ParBlocks - A New Methodology for Specifying Concurrent Method Executions in Opus[⋆]

Erwin Laure

Institute for Software Technology and Parallel Systems
University of Vienna
erwin@par.univie.ac.at

Abstract. Many applications make use of hybrid programming models intermixing task and data parallelism in order to exploit modern architectures more efficiently. However, unbalanced computational load or idle times due to tasks that are blocked in I/O or waiting on results from other tasks can cause significant performance problems. Fortunately, such idle times can be overlapped with useful computation in many cases. In this paper we propose a simple, yet powerful methodology for specifying intra-object parallelism and synchronization in the context of the coordination language Opus.

1 Introduction

We recently introduced the coordination Language Opus [2] which allows a high level management of data parallel tasks. Its central concept is the *shared abstraction (SDA)*, which generalizes Fortran 90/HPF modules using an object-based approach and imposing monitor semantics. SDAs can be internally data parallel while task parallelism is exploited between different SDAs. SDAs communicate with one another via synchronous or asynchronous method invocation; arguments are passed with copy-in/copy-out semantics.

The monitor semantics of SDAs ensure a consistent state of the SDA data at the expense of potential parallelism losses. In fact, there may well be multiple method executions safely active within an SDA object. Weakening the monitor semantics of SDAs has the benefit of introducing an additional level of parallelism which can be exploited on systems with shared address space; but also on systems with distributed memory idle times, due to communication or synchronization with other tasks, can be overlapped with useful computation, thus reaching a better utilization of the available computation nodes.

Allowing concurrent executions of multiple methods within an SDA poses a number of difficulties (see [7] for a detailed discussion of intra-object concurrency) among which the most important one is how to specify potential parallelism and needed synchronization among methods. Compiler analysis can be used for detecting some potential for intra-SDA parallelism. However, a compiler

[⋆] The work described in this paper was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

is generally not able to detect all cases and therefore some support from the user is needed in order to exploit intra-SDA parallelism to some greater extent. This is also the case in other approaches described below:

Java [3] allows all methods of an object to execute in parallel unless explicitly synchronized. However, the `synchronized` attribute is ill suited for expressing partial concurrency and synchronization that is based upon the state of an object. Similarly, *OpenMP* [8] allows all methods to be invoked in parallel from work-sharing constructs, unless they are explicitly synchronized. On the contrary, *Fortran 95* [4] allows parallel invocation only for `pure` procedures that are free from certain side effects. *Path Expressions* [1] are an elegant means of specifying synchronization between processes by describing how a process is allowed to execute in relation to others, irrespective of their invocation order. With the help of Path Expressions complex synchronization patterns can be specified, however, it is not possible to specify synchronization which depends on the state of a process.

In this paper we propose a compiler directive called *ParBlock* which can be used for specifying potential parallelism and necessary synchronization among methods in a simple and intuitive way. Synchronization can be specified statically or dynamically.

2 The Opus Approach to Intra-SDA-Parallelism

Due to the specific properties of SDAs (SDAs are kind of "active" objects which are triggered by other objects) we identify a set of properties the specification mechanism for parallelism/synchronization has to fulfill:

Parallelism/synchronization information should be *encapsulated* within an SDA, since SDAs can be accessed by a set of tasks which are not necessarily aware of each other. Hence, it is necessary that a consistent internal state is guaranteed by an SDA itself rather than by synchronizing the accessing tasks. Synchronization should be possible in a *static* (i.e., independent of an SDA internal state) and and *dynamic* (i.e., state dependent) way. Although static synchronization can be seen as a special case of dynamic synchronization, having means for specifying synchronization statically allows a more efficient implementation. All parallelism/synchronization information should be specified on the highest possible level. Finally, the user should only be compelled to specify as much synchronization as necessary. Consequently, exclusive access to an SDA is still the default property of a method.

2.1 ParSets

Opus already provides some support for dynamic synchronization on the method level: the *condition clauses*. Condition clauses can be used to guard the execution of a method with a side-effect free logical condition. However, with this feature only synchronization that depends on the state of the SDA can be specified. It is not possible to synchronize two method executions independently of the internal data of the SDA. Hence, new means for specifying *static* parallelism and synchronization, in particular pairwise interference freedom among methods, are required. We propose that every method should be annotated with a set of method names representing all the methods with which its execution can safely overlap. This set is called *ParSet.* Note that ParSets are symmetric but not transitive. By default, the ParSet of a method is empty and thus the method has exclusive access to the SDA.

ParSets and condition clauses can be used in conjunction: while ParSets *statically* specify potential parallelism, condition clauses can be used to synchronize method executions in a *dynamic* way. The execution order of methods is derived implicitly from both, the parallelism specification and condition clauses, since before launching the execution of a method it is necessary to check if (1) the method is allowed to execute in parallel with all other methods currently being executed, and (2) its condition clause is satisfied. Obviously, both checks have to form an atomic action.

The direct specification of ParSets for every method is a cumbersome task and specifying ParSets in a consistent way is not trivial. Hence, we need higher level constructs for specifying static parallelism/synchronization.

In Section 1 we discussed *Path Expressions* which can be used to specify process parallelism at a high level. Such a technique could also be applied to Opus, however, Path Expressions explicitly specify the execution order of methods, irrespective of the invocation order. The direct specification of the execution order, however, is unwanted, since non-deterministic executions are deliberately enabled in Opus; condition clauses can be used for imposing specific execution orders, instead.

2.2 ParBlocks

Instead of specifying ParSets for every method we propose a new compiler directive called *ParBlock* for the static specification of parallelism/synchronization. ParSets can be derived from ParBlocks by the compiler. ParBlocks borrow from Path Expressions in that they allow the specification of parallel and mutual exclusive method executions, but without fixing the execution order of methods. The body of the ParBlock directive is a list of method names where all comma separated methods can execute in parallel while semi-colon separated methods need to execute mutually exclusive. We refer to a comma separated list as *par-section* and to a semi-colon separated one as *sync-section.* Both sections can be arbitrarily nested (using parenthesis) thus allowing complex synchronization patterns. In addition, multiple ParBlocks can be specified for an SDA. However, no method name may occur more than once in a given ParBlock nor in more than one ParBlock to avoid inconsistent declarations.

The syntax of the ParBlock directive is similar to HPF directives: the `PARBLOCK` keyword, which is preceded by the Opus compiler-directive-origin `!OC$` is followed by arbitrarily nested par- or sync-sections.

Although ParBlocks have enough expressiveness for inter-method parallelism, it is not possible to specify an overlapping of different instances of the same method. This can be accomplished by giving the method the F95 "pure" attribute. Moreover, pure methods can safely run mutually in parallel. Therefore, the compiler will generate an additional ParBlock containing all pure methods which is consistent with the F95/HPF standard.

Summarizing the above, Opus provides a set of features for specifying intra-SDA parallelism and synchronization, both dependent and independent of the SDA's internal state:

– **condition clauses:** for specifying dynamic synchronization based upon the internal state of the SDA,
– **ParBlocks:** for specifying parallelism as well as synchronization independently of the SDA's internal state, and
– **pure attributes:** for specifying potential parallelism according to the Fortran 95 standard.

The following example illustrates the use and expressiveness of ParBlocks:

Example 1. Consider an SDA with 6 methods, `a, b, c, d, e,` and `f`. All methods are allowed to execute in parallel but with the restrictions that (1) method `b` and `c` cannot execute concurrently and (2) method `d` cannot execute concurrently with neither `e` nor `f`. A sync-section is used for restriction (1): `(b;c)`. For restriction (2) we need to nest a sync-section with a par-section. Let's first specify that method `e` and `f` can run in parallel: `(e,f)`. Now we extend this expression specifying the synchronization of `d`: `(d;(e,f))`. We have now specified all the necessary synchronization and can put everything in a par-section. The resulting ParBlock for our example is:
`(a,(b;c),(d;(e,f)))`. In an Opus program the required directive would look like: `!OC$ PARBLOCK(a,(b;c),(d;(e,f)))`. ■

2.3 Implementation

The Opus compiler parses the ParBlock-directives of an SDA and constructs a ParSet for each method of an SDA (see [5] for a detailed description of the algorithms).

These ParSets are used at runtime to check if a method can start executing in parallel with other methods. In particular, a new method can start its execution if and only if the set of the currently executing methods is a subset of its ParSet and its condition clause is satisfied as well.

The Opus implementation design must also be modified to facilitate overlapping method execution. In particular, we described in [6] that an SDA is compiled to an *active object* consisting of two threads: a *server thread* responsible for retrieving incoming request and an *execution thread* which executes the methods. To allow concurrent method executions within an SDA, an SDA object requires a set of execution threads instead of only one.

3 Conclusions

In this paper we introduced a simple, yet expressive method for specifying intra-SDA parallelism. In [5] we present the applicability of our approach to a set of synchronization problems and discuss its benefits wrt. other approaches, like the Java multithreading model.

The proposed methodology is currently being implemented in our Opus compilation and runtime framework.

References

[1] R.H. Campbell. *Path Expressions: A technique for specifying process synchronization.* PhD thesis, The University of Newcastle Upon Tyne, 1976.

[2] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6/9:345–362, Winter 1997.

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesely, 1996.

[4] ISO. Fortran 95 Standard. ISO/IEC 1539 :1997.

[5] E. Laure. ParBlocks - A new Methodology for Specifying Concurrent Method Executions in Opus. Technical Report TR99-05, Institute for Software Technology and Parallel Systems, University of Vienna, 1999.

[6] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. *Concurreny: Practice and Experience*, to appear 1999.

[7] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[8] *OpenMP C and C++ Application Program Interface Version 1.0.* http://www.openmp.org/, October 1998.