Common Mistakes in OpenMP and How To Avoid Them

A Collection of Best Practices

Michael Süß and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies,
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
{msuess, leopold}@uni-kassel.de

Abstract. Few data are available on common mistakes made when using OpenMP. This paper presents a study on the programming errors observed in our courses on parallel programming during the last two years, along with numbers on which compilers and tools were able to spot them. The mistakes are explained and best practices for programmers are suggested to avoid them in the future. The best practices are presented in the form of an OpenMP checklist for novice programmers.

1 Introduction

One of the main design goals of OpenMP was to make parallel programming easier. Yet, there are still fallacies and pitfalls to be observed when novice programmers are trying to use the system. We have therefore conducted a study on a total of 85 students visiting our lecture on parallel programming, and observed the mistakes they made when asked to prepare assignments in OpenMP. The study is described in detail in Sect. 2.

We are concentrating on the most common mistakes from our study for the rest of this paper. They are briefly introduced in Tab. 1, along with a count of how many teams (consisting of two students each) have made the mistake each year. We have chosen to divide the programming mistakes into two categories:

1. *Correctness Mistakes:* all errors impacting the correctness of the program.
2. *Performance Mistakes:* all errors impacting the speed of the program. These lead to slower programs, but do not produce incorrect results.

Sect. 3 explains the mistakes in more detail. Also in this section, we propose possible ways and best practices for novice programmers to avoid these errors in the future. Sect. 4 reports on tests that we conducted on a variety of OpenMP-compilers to figure out, if any of the programming mistakes are spotted and/or possibly corrected by any of the available compilers. In Sect. 5, all suggestions made this far are condensed into an OpenMP programming checklist, along with others from our own programming experiences. Sect. 6 reviews related work, while Sect. 7 sums up our results.

No.	*Problem*	*2004*	*2005*	*Sum*
	Correctness Mistakes			
1.	Access to shared variables not protected	8	10	18
2.	Use of locks without `flush`	7	11	18
3.	Read of shared variable without `flush`	5	10	15
4.	Forget to mark `private` variables as such	6	5	11
5.	Use of `ordered` clause without `ordered` construct	2	2	4
6.	Declare loop variable in `#pragma omp parallel for` as `shared`	1	2	3
7.	Forget to put down `for` in `#pragma omp parallel for`	2	0	2
8.	Try to change num. of thr. in parallel reg. after start of reg.	0	2	2
9.	`omp_unset_lock()` called from non-owner thread	2	0	2
10.	Attempt to change loop variable while in `#pragma omp for`	0	2	2
	Performance Mistakes			
11.	Use of `critical` when `atomic` would be sufficient	8	1	9
12.	Put too much work inside `critical` region	2	4	6
13.	Use of orphaned construct outside parallel region	2	2	4
14.	Use of unnecessary `flush`	3	1	4
15.	Use of unnecessary `critical`	2	0	2
	Total Number of Groups	26	17	43

Table 1. The list of frequently made mistakes when programming in OpenMP

2 Survey Methodology

We have evaluated two courses for this study. Both consisted of students on an undergraduate level. The first course took place in the winter term of 2004 / 2005, while the second one took place in the winter term of 2005 / 2006. The first course had 51 participants (26 groups of mostly two students), the second one had 33 participants (17 groups). The lecture consisted of an introduction to parallel computing and parallel algorithms in general, followed by a short introduction of about five hours on OpenMP. Afterwards, the students were asked to prepare programming assignments in teams of two people, which had to be defended before the authors. During these sessions (and afterwards in preparation for this paper), we analyzed the assignments for mistakes, and the ones having to do with OpenMP are presented in this paper.

The assignments consisted of small to medium-sized programs, among them:

- find the first N prime numbers
- simulate the dining philosophers problem using multiple threads
- count the number of connected components in a graph
- write test cases for OpenMP directives / clauses / functions

A total of 231 student programs in C or C++ using OpenMP were taken into account and tested on a variety of compilers (e. g. from SUN, Intel, Portland

Group, IBM, as well as on the free OMPi compiler). Before we begin to evaluate the results, we want to add a word of warning: Of course, the programming errors presented here have a direct connection to the way we taught the lecture. Topics we talked about in detail will have led to fewer mistakes, while for other topics, the students had to rely on the specification. Moreover, mistakes that have been corrected by the students before submitting their solution are not taken into account here. For these reasons, please take the numbers presented in Tab. 1 as what they are - mere indications of programming errors that novice programmer might make.

3 Common Mistakes in OpenMP and Best Practices to Avoid Them

In this section, we will discuss the most frequently made mistakes observed during our study, as well as suggest possible solutions to make them occur less likely. There is one universal remark for instructors that we want to discuss beforehand: We based our lecture on assignments and personal feedback, and found this approach to be quite effective: As soon as we pointed out a mistake in the students programs during the exam, a group would rarely repeat it again. Only showing example programs in the lecture and pointing out possible problems did not have the same effect.

There are some mistakes, where we cannot think of any best practises to avoid the error. Therefore, we will just shortly sketch these at this point, while all other mistakes are discussed in their own section below (the number before the mistake is the same as in Tab. 1):

2. *Use of locks without flush*: Before version 2.5 of the OpenMP specification, lock operations did not include a `flush`. The compilers used by our students were not OpenMP 2.5 compliant, and therefore we had to mark a missing `flush` directive as a programming error.
5. *Use of ordered clause without ordered construct*: The mistake here is to put an `ordered` clause into a `for` worksharing construct, without specifying with a separate `ordered` clause inside the enclosed `for` loop, what is supposed to be carried out in order.
8. *Try to change number of threads in parallel region after start of region*: The number of threads carrying out a parallel region can only be changed before the start of the region. It is therefore a mistake to attempt to change this number from inside the region.
10. *Attempt to change loop variable while in #pragma omp for*: It is explicitly forbidden in the specification to change the loop variable from inside the loop.
11. *Use of critical when atomic would be sufficient*: There are special cases when synchronisation can be achieved with a simple `atomic` construct. Not using it in this case leads to potentially slower programs and is therefore a performance mistake.

13. *Use of orphaned construct outside parallel region*: When using a combined worksharing construct, sometimes our students would forget to put down the `parallel`, producing an orphaned construct. In other cases, the parallel region was forgotten altogether, leading e. g. to orphaned `critical` constructs.
14. *Use of unnecessary flush*: `flush` constructs are implicitly included in certain positions of the code by the compiler. Explicitly specifying a `flush` immediately before or after these positions is considered a performance mistake.
15. *Use of unnecessary critical*: The mistake here is to protect memory accesses with a `critical` construct, although they need no protection (e.g. on private variables or on other occasions, where only one thread is guaranteed to access the location).

3.1 Access to Shared Variables Not Protected

The most frequently made and most severe mistake during our study was to not avoid concurrent access to the same memory location. OpenMP provides several constructs for protecting critical regions, such as the `critical` construct, the `atomic` construct and locks. Although all three of these constructs were introduced during the lecture, many groups did not use them at all, or forgot to use them on occasions. When asked about it, most of them could explain what a critical region was for and how to use the constructs, yet to spot these regions in the code appears to be difficult for novice parallel programmers.

A way to make novice programmers aware of the issue is to use the available tools to diagnose OpenMP programs. For example, both the Intel Thread Checker and the Assure tool find concurrent accesses to a memory location.

3.2 Read of Shared Variable without Flush

The OpenMP memory model is a complicated beast. Whole sections in the OpenMP specification have been dedicated to it, as well as a whole paper written about it [1]. One of its complications is the error described here. Simply put, when reading a shared variable without flushing it first, it is not guaranteed to be up to date. Actually, the problem is even more complicated, as not only the reading thread has to flush the variable, but also any thread writing to it beforehand. Many students did not realize this and just read shared variables without any further consideration. On many common architectures this will not be a problem, because flushes are carried out frequently there. Quite often, the problem does not surface in real-world programs, because there are implicit flushes contained in many OpenMP constructs, as well.

In other cases, students simply avoided the problem by declaring shared variables as `volatile`, which puts an implicit `flush` before every read and after every write of any such variable. Of course, it also disables many compiler optimizations for this variable and therefore is often the inferior solution.

The proper solution, of course, is to make every OpenMP programmer aware of this problem, by clearly stating that every read to a shared variable must be preceded by a `flush`, except in very rare edge-cases not discussed here. This

`flush` can be explicitly written down by the programmer, or it can be implicit in an OpenMP construct.

Version 2.5 of the OpenMP specification includes a new paragraph on the memory model. Whether or not this is enough to make novice programmers aware of this pitfall remains to be seen.

3.3 Forget to Mark Private Variables as such

This programming error has come up surprisingly often in our study. It was simply forgotten to declare certain variables as private, although they were used in this way. The default sharing attribute rules will make the variable shared in this case.

Our first advice to C and C++ programmers to avoid this error in the future is to use the scoping rules of the language itself. C and C++ both allow variables to be declared inside a parallel region. These variables will be private (except in rare edge cases described in the specification, e. g. static variables), and it is therefore not necessary to explicitly mark them as such, avoiding the mistake altogether.

Our second advice to novice programmers is to use the `default(none)` clause. It will force each variable to be explicitly declared in a data-sharing attribute clause, or else the compiler will complain. We will not go as far as to suggest to make this the default behaviour, because it certainly saves the experienced programmer some time to not have to put down each and every shared variable in a shared clause. But on the other hand, it would certainly help novice programmers who probably do not even know about the `default` clause.

It might also help if the OpenMP compilers provided a switch for showing the data-sharing attributes for each variable at the beginning of the parallel region. This would enable programmers to check if all their variables are marked as intended. An external tool for checking OpenMP programs would be sufficient for this purpose as well.

Another solution to the problem is the use of autoscoping as proposed by Lin et al. [2]. According to this proposal, all data-sharing attributes are determined automatically, and therefore the compiler would correctly privatize the variables in question. The proposed functionality is available in the Sun Compiler 9 and newer.

Last but not least, the already mentioned tools can detect concurrent accesses to a shared variable. Since the wrongly declared variables fall into this category, these tools should throw a warning and alert the programmer that something is wrong.

3.4 Declare Loop Variable in #pragma omp parallel for as Shared

This mistake shows a clear misunderstanding of the way the `for` worksharing construct works. The OpenMP specification states clearly that these variables are implicitly converted to private, and all the compilers we tested this on performed

the conversion. The surprising fact here is that many compilers did the conversion silently, ignoring the shared declaration and not even throwing a warning. More warning messages from the compilers would certainly help here.

3.5 Forget to Put down for in #pragma omp parallel for

The mistake here is, to attempt to use the combined worksharing construct `#pragma omp parallel for`, but forget to put down the `for` in there. This will lead to every thread executing the whole loop, and not only parts of it as intended by the programmer.

In most cases, this mistake will lead to the mistake specified in Sect. 3.1, and therefore can be detected and avoided by using the tools specified there.

One way to avoid the mistake altogether is to specify the desired `schedule` clause, when using the `for` worksharing construct. This is a good idea for portability anyways, as the default `schedule` clause is implementation defined. It will also lead to the compiler detecting the mistake we have outlined here, as `#pragma omp parallel schedule(static)` is not allowed by the specification and yields compiler errors.

3.6 omp_unset_lock() Called from Non-Owner Thread

The OpenMP-specification clearly states:

> *The thread which sets the lock is then said to own the lock. A thread which owns a lock may unset that lock, returning it to the unlocked state. A thread may not set or unset a lock which is owned by another thread.*
> ([3, p. 102])

Some of our students still made the mistake to try to unlock a lock from a non-owner thread. This will even work on most of the compilers we tested, but might lead to unspecified behaviour in the future.

To avoid this mistake, we have proposed to our students to use locks only when absolutely necessary. There are cases when they are needed (for example to lock parts of a variable-sized array), but most of the times, the `critical` construct provided by OpenMP will be sufficient and easier to use.

3.7 Put too much Work inside Critical Region

This programming error is probably due to the lack of sensitivity for the cost of a critical region found in many novice programmers. The issue can be split into two subissues:

1. Put more code inside a critical region than necessary, thereby potentially blocking other threads longer than needed.
2. Go through the critical region more often than necessary, thereby paying the maintenance costs associated with such a region more often than needed.

The solution to the first case is obvious: The programmer needs to check if each and every line of code that is inside a critical region really needs to be there. Complicated function calls, for example, have no business being in there most of the time, and should be calculated beforehand if possible.

As an example for the second case, consider the following piece of code, which some of our students used to find the maximum value in an array:

```
1  #pragma omp parallel for
2  for (i = 0; i < N; ++i) {
3    #pragma omp critical
4    {
5      if (arr[i] > max) max = arr[i];
6    }
7  }
```

The critical region is clearly in the critical path in this version, and the cost for it therefore has to be paid N times. Now consider this slightly improved version:

```
1  #pragma omp parallel for
2  for (i = 0; i < N; ++i) {
3    #pragma omp flush (max)
4    if (arr[i] > max) {
5      #pragma omp critical
6      {
7        if (arr[i] > max) max = arr[i];
8      }
9    }
10 }
```

This version will be faster (at least on architectures, where the `flush` operation is significantly faster than a critical region), because the critical region is entered less often. Finally, consider this version:

```
1  #pragma omp parallel
2  {
3    int priv_max;
4    #pragma omp for
5    for (i = 0; i < N; ++i) {
6      if (arr[i] > priv_max)  priv_max = arr[i];
7    }
8    #pragma omp flush (max)
9    if (priv_max > max) {
10     #pragma omp critical
11     {
12       if (priv_max > max) max = priv_max;
13     }
14   }
15 }
```

This is essentially a reimplementation of a reduction using the `max` operator. We have to resort to reimplementing this reduction from scratch here, because reductions using the `max` operator are only defined in the Fortran version of OpenMP (which in itself is a fact that many of our students reported to have caused confusion). Nevertheless, it is possible to write programs this way, and

No.	*File*	*icc*	*pgcc*	*sun*	*guide*	*xlc*	*ompi*	*assure*	*itc*
	Correctness Mistakes								
1.	access_shared	-	-	-	-	-	-	eE	eE
2.	locks_flush	-	-	-	-	-	-	-	-
3.	read_shared_var	-	-	-	-	-	-	-	(eE)
4.	forget_private (=access_shared)	-	-	-	-	-	-	eE	eE
5.	ordered_without_ordered	-	-	-	-	-	-	-	eW
6.	shared_loop_var	cE	cC	cW+C	cE	cC	cC	cE	cE
7.	forget_for	-	-	-	-	-	-	(eE)	(eE)
8.	change_num_threads	-	-	-	-	-	-	-	-
9.	unset_lock_diff_thread	-	-	-	-	-	-	-	-
10.	change_loop_var	-	-	-	-	cW	-	-	-
	Performance Mistakes								
11.	crit_when_atomic	-	-	-	-	-	-	-	-
12.	too_much_crit (no test!)								
13.	orphaned_const	-	-	rW	-	-	-	-	-
14.	unnec_flush	-	-	-	-	-	-	-	-
15.	unnec_crit	-	-	-	-	-	-	-	-

Table 2. How Compilers deal with the Problems

by showing novice programmers techniques like the ones sketched above, they get more aware of performance issues.

4 Compilers and Tools

There are a multitude of different compilers for OpenMP available, and we wanted to know, if any of them were able to detect the programming errors sketched in Sect. 3. Therefore we have written a short testcase for each of the programming mistakes. Tab. 2 describes the results of our tests on different compilers.

The numbers in the first column are the same as in Tab. 1. The second column contains the names of our test programs. We could not think of a sound test for problem 12 (put too much work inside `critical` region), and therefore the results for this problem are omitted. Test program four is the same as test program one, and therefore the results are the same as well. The rest of the table depicts results for the following compilers (this list is not sorted by importance, nor in any way representative, but merely includes all the OpenMP-compilers we had access to):

- Intel Compiler 9.0 (icc)
- Portland Group Compiler 6.0 (pgcc)
- Sun Compiler 5.7 (sun)
- Guide component of the KAP/Pro Toolset C/C++ 4.0 (guide)

– IBM XL C/C++ Enterprise Edition 7.0 (xlc)
– OMPi Compiler 0.8.2 (ompi)
– Assure component of the KAP/Pro Toolset C/C++ 4.0 (assure)
– Intel Thread Checker 2.2 (itc)

The last two entries (assure and itc) are not compilers, but tools to help the programmer find mistakes in their OpenMP programs. As far as we know, Assure was superseded by the Intel Thread Checker and is no longer available, nevertheless it is still installed in many computing centers. We were not able to find any lint-like tools to check OpenMP programs in C, there are however solutions for Fortran available commercially.

The alphabetic codes used in the table are to be read as follows: the first (uncapitalized) letter is one of (c)ompiletime, (r)untime or (e)valuation time, and describes, when the mistake was spotted by the compiler. Only Assure and the Intel Thread Checker have an evaluation step after the actual program run. The second (capitalized) letter describes, what kind of reaction was generated by the compiler, and is one of the following: (W)arning, (E)rror or (C)onversion. Conversion in this context means that the mistake was fixed by the compiler without generating a warning. Conversion was done for problem six, where the compilers privatized the shared loop variable. W+C means, that the compiler generated a warning, but also fixed the problem at the same time. There is one last convention to describe in the alphabetic codes: When there are braces around the code, it means that a related problem was found by the program, which could be traced back to the actual mistake. An example: When the programmer forgets to put down `for` in a parallel worksharing construct (problem seven), it will lead to a data race. This race is detected by the Intel Thread Checker, and therefore the problem becomes obvious. All tests were performed with all warnings turned to the highest level for all compilers.

It is obvious from these numbers that most of the compilers observed are no big help in avoiding the problems described in this paper. Tools such as the Intel Thread Checker are more successful, but still it is most important that programmers avoid the mistakes in the first place. This paper and the programmers checklist presented in the next section are a step in this direction.

5 OpenMP Programmers Checklist

In this section, we summarize the advice given to novice programmers of OpenMP this far and rephrase it to fit into the easy to use format of a checklist. For this reason, the form of address is changed and the novice programmer is addressed directly. The checklist also contains other items, which we have accumulated during our own use of and experiences with OpenMP.

General

– It is tempting to use fine grained parallelism with OpenMP (throwing in an occasional `#pragma omp parallel for` before loops). Unfortunately, this

rarely leads to big performance gains, because of overhead such as thread creation and scheduling. You therefore have to search for potential for coarser-grained parallelism.
- Related to the point above, when you have nested loops, try to parallelize only the outer loop. Loop reordering techniques can sometimes help here.
- Use reduction where applicable. If the operation you need is not predefined, implement it yourself as shown in Sect. 3.7.
- Beware of nested parallelism, as many compilers still do not support it, and even if it is supported, nested parallelism may not give you any speed increases.
- When doing I/O (either to the screen or to a file), large time savings are possible by writing the information to a buffer first (this can sometimes even be done in parallel) and then pushing it to the device in one run.
- Test your programs with multiple compilers and all warnings turned on, because different compilers will find different mistakes.
- Use tools such as the Intel Thread Checker or Assure, which help you to detect programming errors and write better performing programs.

Parallel Regions

- If you want to specify the number of threads to carry out a parallel region, you must invoke `omp_set_num_threads()` *before* the start of that region (or use other means to specify the number of threads *before* entering the region).
- If you rely on the number of threads in a parallel region (e.g. for manual work distribution), make sure you actually get this number (by checking `omp_get_num_threads()` after entering the region). Sometimes, the runtime system will give you less threads, even when the dynamic adjustment of threads is off!
- Try to get rid of the `private` clause, and declare private variables at the beginning of the parallel region instead. Among other reasons, this makes your data-sharing attribute clauses more manageable.
- Use `default(none)`, because it makes you think about your data-sharing attribute clauses for all variables and avoids some errors.

Worksharing Constructs

- For each loop you parallelize, check whether or not every iteration of the loop has to do the same amount of work. If this is not the case, the static work schedule (which is often the default in compilers) might hurt your performance and you should consider dynamic or guided scheduling.
- Whatever kind of schedule you choose, explicitly specify it in the worksharing construct, as the default is implementation-defined!
- If you use `ordered`, remember that you always have to use both the `ordered` clause and the `ordered` construct.

Synchronisation

- If more than one thread accesses a variable and one of the accesses is a write, you must use synchronization, even if it is just a simple operation like $i = 1$. There are no guarantees by OpenMP on the results otherwise!
- Use `atomic` instead of `critical` if possible, because the compiler might be able to optimize out the `atomic`, while it can rarely do that for `critical`.
- Try to put as little code inside critical regions as possible. Complicated function calls, for example, can often be carried out beforehand.
- Try to avoid the costs associated with repeatedly calling critical regions, for instance by checking for a condition before entering the critical region.
- Only use locks when necessary and resort to the `critical` clause in all other cases. If you have to use locks, make sure to invoke `omp_set_lock()` and `omp_unset_lock()` from the same thread.
- Avoid nesting of critical regions, and if needed, beware of deadlocks.
- A critical region is usually the most expensive synchronisation construct (and takes about twice as much time to carry out as e. g. a barrier on many architectures), therefore start optimizing your programs accordingly — but keep in mind that these numbers only account for the time needed to actually perform the synchronisation, and not the time a thread has to wait on a barrier or before a critical region (which of course depends on various factors, among them the structure of your program or the scheduler).

Memory Model

- Beware of the OpenMP memory model. Even if you only read a shared variable, you have to flush it beforehand, except in very rare edge cases described in the specification.
- Be sure to remember that locking operations do not imply an implicit flush before OpenMP 2.5.

6 Related Work

We are not aware of any other studies regarding frequently made mistakes in OpenMP. Of course, in textbooks [4] and presentations teaching OpenMP, some warnings for mistakes are included along with techniques to increase performance, but most of the time, these are more about general pitfalls regarding parallel programming (like e. g. warnings to avoid deadlocks). There is one interesting resource to mention though: the blog of Yuan Lin [5], where he has started to describe frequently made errors with OpenMP. Interestingly, at the time of this writing, he has not touched any errors that we have described as well, which leads us to think that there are many more sources of errors hidden inside the OpenMP specification and that our OpenMP checklist is by no means all-embracing and complete.

7 Summary

In this paper, we have presented a study on frequently made mistakes with OpenMP. Students visiting the authors' courses on parallel programming have been observed for two terms to find out, which were their most frequent sources of errors. We presented 15 mistakes and recommendations for best practices to avoid them in the future. These best practices have been put into a checklist for novice programmers, along with some practices from the authors' own experiences. It has also been shown, that the OpenMP-compilers available today are not able to protect the programmer from making these mistakes.

Acknowledgments

We are grateful to Björn Knafla for proofreading the paper and for his insightful comments. We thank the University Computing Centers at the RWTH Aachen, TU Darmstadt and University of Kassel for providing the computing facilities used to test our sample applications on different compilers and hardware. Last but not least, we thank the students of our courses, without whom it would have been impossible for us to look at OpenMP from a beginners perspective.

References

1. Hoeflinger, J.P., de Supinski, B.R.: The OpenMP memory model. In: Proceedings of the First International Workshop on OpenMP - IWOMP 2005. (2005)
2. Lin, Y., Copty, N., Terboven, C., an Mey, D.: Automatic scoping of variables in parallel regions of an OpenMP program. In: Proceedings of the Workshop on OpenMP Applications & Tools - WOMPAT 2004. (2004)
3. OpenMP Architecture Review Board: OpenMP specifications. `http://www.openmp.org/specs` (2005)
4. Chandra, R., Dagum, L., Kohr, D.: Parallel Programming in OpenMP. Morgan Kaufmann Publishers (2000)
5. Lin, Y.: Reducing the complexity of parallel programming. `http://blogs.sun.com/roller/page/yuanlin` (2005)