ParBlocks - A new Methodology for Specifying Concurrent Method Executions in Opus*

Erwin Laure

Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
erwin@par.univie.ac.at

A short version of this report appears in: Proceedings Euro-Par'99, Springer, 1999

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 either 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. Our design combines both, static and dynamic synchronization on a high level. We motivate our design with some examples and discuss implementation strategies for compilation as well as runtime support.

1 Introduction

With the advent of teraflops supercomputers and the usage of the Internet as a huge "meta-computer", the complexity of simulations being tackled by scientists and engineers is increasing exponentially. In order to utilize the available architectures efficiently, several levels of parallelism need to be exploited by such simulations.

We recently introduced the coordination Language Opus [4, 5] 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.

With the monitor semantics of SDAs a consistent state of the SDA data is ensured 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 work described in this paper was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

the benefit of introducing an additional level of parallelism which can e.g. 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 e.g. [10] 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. In general, methods can safely execute concurrently if they do not interfere, i.e. if all write accesses affect disjoint data segments [1].

Compiler analysis can be used for detecting some potential for intra-SDA parallelism. However, a compiler 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.

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 (i.e., independent of an SDA's internal state) or dynamically (i.e., state dependent).

The remainder of this paper is organized as follows: We motivate the need for intra-SDA parallelism in Section 2 with some application examples. After a short review of some related approaches in Section 3 we present our approach in Section 4 and discuss various implementation issues in Section 5. In Section 6 we apply our approach to the example applications and conclude the paper with summarizing comments.

2 Application Examples

2.1 Intra-SDA Pipelines

One important class of applications that can take advantage of intra-SDA parallelism are pipelined problems which can for instance be found in FFT computations. A FFT code is typically structured into four tasks (*input, column-fft, row-fft, output*); each of those components could be scheduled on a separate stage of a pipeline. Although such pipelines can be expressed in Opus by assigning each stage to a separate SDA, this kind of structure imposes significant data transfer overhead among the SDAs. The overhead can be reduced by overlapping I/O with computation within an SDA in a pipelined manner. Figure 1 shows the SDA declarations for such a scenario (without the specification of the pipelined parallelism which is given in Section 6).

2.2 Partial Concurrency

Many problems can exploit "partial concurrency" between methods. With partial concurrency we refer to situations where some methods can execute concurrently but others need to have exclusive access to the object. One example of such problems are readers/writers-like accesses to data repositories. In a previous paper [4] we demonstrated how the coordination features of Opus can be used for synchronizing readers and writers to avoid deadlocks and starvation. However, apart from synchronizing these computational tasks, all accesses to the shared data-SDA have to be executed in a mutual exclusive way, currently. This is too restrictive since multiple read requests may obviously be active at the same time, but also multiple write

SDA TYPE fft_stage1
  TYPE (fft_data) :: data
 	
CONTAINS
  SUBROUTINE read_data()
    ...
  END SUBROUTINE
 	
  SUBROUTINE col_fft(result)
    TYPE (fft_data), INTENT(out) :: result
    ...
  END SUBROUTINE
END SDA TYPE
	SDA TYPE fft_stage2
	  TYPE (fft_data) :: data
	
	CONTAINS
	  SUBROUTINE row_fft(input)
	    TYPE (fft_data), INTENT(in) :: input
	    ...
	  END SUBROUTINE
	
	  SUBROUTINE output()
	    ...
	  END SUBROUTINE
	END SDA TYPE

Figure 1: 2stage FFT

requests may overlap, given that different parts of the data are accessed. The code excerpt in Figure 2 shows a data repository for this kind of scenario.

3 Related Approaches

Before we discuss our proposal for specifying intra-SDA parallelism in Opus we briefly review some existing approaches for specifying concurrent method executions below. Focus is hereby laid on high level synchronization mechanisms; techniques such as the use of mutexes and condition variables or the direct specification of data accesses (that can e.g. be found in Jade [12]) are beyond our scope.

Java [6] The Java multithreading model allows all methods of an object to execute in parallel unless they are explicitly synchronized. This means that the user has to ensure a correct program behavior by either synchronizing methods or code regions. Although the `synchronized` attribute for methods is an elegant feature there are some non-trivial problems involved: synchronizing a method only means that it is executed mutually exclusive with other synchronized methods - unsynchronized methods may well interfere with synchronized ones. As a consequence of this, the `synchronized` attribute is ill suited for expressing partial concurrency. Moreover, synchronization which is based upon the state of an object's internal data needs more low level constructs such as `wait` and `notify` (cf. Section 6).

OpenMP [11] OpenMP, designed for exploiting shared memory parallelism, allows parallel executions of methods via *work-sharing* constructs like the `for` directive or the `sections` directive. Similar to Java, methods are called from within such constructs in parallel without synchronization. The user is required to either avoid invoking interfering methods from within work-sharing constructs or to synchronize method executions properly by the use of *synchronization* constructs like the `critical` or `atomic` construct.

```
SDA TYPE repository
  TYPE (user_type) :: data(...)
CONTAINS
  SUBROUTINE read1(...)
    ...
  END SUBROUTINE

  SUBROUTINE read2(...)
    ...
  END SUBROUTINE

 ! write1 and write2 affect disjoint data
  SUBROUTINE write1(...)
    ...
  END SUBROUTINE

  SUBROUTINE write2(...)
    ...
  END SUBROUTINE
END SDA TYPE repository
```

Figure 2: Data Repository

Fortran 95/HPF [8, 7] Parallelism in Fortran is exploited primarily in the form of data parallelism. Nevertheless, procedures can be called from within explicitly parallel constructs (e.g. the `forall` construct) but only if they have the `pure` attribute. This attribute assures that a procedure is free of certain side effects that prevent parallel execution. Fortran allows the specification of concurrent procedure executions only in a quite restrictive way which does not cover all interference-free cases. Moreover, only executions of the same procedure can be overlapped; specifying concurrent executions of different procedures is not possible.

HPF adds a set of explicitly parallel constructs to Fortran. Parallel executions of procedures are in particular enabled by the `INDEPENDENT` directive and the `TASK_REGION` construct. While to `INDEPENDENT` loops similar restrictions as to the F95 `forall` construct apply, the `TASK_REGION` construct is restricted in that a procedure called from within a `TASK_REGION` is only allowed to access data which is mapped to the executing set of processors.

Path Expressions [2, 3] Path Expressions 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. In particular, Path Expressions allow the specification of *sequences, selections, repetitions* and *simultaneous execution* among a set of processes. Path Expressions not only specify parallelism or synchronization among processes, but also their execution 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.

4 The Opus Approach

4.1 Introduction

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:

- *Encapsulation*: All parallelism/synchronization information should be encapsulated within an SDA. SDAs can be accessed by a set of tasks which do not necessarily have to be 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.
- *Static and Dynamic Synchronization*: 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 more efficient compilation. Moreover, specifying synchronization statically is often a more natural approach, as can e.g. be seen in Section 6.
- *High Level*: All parallelism/synchronization information should be specified on the highest possible level. We believe that parallelism/synchronization should be specified on the method level since methods are the main locus for parallelism in Opus.
- *User Friendly*: User friendliness is required in two ways: apart from having an intuitive means of specifying parallelism/synchronization, 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.

4.2 Existing Features

Opus 1.0 [4] 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.

In addition, the HPF binding of Opus provides means for expressing parallelism: the Fortran *pure* attribute with all the properties described in Section 3.

4.3 ParSets

Apart from these features, 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 if
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 3 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.

4.4 ParBlocks

Instead of specifying ParSets for every method we propose a new compiler directive called *ParBlock* for the static specification of parallelism/synchronization. ParSets are derived from ParBlocks by the compiler as described in Section 5. 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. Therefore, Path Expression features such as *sequences* or *repetition* are not available in ParBlocks.

ParBlocks are compiler directives specifying either potential parallelism or the need for synchronization between 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. These restrictions prohibit inconsistencies in the declaration of ParBlocks.

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. Thus, the concurrent execution of different instances of the same method is only allowed for pure methods. 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.

Syntax: ParBlock Directive

parblock-directive	**is**	*oc-directive-origin parblock-stmt parblock-body*
oc-directive-origin	**is**	`!OC$`
parblock-stmt	**is**	`PARBLOCK`
parblock-body	**is**	(*par-section*)
	or	(*sync-section*)
par-section	**is**	*parblock-element* [, *parblock-element*] ...
sync-section	**is**	*parblock-element* ; *parblock-element* [; *parblock-element*] ...
parblock-element	**is**	*method-name*
	or	*parblock-body*

Constraint: No *method-name* may occure more than cone in a given `PARBLOCK`.

Constraint: No *method-name* may occur in more than one `PARBLOCK`.

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.

Before discussing the associated compilation techniques and runtime support in more detail, let us illustrate the use and expressiveness of ParBlocks in the following examples:

Example 4.1:

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`.

As already mentioned, ParBlocks provide sync-sections for specifying mutual exclusion. This concept can be applied to restriction (1) resulting in the following expression: `(b;c)`.

Restriction (2) is slightly more difficult, because we need to synchronize a method with two other methods which in turn may execute concurrently. Thus, we need

Method	ParSet
a	b, c, d, e, f
b	a, d, e, f
c	a, d, e, f
d	a, b, c
e	a, b, c, f
f	a, b, c, e

Figure 3: ParSets for Example 4.1

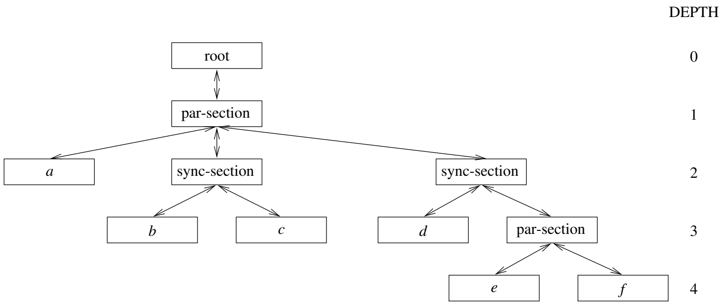

Figure 4: AST for Example 4.1

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)))`. ■

Example 4.2:

Consider another SDA with 4 methods, `a, b, c,` and `d`. We want to specify partial parallelism such that `a` is allowed to execute in parallel with `b` and `c`, and `c` is allowed to execute in parallel with `d`. All other combinations cannot be executed in parallel.

The ParBlock expressing this partial parallelism is `(c,(d;(a,b)))`. ■

ParBlocks can be analyzed by the compiler which annotates every method of an SDA with a ParSet. Figure 3 shows all ParSets for Example 4.1. The algorithm for generating these sets is introduced in the following section.

5 Implementation

5.1 Compilation

The Opus compiler parses the ParBlock-directives of an SDA and constructs an AST for every directive. The nodes of an AST are either *par-sections*, *sync-sections*, or *method names* where the method name nodes are the leaf nodes.

Example 5.1:

Consider the ParBlock from Example 4.1:

`(a,(b;c),(d;(e,f)))`

Figure 4 shows the AST which is generated out of this ParBlock. ■

Based upon the AST representation a ParSet is generated for each method of an SDA using Algorithm 5.1:

First, the AST is traversed in order to find the node representing method x. If such a node is found, all method names belonging to a par-section which lies on the path from the root to the x-node are added to the ParSet. This is accomplished with help of the routine "AddElements" which adds all method names belonging to the subtree rooted by "Child" to a ParSet "Set".

Algorithm 5.1:

Notation: Let $\mathcal{M}$ denote the Set of all methods of an SDA. The ParSet of a method x is denoted by $\mathcal{P}^x$. $\oslash$ is the empty set.

```
do ∀ x ∈ M {
  P^x = ⊘;
  do ∀ root ∈ ParBlocks {
    CurrentNode = TraverseTreeFind(root, x);
    if (CurrentNode == NULL)
      continue;
    while (CurrentNode→Father != root) {
      if (CurrentNode→Father == par-section)
        AddOtherElements( P^x, CurrentNode);
      CurrentNode = CurrentNode→Father;
    }
  }
}

AddOtherElements(Set, Node) {
  do ∀ Child(Node→Father) {
    if (Child != Node)
      AddElements(Set,Child);
  }
}
```

■

Example 5.2:

Consider the construction of $\mathcal{P}^e$ from the AST of Example 5.1:
Initially, $\mathcal{P}^e = \oslash$. The matching node for e is found at depth 4 and its father is a par-section. Hence, all elements belonging to this par-section are added to $\mathcal{P}^e$ which now is the set $\{f\}$. We follow the path from e to the root-node passing a sync-section at depth 2 whose father at depth 1 is a par-section again. Consequently, all elements belonging to this par-section are added to $\mathcal{P}^e$ resulting in the set $\{f, a, b, c\}$. Since the father of this par-section is the root-node, our algorithm terminates for this set.

■

5.2 Runtime Support

In [9] we discuss in detail the compilation and runtime support for Opus. The main concept is that an SDA is compiled into an *active object* consisting of two threads: a *server thread* responsible for retrieving incoming request and storing them in a shared memory area (consisting of Method Invocation - *MI* - queues) in form of *execution records*; and an *execution thread* which retrieves records from the MI-queues and executes the associated methods. To allow concurrent method executions within an SDA, an SDA object requires a set of execution threads instead of only one (cf. Figure 5).

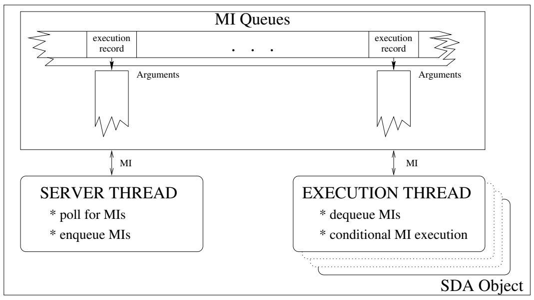

Figure 5: Structure of an SDA Object

The total number of execution threads is system dependent: while on an SMP the number of execution threads should at least be equal to the number of available processors, on a single processor machine only a few of them are needed. The actual number can be determined dynamically at runtime via environment variables.

The execution threads are in duty of validating, whether a new method can start executing in parallel with others. For this purpose, it is kept track of all methods currently being executed. A new method can start its execution if and only if the set of the currently executing methods is a subset of its ParSet. In addition, its condition clause must be satisfied as well. Obviously, both checks need to be performed in an atomic action. Algorithm 5.2 illustrates the method invocation mechanism of an execution thread.

Algorithm 5.2:

Notation: Let Π denote the set of all methods currently being executed.

...
/* begin atomic */
if ($\Pi \subseteq \mathcal{P}^x$) {
   if (/* condition-clause(x) */) {
      $\Pi = \Pi \cup \{x\}$
/* end atomic */
/* launch execution of x */
      $\Pi = \Pi \setminus \{x\}$
   }
}
...

■

```
SDA TYPE fft_stage1
  TYPE (fft_data) :: data(0:1)
   ! flags used in condition clauses
  LOGICAL :: read_allowed = .TRUE.
  LOGICAL :: col_allowed = .FALSE.
   ! data set index
  INTEGER :: active_read = 0
  INTEGER :: active_col = 0
CONTAINS
  SUBROUTINE read_data() &
   & WHEN (read_allowed)
    read_allowed = .FALSE.
     ! read fft_data using data(active_read)
    active_read = MOD(active_read+1,2)
    col_allowed = .TRUE.
  END SUBROUTINE

  SUBROUTINE col_fft(result) &
   & WHEN (col_allowed)
    TYPE (fft_data), INTENT(out) :: result
    col_allowed = .FALSE.
    read_allowed = .TRUE.
     ! call column_fft(data(active_col))
    active_col = MOD(active_col+1,2)
  END SUBROUTINE

 !OC$ PARBLOCK(read_data, col_fft)

END SDA TYPE
```

```
SDA TYPE fft_stage2
  TYPE (fft_data) :: data(0:1)
   ! flags used in condition clauses
  LOGICAL :: row_allowed = .TRUE.
  LOGICAL :: output_allowed = .FALSE.
   ! data set index
  INTEGER :: active_row = 0
  INTEGER :: active_out = 0
CONTAINS
  SUBROUTINE row_fft(input) &
   & WHEN (row_allowed)
    TYPE (fft_data), INTENT(in) :: input
    row_allowed = .FALSE.
     ! call row_fft(data(active_row)=input)
    active_row = MOD(active_row+1,2)
    output_allowed = .TRUE.
  END SUBROUTINE

  SUBROUTINE output()&
   &  WHEN (output_allowed)
    output_allowed = .FALSE.
    row_allowed = .TRUE.
     ! call output(data(active_out))
    active_out = MOD(active_out+1,2)
  END SUBROUTINE

 !OC$ PARBLOCK(row_fft, output)

END SDA TYPE
```

Figure 6: 2stage FFT with ParBlocks

6 Applied Application Examples

In this Section we show the applicability of our approach for the problems introduced in Section 2.

6.1 Intra-SDA pipelines

Our first example shows the pipelined execution of SDA methods which can e.g. be applied to FFT computations. The original code from Figure 1 has to be modified in three aspects:

1. An appropriate *ParBlock* is required.

2. A correct pipelined behavior needs to be guaranteed by *condition clauses*.

3. For each stage of the pipeline a separate copy of the FFT data has to be allocated to guarantee interference freedom.

Figure 6 shows the necessary code excerpts. Note the combination of a static parallelism specification and the dynamic synchronization via condition clauses.

6.2 Partial Concurrency

The data repository examples of Section 2.2 is much simpler, because it is sufficient to specify parallelism among readers as well as among writers, but to guarantee that no overlapping of readers and writers may occur. Thus, all parallelism and synchronization information can be specified statically via ParBlocks. Figure 7 (a) shows the modified code. The expressiveness and intuitiveness of ParBlocks is also demonstrated in this example by a comparison with an equivalent Java code which is given in Figure 7 (b).

```
SDA TYPE repository
  TYPE (user_type) :: data(...)

CONTAINS
  SUBROUTINE read1(...)
    ...
  END SUBROUTINE

  SUBROUTINE read2(...)
    ...
  END SUBROUTINE

  SUBROUTINE write1(...)
    ...
  END SUBROUTINE

  SUBROUTINE write2(...)
    ...
  END SUBROUTINE

!OC$ PARBLOCK &
& ((read1,read2);(write1,write2))

END SDA TYPE repository
```

(a)

```
public class repository {
  // data
  private int n_read=0, n_write=0;
  private Object mutex1, mutex2;

  public data read1(...) {
    beforeRead(); ...; afterRead();}

  public data read2(...) {
    beforeRead(); ...; afterRead(); }

  public void write1(data) {
    synchronized(mutex1) {
      beforeWrite(); ...; afterWrite(); }}

  public void write2(data) {
    synchronized(mutex2) {
      beforeWrite(); ...; afterWrite(); }}

  private synchronized void beforeRead() {
    while (n_write != 0) { wait();}
    n_read++; }

  private synchronized void afterRead() {
    if (--n_read == 0) notifyAll(); }

  private synchronized void beforeWrite() {
    while (n_read != 0) { wait();}
    n_write++; }

  private synchronized void afterWrite() {
    if (--n_write == 0) notifyAll(); }
}
```

(b)

Figure 7: Data Repository with ParBlocks (a); and in Java (b)

One can see that due to the lack of specification possibilities for partial concurrency additional counters together with synchronized functions modifying these counters are needed. Moreover, low level *wait* and *notifyAll* functions, as well as mutex synchronization of the *write* methods are necessary to guarantee correct behavior.

7 Conclusions

In this paper we introduced a simple, yet expressive method for specifying intra-SDA parallelism and showed how this method cooperates with a set of powerful synchronization mechanisms. Moreover, strategies for implementing both the compilation and runtime support were discussed in detail. Examples of important application classes proved the applicability of our approach and its benefits wrt. other approaches, like the Java multithreading model.

However, there are some subtleties that have to be considered when using ParBlocks. Most important is that in some special cases a synchronized method may be life-locked. Consider e.g. the following ParBlock: `(a;(b,c))`. If methods `b` or `c` are invoked frequently, there might be little chances for `a` to ever start executing since it must wait until no other method is being executed. However, this situation can be overcome by specifying proper condition clauses for `b` and `c` which will prevent them from executing at times, thus giving `a` a chance for being executed.

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

References

[1] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

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

[3] R.H. Campbell and A.N. Habermann. The Specification of Process Synchronization by Path Expressions. In G. Goos and J. Hartmanis, editors, *LNCS*, volume 16. Springer Verlag, 1974.

[4] B. Chapman, M. Haines, E. Laure, P. Mehrotra, J. Van Rosendale, and H. Zima. Opus 1.0 Reference Manual. Technical Report TR 97-13, Institute for Software Technology and Parallel Systems, University of Vienna, October 1997.

[5] 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.

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

[7] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 2.0*, January 1997.

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

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

[10] B. Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, 1997.

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

[12] M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, June 1993.