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

August, 2004

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 presented. 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 [2] 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 [1] 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.

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).

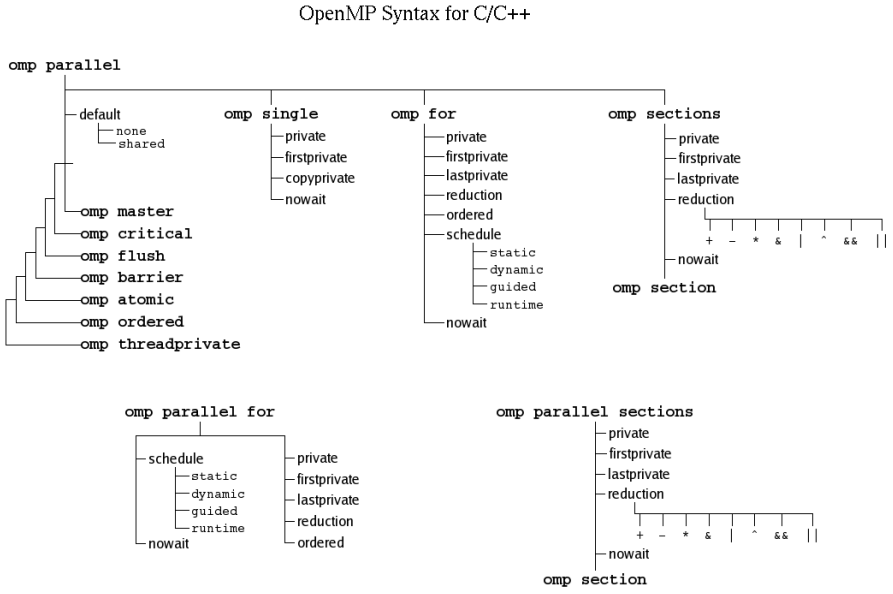

Figure 1: Overview of OpenMP directives and clauses.

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. 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. Tab. 1 contains a list of directives and their replacements.

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

3.1 Reductions

In the previous verion 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 (`+,-,*,,&,|,^,&&,||`) and for integer as well as floating point variables. The input data sets for the boolean operations are constructed to detect any omission 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.

3.2 Extension to Fortran

We realized the Fortran OpenMP validation suite by converting the original C version [1] 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 [1]. 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. 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. Fortran OpenMP 2.0 features are partially available. We plan to support the complete Fortran OpenMP 2.0 specification.

3.3 Schedule clauses

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. 3.4) 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.

3.4 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

The final paper will contain the results of applying our tests to 10 different compilers.

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.

The final paper will contain a much more extensive descriptions of the tests and a more detailed discussion of the clarifications contained in OpenMP 2.5. We hope that by the time of Ewomp 2004, the public comment period of OpenMP 2.5 has started. We are also working on a coverage analysis with the Open64 compiler to show that the suite covers the OpenMP relevant parts of the compiler as well as the runtime library.

References

[1] Matthias S. Müller and Pavel Neytchev. An OpenMP validation suite. In *Fifth European Workshop on OpenMP*, Aachen University, Germany, Sept. 2003.

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