests have been added to cover the complete specification of OpenMP. Some of the added tests have only been possible with the added clarifications of OpenMP 2.5.

References

[1] Open64 compiler tools. `http://open64.sourceforge.net`.

[2] M. S. Müller and P. Neytchev. An OpenMP validation suite. In *Fifth European Workshop on OpenMP*, Aachen University, Germany, Sept. 2003.

[3] OpenMP Architecture Review Board. *OpenMP Specifications*. `http://www.openmp.org/specs`.

Compiler	1	2	3	4	5	6	7	8	9	10
check_has_openmp	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	100
for REDUCTION	100	100	100	100	100	100	100	100	100	100
for PRIVATE	-1	100	100	70	100	100	100	100	95	100
for FIRSTPRIVATE	0	100	100	100	100	55	100	100	100	0
for LASTPRIVATE	100	100	100	55	0	5	100	0	70	100
section PRIVATE	0	100	95	0	100	100	100	100	100	95
section FIRSTPRIVATE	100	100	100	100	100	100	100	95	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	40	0	45	15	10	0	0	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	100
parallel for REDUCTION	100	100	100	100	100	100	100	100	100	100
parallel for PRIVATE	0	100	100	55	100	100	100	100	100	100
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 PRIVATE	0	100	100	0	100	100	100	100	100	5
parallel section FIRSTPRIVATE	100	100	100	100	100	100	100	95	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	35	100	100	0	15	100	100	100	100	100
omp ATOMIC	60	95	100	0	35	100	100	100	100	5
omp BARRIER	100	100	100	100	100	100	-1	100	100	100
omp FLUSH	0	0	0	0	0	0	-1	0	-1	0
omp THREADPRIVATE	100	100	100	100	-1	100	-1	100	100	100
omp COPYIN	100	100	100	100	-1	95	-1	100	100	100
omp LOCK	-1	100	100	20	100	85	100	100	100	100
omp TESTLOCK	-1	100	100	0	100	90	100	100	100	100
omp NEST_LOCK	-1	100	100	65	100	65	100	100	100	100
omp NEST_TESTLOCK	-1	100	100	20	100	80	100	100	100	100
Summary	fail	pass	pass	pass	fail	pass	fail	pass	fail	pass

Table 2. Results of the cross checks. A -1 indicates the failure of a test.