Problems, Workarounds and Possible Solutions Implementing the Singleton Pattern with C++ and OpenMP

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. Programs written in C++ and OpenMP are still relatively rare. This paper shows some problems with the current state of the OpenMP specification regarding C++. We illustrate the problems with various implementations of the singleton pattern, measure their performance and describe workarounds and possible changes to the specification. The singletons are available in the AthenaMP open-source project.

1 Introduction

Programming with OpenMP and C++ is still challenging today. With present compilers, even basic C++ features such as exceptions often do not work. To provide more test cases and experiment with advanced C++ features in combination with OpenMP, the AthenaMP [1] open source project was created. This work is part of AthenaMP.

One of the most widely known patterns among programmers is the *singleton*. It has been thoroughly analyzed and implemented. This paper deploys thread-safe singletons as an example to show how OpenMP and C++ can be used together. Several implementations are described in depth, which is our first contribution. The second contribution is pointing out problems with the use of OpenMP in combination with C++. We provide workarounds for these problems and suggest changes to the OpenMP specification to solve them fully.

The paper is organized as follows: In Section 2, the singleton pattern is introduced, along with a simple, non-threaded implementation in C++. Section 3 describes various thread-safe implementations, highlights problems with OpenMP and explains possible workarounds, while their performance is benchmarked in Section 4. Some implementations require changes to the OpenMP specification, which are collected in Section 5. The work of this paper is embedded in the AthenaMP open-source project described in Section 6. The paper finishes with related work in Section 7, and a summary in Section 8.

2 The Singleton Pattern

The most famous description of the intent of the singleton pattern is from Gamma et al. [2]:

Ensure a class only has one instance, and provide a global point of access to it. (Gamma et al. [2])

Singletons are useful to ensure that only one object of a certain type exists throughout the program. Possible applications are a printer spooler or the state of the world in a computer game. While singletons provide a convenient way to access a resource throughout the program, one needs to keep in mind that they are not much more than glorified global variables, and just like them need to be used with care (or not at all, if possible). Yegge explains this in great detail in one of his weblog posts [3].

While singletons can be implemented in C++ using inheritance, we have decided to implement them using wrappers and templates, as described by Schmidt et al. [4]. These authors call their classes adapters, but we stick to the more general name *singleton wrapper* instead, to avoid confusion with the well-known adapter pattern (which is different from what we are doing). We provide wrapper classes that can be instantiated with any class to get the singleton functionality. For example, to treat the class `my_class` as a singleton and call the method `my_method` on it, one needs to do the following:

```
singleton_wrapper<my_class>::instance().my_method();
```

Provided that all accesses to the class `my_class` are carried out in this way, then and only then the class is a singleton. The code of class `my_class` does not need to be changed in any way, but all of the singleton functionality is provided by the wrappers. This is the biggest advantage of this implementation over an inheritance-based approach. Typical requirements for an implementation state that the singleton:

- must not require prior initialization, because forgetting to do so is a frequent mistake by programmers
- must be initialized only when it is needed (lazy initialization)
- must return the same instance of the protected object, regardless of whether it is called from within or outside a parallel region

A non-threadsafe version of the `instance` method of a singleton wrapper is shown in Figure 1. Here, `instance_` is a private static member variable, omitted for brevity in all of our examples. This implementation, of course, has problems in a multi-threaded environment, as the `instance_` variable is a shared resource and needs to be protected from concurrent access. Ways to deal with this problem are shown in the next section.

3 Thread-Safe Singleton Implementation Variants

A safe and simple version of a multi-threaded singleton wrapper is shown in Figure 2. It uses a named critical region to protect access to the singleton instance. While this solution solves the general problem of protecting access to

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  if (instance_ == 0) {
    instance_ = new Type;
  }
  return *instance_;
}
};
```

Fig. 1. The `instance` method of a sequential singleton wrapper

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp critical (ATHENAMP_1)
  {
    if (instance_ == 0) {
      instance_ = new Type;
    }
  }
  return *instance_;
}
};
```

Fig. 2. The `instance` method of a simple, thread-safe singleton wrapper

the singleton class, it has two major drawbacks, both of which we are going to solve later: First, it uses the same named critical construct to protect all classes. If e. g. a singleton for a printer spooler and a singleton for the world state is needed in the same program, access to these classes goes through the same critical region and therefore these accesses can not happen concurrently. This restriction is addressed in Section 3.1. The second problem is that each access to the singleton has to pay the cost of the critical region, although technically it is safe to work with the singleton after it has been properly initialized and published to all threads. An obvious (but incorrect) attempt to solve this problem is shown in Section 3.2, while Sects. 3.3 and 3.4 solve the problem, but have other restrictions.

3.1 The Safe Version using One Lock per Protected Object

In the last section, we have shown a thread-safe singleton that used one critical region to protect accesses to all singletons in the program. This introduces unnecessary serialization, as a different critical region per singleton is sufficient. We will show four different ways to solve the problem. The first two require changes in the OpenMP specification to work but are quite simple from a programmer's point of view, the third can be done today but requires a lot of code. The fourth solution requires a helper-class and has problems with some compilers.

Attempt 1: Extending Critical: Our first attempt at solving the problem is shown in Figure 3. The idea is to give each critical region a unique name by using the template parameter of the singleton wrapper. Unfortunately, this idea does not work, because the compilers treat the name of the critical region as a string and perform no name substitution on it. While it would be theoretically feasible to change this in compilers, because template instantiation happens at compile-time, it would still require a change in the OpenMP specification, and we suspect the demand for that feature to be small. For this reason, we are not covering this attempt any further in Section 5.

```
// this code works, but does
// not solve the problem!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp critical(Type)
  {
    if (instance_ == 0) {
      instance_ = new Type;
    }
  }
  return *instance_;
}
};
```

Fig. 3. `instance` method with multiple critical regions - Attempt 1

```
// this code does not work!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static omp_lock_t my_lock
    = OMP_LOCK_INIT;
  omp_set_lock (&my_lock);
  if (instance_ == 0) {
    instance_ = new Type;
  }
  omp_unset_lock (&my_lock);
  return *instance_;
}
};
```

Fig. 4. `instance` method with multiple critical regions - Attempt 2

Attempt 2: Static Lock initialized with OMP_LOCK_INIT: Our second attempt to solve the problem uses OpenMP locks. The code employs a static lock to protect access to the shared instance variable. Since each instance of the template function is technically a different function, each instance gets its own lock as well, and therefore each singleton is protected by a different critical region.

The big problem with this approach is to find a way to initialize the lock properly. There is only one way to initialize a lock in OpenMP – by calling `omp_init_lock`. This must only be done once, but OpenMP does not provide a way to carry out a piece of code only once (a solution to this shortcoming is presented later in this section). One of our requirements for the singleton is that it must not need initialization beforehand, therefore we are stuck.

A solution to the problem is shown in Figure 4 and uses static variable initialization to work around the problem, by initializing the lock with the constant `OMP_LOCK_INIT`. This is adapted from POSIX Threads, where a mutex can be initialized with `PTHREAD_MUTEX_INITIALIZER`. In a thread-safe environment (which OpenMP guarantees for the base language), the runtime system should make sure this initialization is carried out only once, and all the compilers we have tested this with actually do so. Of course, `OMP_LOCK_INIT` is not in the OpenMP specification, but we believe it would be a worthy addition to solve this and similar problems, not only related to singletons. Therefore, although quite an elegant solution, this attempt does not work with OpenMP today as well.

Attempt 3: Doing it Once: As shown in the previous paragraph, method `omp_init_lock` needs to be called only once, but OpenMP provides no facilities to achieve that. It is possible to code this functionality in the program itself, though, as one of the authors has described in his weblog [5]. The necessary methods to achieve this are available and described in the AthenaMP library [1] as well as

```
1  struct init_func {
2    omp_lock_t *lock;
3    void operator() ()
4    {
5      omp_init_lock (lock);
6    }
7  };
8
9  template <class Type>
10 class singleton_wrapper {
11 static Type& instance ()
12 {
13   static omp_lock_t my_lock;
14   static once_flag flag
15     = ATHENAMP_ONCE_INIT;
16   init_func my_func;
17   my_func.lock = &my_lock;
18
19   once (my_func, flag);
20
21   omp_set_lock (&my_lock);
22   if (instance_ == 0) {
23     instance_ = new Type;
24   }
25   omp_unset_lock (&my_lock);
26   return *instance_;
27 }
28 };
```

Fig. 5. `instance` method with multiple critical regions - Attempt 3

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static omp_lock_ad my_lock;
  my_lock.set();
  if (instance_ == 0) {
    instance_ = new Type;
  }
  my_lock.unset();
  return *instance_;
}
};
```

Fig. 6. `instance` method with multiple critical regions - Attempt 4

the above-mentioned weblog entry. Here, we restrict ourselves to showing how the functionality can be used.

Figure 5 depicts the `instance` method with once functionality. Line 19 has the actual call to the `once` template-function defined in AthenaMP. The function takes two parameters, the first one being a functor that includes what needs to happen only once (also shown at the top of Figure 5). The second parameter is a flag of the type `once_flag` that is an implementation detail to be handled by the user.

This attempt to solve our problem works. Nevertheless, it needs a lot of code, especially if you also count the code in the `once` method. Moreover, it relies on static variable initialization being thread-safe, as our last attempt. For these reasons, we are going to solve the problem one last time.

Attempt 4: Lock Adapters to the Rescue: Figure 6 shows our last attempt at protecting each singleton with its own critical region. It is substantially smaller than all previous versions and should also work with OpenMP today. It uses lock adapters (called `omp_lock_ad`) that are part of AthenaMP and are first introduced in a different publication by the same authors [6].

Lock adapters are small classes that encapsulate the locking functionality provided by the parallel programming system. In AthenaMP, two of these classes are included, `omp_lock_ad` for simple OpenMP locks and `omp_nest_lock_ad` for nested OpenMP locks. Lock adapters for different parallel programming systems

are trivial to implement. The initialization code for the locks is in the constructors of the adapter classes, therefore their initialization happens automatically. Destruction of the encapsulated lock is done in the destructor of the adapter. A common mistake in OpenMP (forgetting to initialize a lock before using it) is therefore not possible when using lock adapters. For more details about the lock adapters please refer to another publication by the same authors [6].

Because lock initialization happens automatically when an instance of the class `omp_lock_ad` is created, our problem with the initialization having to be carried out only once disappears. Or at least: it should disappear. Unfortunately, some compilers we have tested this with call the constructor of the lock adapter more than once in this setting, although it is a shared object. The next paragraph explains this more thoroughly.

The difference between static variable construction and static variable initialization becomes important here, because static variable initialization works correctly in a multi-threaded setting with OpenMP on all compilers we have tested, but static variable construction has problems on some compilers. Static variable initialization basically means declaring a variable of a primitive data type and initializing it at the same time:

```
static int my_int = 5;
```

This makes `my_int` a shared variable and as in the sequential case, it's initialization is carried out once. This works on all compilers we have tested. Static variable construction on the other hand looks like this:

```
static my_class_t my_class;
```

Since `my_class_t` is a user-defined class, it's constructor is called at this point in the program. Since the variable is static, it is a shared object and therefore the constructor should only be called once. This does not happen on all compilers we have tested (although we believe this to be a bug in them, since OpenMP guarantees thread-safety of the base language) and is therefore to be used with care. A trivial test program to find out if your compiler correctly supports static variable construction with OpenMP is available from the authors on request.

This solution is the most elegant and also the shortest one to provide each singleton with its own critical region. Nevertheless, our second problem is still there: each access to the singleton has to go through a critical region, even though only the creation of the singleton needs to be protected. We are going to concentrate on this problem in the next section.

3.2 Double-Checked Locking and Why it Fails

Even in some textbooks, the *double-checked locking* pattern is recommended to solve the problem of having to go through a critical region to access a singleton [4]. Our `instance` method with this pattern is shown in Figure 7.

```
// this code is wrong!!
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  #pragma omp flush(instance_)
  if (instance_ == 0) {
    #pragma omp critical(A_2)
    {
      if (instance_ == 0)
        instance_ = new Type;
    }
  }
  return instance_;
}
};
```

Fig. 7. `instance` method with double-checked locking

```
template <class Type>
class singleton_wrapper {
static Type* cache_;
static Type& instance ()
{
  #pragma omp threadprivate(cache_)
  if (cache_ == 0) {
    cache_ =
      &singleton<Type>::instance();
  }
  return *cache_;
}
};
```

Fig. 8. `instance` method using caching

Unfortunately, the pattern has multiple problems, among them possible instruction reorderings by the compiler and missing memory barriers, as explained by Meyers and Alexandrescu [7], as well as problems with the OpenMP memory model, as explained by de Supinski on the OpenMP mailing list [8]. Since the pattern may and will fail in most subtle ways, it should not be employed.

3.3 Using a Singleton Cache

Meyers and Alexandrescu [7] also suggest caching a pointer to the singleton in each thread, to avoid hitting the critical region every time the `instance` method is called. This can of course be done by the user, but we wanted to know if it was possible to extend our singleton wrapper to do this automatically. We therefore came up with the implementation shown in Figure 8.

The implementation solves the problem, as the critical region for each singleton is only entered once. It relies on declaring a static member variable threadprivate. Unfortunately, the OpenMP specification does not allow to privatize static member variables in this way. In our opinion, this is an important omission not restricted to singletons, and therefore this point is raised again later in this paper, when we put together all enhancement proposals for the OpenMP specification in Section 5.

There is a workaround, however, which was suggested by Meyers in his landmark publication Effective C++ [9] as a solution to a different problem. Instead of making the cache a static member variable (that cannot be declared threadprivate), it can be declared as a static local variable in the `instance` method. Declaring local variables threadprivate is allowed by the specification, and this solves the problem without any further disadvantages.

The whole solution unfortunately has some problems. It relies on threadprivate data declared with the `threadprivate` directive, but in OpenMP these have some restrictions. In a nutshell, these data become undefined as soon as nested parallelism is employed or as soon as the number of threads changes over

```
template <class Type>
class singleton_wrapper {
static Type& instance ()
{
  static Type instance;
  return instance;
}
};
```

Fig. 9. `instance` method using a Meyers singleton

the course of multiple parallel regions (see the OpenMP specification for details [10]). There is no way to work around these limitations for this solution, therefore the user has to be made aware of them by carefully documenting the restrictions. Fortunately, there is a different way to achieve the desired effect and it is explained in the next section.

3.4 The Meyers Singleton

One of the most well-known singleton implementations today is the so-called Meyers Singleton that is described in Effective C++ [9]. It is quite elegant, small, and shown in Figure 9.

Meyers himself warns about using his implementation in a multi-threaded setting, because it relies on static variable construction being thread-safe. Of course, Meyers did not write about OpenMP. OpenMP guarantees thread-safety of the base language in the specification, and this should cover proper construction of static variables in a multi-threaded setting as well. Unfortunately, as described in Section 3.1 in detail, our tests show that some OpenMP-aware compilers still do have problems with this. Some of them were even calling the constructor of the same singleton twice, which should never happen in C++. Therefore, although it is the smallest and most elegant solution to the problem, it cannot be recommended for everyone at this point in time.

4 Performance

This section discusses the performance of the proposed singleton wrapper implementations. We have setup a very simple test to access the performance of our solution. It is shown in Figure 10.

Two different singletons are used in the example, one is an integer and one is a double. The protected singletons will usually be classes, of course, but for our simple performance measurements primitive data-types will do. The singletons are initialized prior to the parallel region. Inside the parallel region, they are read only and their result is added up and tested outside the parallel region for correctness (not shown in the figure). The results of our tests are shown in Table 1.

Here is a short summary of the table headings:

```
int counter = 0;
double fcounter = 0.0;

double start = omp_get_wtime ();

singleton_wrapper<int>::instance() = 1;
singleton_wrapper<double>::instance() = 1.0;
#pragma omp parallel reduction(+:counter) reduction(+:fcounter)
{
  for (int i=0; i<numtries; ++i) {
    counter += singleton_wrapper<int>::instance();
    fcounter += singleton_wrapper<double>::instance();
  }
}

double end = omp_get_wtime ();
```

Fig. 10. The code used to benchmark our singletons

Test Environment	one_crit	multi_crit	local_cache	meyers	dcl
AMD, Intel Comp., 4 Thr.	**32.8**	**18.6**	**1.38**	**0.04**	1.46
Sun, Sun Comp., 8 Thr.	**176**	**182**	n.a.	0.14	0.62
IBM, IBM Comp., 8 Thr.	**72.7**	**71.7**	**0.34**	0.01	0.85

Table 1. Measured benchmark timings in seconds, best of three runs, 10.000.000 Singleton accesses per thread - only the entries printed in bold are correct and safe!

- **one_crit**: a singleton wrapper using the same critical region for all protected singletons (see Figure 2)
- **multi_crit**: a singleton wrapper using a different critical region for all protected singletons with a lock adapter (see Figure 6)
- **local_cache**: a singleton wrapper that caches a pointer to the singleton in threadprivate memory (see Figure 8)
- **meyers**: a singleton wrapper built after the Meyers Singleton (see Figure 9) - the numbers are only representative on the Intel Compiler, because the other two do not construct static classes correctly
- **dcl**: for comparison, we have also included the double-checked locking version (see Figure 7), although it is not safe to use!

As can be clearly seen by these numbers, the Meyers Singleton is to be preferred on all architectures, as it is the fastest by several orders of magnitude. We don't have any numbers for the version using a threadprivate cache on the Sun Compiler, as it was not able to translate our code. We believe this to be a bug in the compiler.

These results leave us with a disappointing situation: we have isolated a best solution (Meyers Singleton), the solution should be legal judging from the OpenMP specification, yet some compilers don't implement it correctly. While we cannot fix the compilers, what we can do at this point is provide a short test program that shows which compilers behave correctly and which do not. It is available from the authors on request.

5 OpenMP Enhancement Proposals for C++

In this section, we suggest enhancements to the OpenMP specification that we found useful during the course of our work on AthenaMP. Most of them have been sketched earlier in this publication already and are summarized here to have them all in one place. All of them are useful beyond our simple singleton implementations in our opinion.

Lock Initialization with OMP_LOCK_INIT: We have shown in Section 3.1 that it would make sense to initialize OpenMP locks not only by using the `omp_init_lock` function, but also with a macro, e. g. `OMP_LOCK_INIT`. This is adapted from POSIX Threads, where `PTHREAD_MUTEX_INITIALIZER` can be used to initialize a lock. If this idea was accepted into the standard, it would be possible to initialize static locks using static variable initialization, a facility that is guaranteed to happen only once in OpenMP.

Declaring Static Member Variables threadprivate: In Section 3.3 we have shown that there is benefit in allowing static member variables to be declared threadprivate. This would be a very simple change in the specification and has already been implemented in the Intel Compiler.

Flushing Reference Variables not allowed: Although it is not explicitly forbidden in the specification to flush reference variables, most compilers do not allow it either. A very common use case for reference variables is to pass parameters to functions by reference, either because they need to be changed inside the function or because the object is large and copying it would include a performance penalty. This is a recommended practice in many textbooks about C++. We have hit this problem when implementing the once functionality touched in Section 3.1, where we pass the second parameter to the `once` function by reference to const and would like to flush it inside the function.

The specification only allows to flush pointers and not pointees. This is a problem in our case, because reference variables are most likely implemented as pointers. The OpenMP specification should therefore explicitly allow the special case of flushing reference variables to be more conforming to recommended C++ practices.

6 AthenaMP

The functionality presented in this paper is part of the AthenaMP open source project [1]. The main goal of this project is to provide implementations for a set of concurrent patterns using OpenMP and C++. It includes both low-level patterns like advanced locks, and higher-level ones. The patterns demonstrate solutions to parallel programming problems as a reference for programmers, and additionally can be used directly as generic components. The code is also

useful for compiler vendors testing their OpenMP implementations against more involved C++ code, an area where many compilers today still have difficulties. A more extensive project description is provided by one of the authors in his weblog [11].

7 Related Work

Lots of work has been put into correctly implementing the singleton pattern. The most famous resource on the topic is the book by Gamma et al. [2]. The idea for our singleton wrapper is described by Schmidt et al. [4], along with double-checked locking that was later proved to be an anti-pattern by Meyers and Alexandrescu [7]. The idea for the singleton cache is also from the latter source. More involved descriptions of singletons with different properties in C++ are given by Alexandrescu [12]. Yegge describes most clearly why singletons should be used with care [3].

8 Summary and Contributions

This work features two main contributions. First, we have described and evaluated different implementation possibilities for a thread-safe singleton with C++ and OpenMP (Section 3), a work that has to our knowledge never been attempted using these systems before. Second, we have highlighted some problems encountered with the present OpenMP specification and C++. Among them are the inability to initialize static locks with a macro, the inability to declare static member variables threadprivate, and the problem of not being allowed to flush reference variables.

Acknowledgments

We are grateful to Björn Knafla for proofreading the paper and for guiding us through the dark corners of C++ when needed. We thank the University Computing Centers at the RWTH Aachen, TU Darmstadt and University of Kassel for providing the computing facilities used to carry out our unit tests on different compilers and architectures.

References

1. Süß, M.: AthenaMP. `http://athenamp.sourceforge.net/` (2006)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
3. Yegge, S.: Singleton considered stupid. `http://steve.yegge.googlepages.com/singleton-considered-stupid` (2004)
4. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects. John Wiley and Sons (2000)

5. Süß, M.: How to do it ONCE in OpenMP. http://www.thinkingparallel.com/2006/09/20/how-to-do-it-once-in-openmp/ (2006)
6. Süß, M., Leopold, C.: Generic locking and deadlock-prevention with C++. (2007) Proceedings of Parallel Computing (ParCo2007), to appear.
7. Meyers, S., Alexandrescu, A.: C++ and the perils of double-checked locking - part 1. Dr. Dobb's Journal (2004) 46–49
8. de Supinski, B.R.: [omp] A memory model question. http://openmp.org/pipermail/omp/2006/000479.html (2006)
9. Meyers, S.: Effective C++: 55 Specific Ways to Improve Your Programs and Designs. 3. edn. Addison-Wesley Professional (2005)
10. OpenMP Architecture Review Board: OpenMP specifications. http://www.openmp.org/specs (2005)
11. Süß, M.: A Vision for an OpenMP Pattern Library in C++. http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/ (2006)
12. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Professional (2001)