Implementing Irregular Parallel Algorithms with OpenMP

What's Missing and How to Solve It

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. Writing irregular parallel algorithms with OpenMP has been rarely practised in the past. Yet it is possible, and in this paper we will use a simple breadth–first search application as an example to show a possible stepping stone and deficiency of the OpenMP specification: It is very difficult to cancel the threads in a parallel region. A way to work around the issue within the existing OpenMP specification is sketched, while our main contribution is a proposal for an extension of OpenMP to solve the issue in an easier way.

1 Introduction

OpenMP[1] is a parallel programming system that aims to be powerful and easy to use, while at the same time allowing the programmer to write high performance programs. Its initial focus was on numerical applications involving loops written in Fortran or C/C++, but it includes the necessary constructs to deal with more kinds of parallel algorithms.

Irregular parallel algorithms involve subcomputations whose amount of work is not known in advance, and hence the work can only be distributed at runtime. Important subclasses include algorithms using taskpools, as well as speculative algorithms. We are concentrating on the first type, although the problem and solutions we present apply to other types as well. Examples for irregular algorithms are search and sorting algorithms, graph algorithms, and more involved applications like volume rendering.

According to Mattson [2], one of the initial designers of the OpenMP specification, OpenMP was never meant for irregular applications (where an irregular application in this context is one containing irregular algorithms as sketched above). Other people have tried to use OpenMP for this kind of applications, though, and have gotten mixed results [3–5]. This paper explores an important issue in developing irregular parallel algorithms with OpenMP, which is the missing ability to cancel threads in a parallel region. While a (not completely functional) workaround for the issue is suggested in Sect. 2, the main contribution of this paper is a proposal for new functionality to solve the problem in a convenient and easy to use way on the language level (in Sect. 3). The suggested

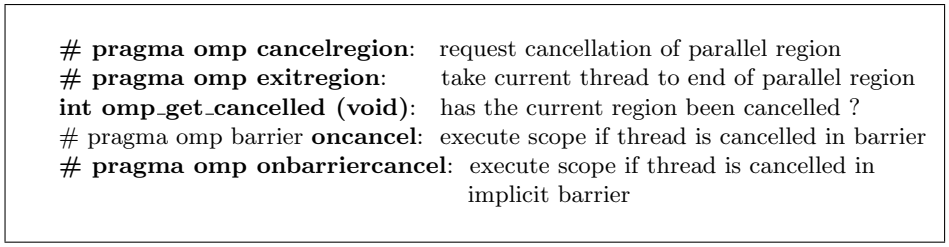

Fig. 1. Thread cancellation in a nutshell

additions to OpenMP are previewed in Fig. 1. A working implementation can be found in a special version of the OMPi compiler[6], which is available from the authors on request.

As a running example, we use breadth–first search on a labyrinth. The algorithm and its implementation are explained shortly in Sect. 2, where we will also explain why thread cancellation is a problem. Furthermore, a workaround for the issue is presented in this section, while a more advanced solution on the language level is described in Sect. 3. At the end of the paper, Sect. 4 summarizes our findings and shows some prospects for future work.

2 Problem Description

In labyrinth search, the objective is to find the shortest path through a labyrinth, from a given entry to a single exit. This problem is not merely a theoretical one, but has practical relevance e. g. for mapping electrical circuits on a chip. We consider a breadth-first search algorithm, which is not necessarily the fastest choice, but is simple enough to serve as an example here and to still include all the problems we want to illustrate. A very broad sketch of the algorithm is presented in pseudocode in Fig. 2.

The algorithm starts by putting the entry position of the labyrinth into the taskpool (not shown in the pseudocode). Afterwards, it spawns a parallel region (line 1). Then, one of the threads takes a position out of the taskpool (line 4), marks it on a map as processed (line 5), evaluates all neighbours by checking the four possible directions for walls (line 6), and checks if an exit is found on any of them. If no exit was found and the neighbour-positions have not been evaluated before (this check is not shown in our pseudocode), the neighbours are put into the taskpool to be processed in the next step (line 7), possibly by a different thread. If an exit is found, a flag is set that indicates this fact (line 9). We need to be careful with the different positions in the taskpool, since only positions with the same distance to the start should be evaluated together, or else the breadth-first search will degenerate. Therefore, only positions with the same distance to the entry are kept in the taskpool, while the neighbours are put into a different one (called next_taskpool). As soon as the taskpool is empty, both taskpools are switched by a single thread, and the computation proceeds

```
1  #pragma omp parallel
2  {
3    while (!exit_found) {
4      while ((task = pop (taskpool)) != NULL) && (!exit_found)) {
5        mark (labyrinth, task);
6        if (!inspect_field_for_exit (task))   // inspect all neighbours
7          push (neighbours (task), next_taskpool);  // no exit was found
8        else
9          exit_found = true;  // an exit was found
10       #pragma omp flush (exit_found)
11     }
12     #pragma omp barrier
13     #pragma omp single
14     {
15       taskpool = next_taskpool; // switch the taskpools
16       next_taskpool = NULL;
17     } // implicit barrier (includes flush)
18   }
19 } // end of parallel region with implicit barrier
```

Fig. 2. Parallel breadth first search, using a flag for thread cancellation

with the former next_taskpool (lines 15–16). When the algorithm depicted in Fig. 2 is done, a single thread follows the marks set in the labyrinth (line 5) from the exit point back to the entry point and identifies the shortest way.

In the figure, a flag is used to indicate when the threads in the parallel region should finish their work, because an exit was found (indicated by exit_found == true). We know of no other way in OpenMP to indicate that the threads should end their work in a parallel region. In Sect. 2.1, we will point to problems with this approach. Section 3 will present an extension of OpenMP that leads to an easier solution, which we will discuss in Sect. 3.5.

2.1 The Problem with Flags

When using flags to indicate that the parallel region should be aborted, great care has to be taken with checking these flags by the programmer. In our example, it might happen that one thread enters the while loop (line 3), finds an exit, sets the appropriate flag, and afterwards hangs in the barrier (line 12), because another thread does not enter the next iteration of the while loop at all, as the flag is indicating now that an exit was found! The program will exhibit undefined behaviour in this case (most likely a deadlock), because in OpenMP the sequence of barrier constructs encountered must be the same for every thread in the team. Thus, the code in Fig. 2 is not correct, and it is not safe to use without further adjustments that would make it even harder to read and explain!

Flags that indicate when a parallel region is to be cancelled give rise to yet another problem: Due to the OpenMP memory model, the flags have to be updated with a flush directive before their values are guaranteed to be up to date. This step is frequently missed by inexperienced OpenMP programmers [7]. The consequence is similar as sketched above: the program will potentially deadlock, because the thread which set the cancel flag has got its current correct value

and will exit the loop, whereas other threads might still use the old value and continue with it[1].

Let us summarize the problems we have identified so far with thread cancellation in OpenMP:

- there is no easy way to branch out of a parallel region, the only possible workaround is to use flags
- it is difficult to work with flags indicating that a region should end, at least as soon as barriers come into play
- if one forgets to flush a flag, a deadlock may arise

While we have presented a workaround for the main problem (flags manually set and checked by the programmer), it is still cumbersome and error-prone. Therefore we will present another possible solution in Sect. 3, based on a proposal to add thread cancellation to OpenMP. The proposal is also useful for the following common scenarios, which could benefit from thread cancellation:

- a cancel button from a user interface was pressed
- a solution has been found in a speculative algorithm

3 Thread Cancellation

This section shows a possible way to extend OpenMP with thread cancellation support. Sect. 3.1 shortly introduces a few basic terms often used when talking and writing about thread cancellation. An actual specification of the new functionality is given in Sect. 3.2, followed by the rationale for some of our design decisions in Sect. 3.3 and a short discussion on implementation and performance issues in Sect. 3.4. Sect. 3.5 puts the specification in perspective, by applying it to the labyrinth example.

3.1 Terms

We speak of *forceful cancellation* when a thread has the ability to cancel another thread from the outside. The cancelled thread may get the opportunity to clean up after itself, yet it does not have the power to decide when to be cancelled, nor to prevent cancellation at all. Asynchronous cancellation in POSIX Threads is an example of forceful cancellation. *Deferred cancellation* is an important subcase of asynchronous cancellation, in which the cancelled thread is not terminated immediately, but only at certain predefined cancellation points. Deferred cancellation is supported in POSIX Threads as well. With *cooperative cancellation*, in contrast, a thread can only ask for the cancellation of another thread. The cancelled thread has the opportunity to honor this request and cancel itself, to process the request at a later time, or even to ignore it altogether. Java threads support cooperative cancellation.

[1] This is not an issue in our example, as there is a flush included in many OpenMP directives (e. g. in the implicit barrier on line 17). Nevertheless, when the code is only slightly altered, the problem may surface.

3.2 Specification

The following directives to support cooperative thread cancellation in OpenMP are proposed:

#pragma omp cancelregion
This directive asks all threads in the team to stop their work and go to the end of the parallel region, where only the master thread will continue execution as usual. The emphasis here is on *asks*. The threads in the team are not cancelled immediately, but merely an internal cancel flag is set. The threads are not interrupted in any way and have to poll the flag using one of the directives described below. An exception is the thread that called the directive: it is cancelled immediately by an implicit call of the exitregion directive (explained below). Invoking the cancelregion directive on an already cancelled region has no effect except for the implicit call to exitregion. It is the task of the programmer to check if the cancel flag has been set, using a new OpenMP runtime library function:

int omp_get_cancelled (void)
This function returns 1 (true) if the cancellation of the enclosing parallel region was requested, and 0 (false) otherwise.

#pragma omp exitregion
This directive is not only useful for thread cancellation, but can be invoked at any point in a parallel region to immediately end the execution of the calling thread. This is accomplished by jumping to the end of the present parallel region, right into its closing implicit barrier (which is of course honored).

There is a problem with the proposal so far: barriers. If a region containing barriers is cancelled, at least one thread (the one calling the cancelregion directive) will never reach that barrier. Without further adjustment, one or more of the other threads in the region could hang in the barrier and never recover, since the barrier is not completed.

#pragma omp barrier oncancel
A solution to this problem is proposed in the form of the **oncancel** clause for the barrier directive. A new scope is optionally added to the barrier directive by specifying the oncancel clause. The commands in this scope are carried out only if the present parallel region has been or is being cancelled while the thread is waiting on the barrier. This can be seen on line 12 of Fig. 3.

It is now possible to use barriers in combination with thread cancellation. It remains the task of the programmer to do the right thing when a thread waiting on a barrier is cancelled, although most of the time he will just free the resources associated with the thread and exit the parallel region afterwards (using the newly proposed exitregion directive). Note that if the thread is not finalized with exitregion, it will hang in the barrier again (or phrased differently: there is an implicit barrier at the end of the oncancel clause). The reasons for this design decision are given in Sect. 3.3. The oncancel code is carried out at most once per barrier and thread. Furthermore, if the region is already cancelled when a thread enters the barrier, it will immediately proceed with the oncancel code.

For implicit barriers (at the end of worksharing constructs), a similar construct is proposed:

#pragma omp onbarriercancel

The usage of this directive is similar to the oncancel clause suggested above, except that onbarriercancel is a standalone construct and must be specified immediately after the implicit barrier it references. This is shown on line 21 of Fig. 3.

If the directive is present, all commands in its scope are carried out if the region is cancelled before or while the thread is waiting on the barrier. A nowait clause on the referenced worksharing construct and the onbarriercancel directive cannot be specified together. The directive also cannot be specified after a combined parallel worksharing construct (e. g. #pragma omp parallel for), the reasons for this design decision are also given in Sect. 3.3.

OpenMP allows for nested parallelism, i.e., when a member of a team inside a parallel region encounters a new parallel construct, a new subteam is formed. Our proposed extensions apply to nested parallelism as follows: Cancellation requests from inside the subteam only cause members of the subteam to have their cancellation flag set. If another member of the original team requests cancellation however, the cancellation flags for all members of all subteams are set as well, although technically they are not in the same team.

3.3 Rationale

Some of the suggested changes could be emulated manually by the experienced OpenMP programmer (such as keeping track of the cancel state of each thread). As has been explained in Sect. 2, this is, however, an unnecessary burden and gets difficult when barriers are involved at the latest. Therefore, our proposal introduces the new functionality on a language level.

The exitregion directive can be seen as a convenient shortcut, but even without thread cancellation, it is useful as soon as one gets into deeply nested functions inside parallel regions. It allows the programmer to jump to the end of the parallel region immediately, thereby potentially saving many lines of code of conditional statements. If barriers are involved in the parallel region, care has to be taken with exitregion for the reasons described in Sect. 3.2, or else the program might deadlock.

We have decided against forceful cancellation as in POSIX Threads. On one hand, asynchronous cancellation makes resource deallocation practically impossible. Since one never knows when a thread is cancelled, there is no place to put cleanup code. POSIX Threads solves this problem by utilizing cleanup stacks, but these are difficult to handle and keep track of. The concept of having cancellation points and deferred cancellation in OpenMP, on the other hand, seemed like overkill, as the amount of functions which are cancellation points is difficult to handle for programmers. Therefore, this proposal suggests cooperative cancellation, which can be found in a similar way e.g. in Java threads. Other good arguments for the use of cooperative cancellation can be found in the Java documentation [8].

A major problem with cooperative cancellation are the barrier constructs. The suggested solution (oncancel clause, onbarriercancel directive) may seem like a lot of overhead to cope with barriers, but the proposal is still easier and more natural than the possible alternatives (such as disallowing barriers with thread cancellation, putting the burden on the programmer to carefully work around them with flags, cancelling barriers forcefully).

We have also decided against automatically including an exitregion directive at the end of an oncancel or onbarriercancel scope. The main reason for this is consistency, as automatically including the directive would cancel the threads waiting on barriers forcefully. This would be inconsistent with the rest of the proposal, where cooperative cancellation is employed. Another reason is nested parallelism. We have specified in Sect. 3.2 that cancelling a parallel region will cancel all subregions as well. But as a subregion might be presently doing uninterruptible work and may contain barriers, the decision not to cancel on barriers automatically allows these subregions to complete their work when interrupted from threads in the upper parallel region, while properly shutting down when cancelled from inside their subregion.

The reason for not allowing the onbarriercancel directive after combined parallel worksharing constructs is that the two main reasons for applying the directive are not valid after a combined directive. There is no need to take care of left over threads hanging in the implicit barrier at the end of the combined construct, as these threads are exactly where they would be if an exitregion clause was specified. There is also no need to clean up any resources, as the programmer must have already done this before the end of the parallel region.

During our internal discussions on the topic of thread cancellation, we have worked out a checklist that each and every proposal we came up with had to pass. This checklist and some explanations of why our proposal passes it are spelled out here to make our design decisions yet more clear:

1. **Backwards Source Compatibility**
   Old code must run unchanged, when translated with a compiler that understands thread cancellation. This is the case, as the behaviour of existing OpenMP–constructs is not changed, but only new clauses or directives are added.
2. **Nested Parallelism**
   Each proposal must clearly state how thread cancellation and nested parallelism play together. Our proposal does so, by declaring that when a parallel region is cancelled, all parallel regions that were created by a thread from the cancelled region have their cancel flag set as well.
3. **Barriers**
   Each proposal must cope with the case that a region is cancelled while one or more threads are waiting on a barrier (including implicit barriers), without producing deadlocks. Our proposal does so with the introduction of the oncancel clause and the onbarriercancel directive.
4. **No Resource Leaks**
   The programmer must have the option to free any resources before a thread

is cancelled. Our proposal takes care of this by advocating cooperative cancellation, where the programmer checks if a cancellation request has been put up and can therefore deallocate / free all of his resources before exiting from a thread. Even resource deallocation while waiting on barriers is allowed with the introduction of the new oncancel clause and onbarriercancel directive.

5. **C / C++ / Fortran Compatibility**
   Each proposal must apply to all three supported languages of the OpenMP specification. Although our proposal only spells out the C syntax of the proposed changes, we believe that these are adaptable to C++ and Fortran as well.
6. **Simplicity**
   Each proposal must be as simple and easy to understand as possible, staying in line with the original OpenMP philosophy. Especially the barrier constructs made this a difficult task, but we think to have met that goal with the introduction of only three new directives, one new runtime library function and one new clause.

3.4 Implementation and Performance Issues

We have used the Ompi compiler [6] as a testing ground for our implementation. One of the benefits of employing cooperative cancellation is ease of implementation, and most of our changes were straightforward:

- adaptation of the compiler frontend to the new directives
- addition of new runtime library functions for exitregion, cancelregion, onbarriercancel and omp_get_cancelled
- a few more minor and locally restricted changes in the runtime library

The most difficult part was the implementation of exitregion, which must be able to jump out of deeply nested functions to the end of the parallel region. This was solved using setjmp / longjmp. The second difficulty was adapting the barriers to the oncancel clause. A total rewrite of the runtime support function for barriers was required.

Great care was taken not to impact performance with our changes. Our choice of cooperative cancellation enabled us to implement thread cancellation without any measurable impact on performance. None of our test applications showed any notable slowdown. Neither did the OpenMP Microbenchmarks [9], which we used to measure performance of our adapted barrier implementation.

3.5 Application

In this section, we apply the thread cancellation functionality to our labyrinth search example from Sect. 2. We had isolated three main problems there:

- there is no easy way to branch out of a parallel region, the only possible workaround is to use flags

```
1  #pragma omp parallel
2  {
3    while (!omp_get_cancelled ()) {
4      while ((task = pop (taskpool)) != NULL) && !omp_get_cancelled ()) {
5        mark (labyrinth, task);
6        if (!inspect_field_for_exit (task))   // inspect all neighbours
7          push (neighbours (task), next_taskpool);  // no exit was found
8        else {
9          #pragma omp cancelregion  // an exit was found
10       }
11     }
12     #pragma omp barrier oncancel
13     {
14       #pragma omp exitregion
15     }
16     #pragma omp single
17     {
18       taskpool = next_taskpool; // switch the taskpools
19       next_taskpool = NULL;
20     } // implicit barrier
21     #pragma omp onbarriercancel
22     {
23       #pragma omp exitregion
24     }
25   }
26 } // end of parallel region with implicit barrier
```

Fig. 3. Parallel breadth first search in a labyrinth, using new language constructs for thread cancellation

- it is difficult to work with flags indicating that a region should end, at least as soon as barriers come into play
- if one forgets to flush a flag, a deadlock may arise

All these issues have been solved, as can be seen in Fig. 3. Firstly, it is easy now to branch out of a parallel region, as the cancelregion directive is a natural fit for the problem (see line 9). Just one directive, and the code will branch to the end of the parallel region on line 26. If barriers are involved like in our case, oncancel clauses have to be added (line 12), as well as an onbarriercancel clause at the end of the single worksharing construct (line 21). The second problem is also solved, as there is no need to work with programmer-managed flags to indicate that a parallel region should be finished. Last but not least, the third issue has been made obsolete: there is no need anymore to flush any cancel flags, as they are managed automatically by the OpenMP runtime system. We believe that this change alone will make errors less common in irregular parallel applications.

4 Concluding Remarks and Perspectives

In this paper, we have discussed a major problem with parallelizing irregular applications in OpenMP: lacking support for thread cancellation. A workaround and an extension to OpenMP have been suggested, whose main part is the cancelregion directive that enables cooperative cancellation.

A reference implementation of the extended OpenMP functionality can be found in a special release of the OMPi Compiler [6] that is available from the

authors on request. In the future, we plan to explore more applications with OpenMP, trying to find ways to improve the specification in the process. Our progress will be visible in the UKOMP project [10]. The project will serve as our testing ground for new functionality we discover to be useful, and also enables other developers to give feedback on how they like our changes. Additionally, the proposals are being sent to the OpenMP ARB, for consideration of inclusion into the official OpenMP specification.

5 Acknowledgments

We are grateful to Björn Knafla for proofreading the paper and for his insightful comments. We thank Vassilios V. Dimakopoulos and his group at the University of Ioannina for providing the OMPi compiler. 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 compiler and applications.

References

1. OpenMP Architecture Review Board: OpenMP specifications. `http://www.openmp.org/specs` (2005)
2. Mattson, T.G.: How good is OpenMP. Scientific Proramming **11** (2003) 81–93
3. Hisley, D., Agrawal, G., Satyanarayana, P., Pollock, L.: Porting and performance evaluation of irregular codes using OpenMP. Concurrency: Practice and Experience (2000)
4. Dedu, E., Vialle, S., Timsit, C.: Comparison of OpenMP and classical multithreading parallelization for regular and irregular algorithms. In Fouchal, H., Lee, R.Y., eds.: Software Engineering Applied to Networking Parallel/Distributed Computing (SNPD), Association for Computer and Information Science (2000) 53–60
5. Nikolopoulos, D.S., Polychronopoulos, C.D., Ayguade, E.: Scaling irregular parallel codes with minimal programming effort. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM), ACM Press (2001)
6. Dimakopoulos, V.V., Georgopoulos, A., Leontiadis, E., Tzoumas, G.: OMPi compiler homepage. `http://www.cs.uoi.gr/~ompi/` (2003)
7. Süß, M., Leopold, C.: Common mistakes in OpenMP and how to avoid them. In: Proceedings of the International Workshop on OpenMP - IWOMP'06. (2006)
8. Sun Microsystems: Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated. `http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html` (1999)
9. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. SIGARCH Comput. Archit. News **29**(5) (2001) 41–48
10. Süß, M.: University of Kassel OpenMP – UKOMP homepage. `http://www.plm.eecs.uni-kassel.de/plm/index.php?id=ukomp` (2005)