

## 2 Source Transformation of Pure Parallel Regions

In this chapter, we will first describe the notation that we use for describing a parallel code region and its execution. This work is about the source transformation of pragma-based parallel regions. Therefore, we will define the *SPL* language that is used as input and as output language for our source transformation. After we have defined the source transformation rules, we will show that the resulting code represents the AD model as defined in the introduction chapter. The *SPL* language is shaped such that it comprises one compiler directive in form of a pragma that defines an associated code region as concurrently executable. We will denote this kind of parallel region as a *pure* parallel region.

A pure parallel region is a sequence of statements surrounded by curly brackets and a preceding parallel pragma such as, for example,

```
#pragma omp parallel
{
     $s_1$  ;  $s_2$  ;  $\dots$  ;  $s_q$  ;
}
```

Listing 2.1: A pure parallel region.

where the appearance of the code inside of the parallel region is irrelevant at this point. The term 'pure' means in this context that there are no additional pragmas inside the parallel region. However, we try to stay on a level that allows to apply the results of this chapter to a non-OpenMP parallel region.

### 2.1 Formalism and Notation

In general, we assume a parallel region  $P$  to have a structure as shown in Listing 2.1 where a sequence of  $q$  statements are surrounded by a pragma that declares these statements as parallelizable. A sequence of statements is indicated by  $S$ . There are potentially subsequent statements below  $S$ , which occur, for example, when there is a conditional statement or a loop statement. We display these subsequent statements with  $S'$ . The sequence that is on the first level does not have any

parent sequences and is referred by  $S_0$ . By writing  $P = S_0$  we indicate that the parallel region  $P$  has the sequence  $S_0$  on its first level. When we want to express a special structure of the parallel region, we use sequences of statements for better readability. For example, consider the structure of  $P$  in

```
#pragma omp parallel
{
    S1
    s      ;
    S2
}
```

where the parallel region consists of two sequences of statements  $S_1$ ,  $S_2$  and a statement  $s$  that separates these sequences from one another.

To show the correctness of the source transformations in this work, we need a mathematical abstraction for the concurrent execution of the parallel regions. This abstraction is based on the usage of interleaved statements and was proposed by Ben-Ari in [5, Chapter 2], whereby this approach goes back to [52, Chapter 2]. No matter how many physical processors are available, we only consider one processor and  $p$  different threads. Each thread holds a sequence of  $q$  *atomic* statements. An atomic statement means that this statement cannot be divided into smaller instructions.

The parallel execution consist of  $pq$  statements ( $p$  threads times  $q$  statements). The execution of these  $pq$  statements takes place on one processor. Thus, the statements have to be merged or interleaved by a scheduler into one sequence. This sequence is called *interleaving* and it represents the order in which the processor encounters the individual statements. An important assumption is that the strategy of the scheduler is unknown and every permutation of the  $pq$  statements is possible. The interleaving is empty at the beginning of the parallel execution and all threads have a queue with  $q$  statements waiting for the scheduler to pick them up. Let us consider a possible status of an execution with three threads after the scheduler has already taken six statements. One possible appearance of the interleaving is shown in Figure 2.1.

We denote a statement with  $s$  or with  $s_i$  when the statement is the  $i$ -th statement in a sequence of statements  $(s_1; s_2; \dots; s_q)$ . Each thread has an unique identifier number starting with zero and ending with  $p - 1$  because  $p$  threads are participating the execution. We differ between threads such that we note the different identifier numbers. For example, thread 0 is the thread that is identified with the number zero. Thread  $t$  is a thread where the identifier number is arbitrarily chosen, which means that  $t$  can be a number from zero to  $p - 1$ . The fact that statement

$s$  is executed by thread  $t$  is denoted by  $s^t$ . Consider, for example, the case that a possible interleaving contains

$$s_3^1 s_5^0.$$

This means that the scheduler first took the third statement of the thread 1 and subsequently it chose, for whatever reason, the fifth statement of thread 0.

An atomic statement is one that is executed by a thread  $t$  without any context change to a different thread  $t'$ . In the literature there is often the explanation of an atomic statement as a statement that cannot be divided into smaller instructions. However, given an atomic statement  $s$ , it can in general be divided into several assembly instructions. In [52, Section 2.2] it is shown that a concurrent sequence of statements where each statement has at most one *critical reference* can be assumed to be atomic. And most important, it is shown that the behavior of the result is the same as if the sequence is compiled to a machine architecture with an atomic load and store.

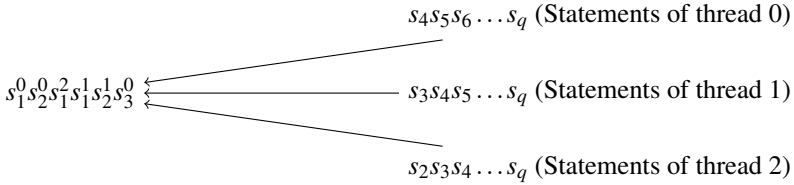


Figure 2.1: This figure shows one possible status of the program execution after processing six statements with three threads. Each thread has  $q$  statements ( $s_1; \dots; s_q$ ) at the beginning of the execution. Then the scheduler chooses some statements from the three threads and puts them to the interleaving. After the scheduler has taken six statements, the status of the execution can appear as in this figure. At this point in the execution, the first three statements of thread 0 have already been executed. The next time the scheduler decides to choose thread 0 to be the execution thread, statement  $s_4^0$  is put into the interleaving. Thread 1 is waiting for statement  $s_3^1$  being put into the interleaving. For thread 2, this holds analogously for statement  $s_2^2$ . Because all statements have a fixed order inside their respective executing thread, statement  $s_{i+1}^t$  cannot be executed before statement  $s_i^t$ . In the figure we see that statement  $s_2^0$  occurs before statement  $s_3^0$ .

In OpenMP one can define certain assignments to be atomic by using the directive `atomic`. We do not mean this atomicity when we write about an atomic statement. Nevertheless, the atomic construct can be associated with a statement  $s$  such that  $s$  is executed atomically as we will see in Chapter 4. This is very helpful in case that statement  $s$  contains a critical reference. We cite the definition of a critical reference that Ben Ari presents in his textbook.

**Definition 34** ([5], p.27). An occurrence of a variable  $v$  is defined to be a *critical reference*,

1. if it is assigned to in one thread and has an occurrence in another thread, or
2. if it has an occurrence in an expression in one thread and is assigned to in another.

A program satisfies the *limited-critical-reference* (LCR) restriction if each statement contains at most one critical reference.  $\square$

The author Ben-Ari presents an example as justification for the abstraction with interleavings of atomic statements, see [5, p. 16]. The example describes the following situation where two threads want to write to the memory location on a multiprocessor machine. One thread wants to write the value 1 and the other thread wants to write the value 2. Thus, both threads write different bit values. The question is what the result of the store operation is. The value might be undefined when both bit values are combined per logical OR operation. In this case we would have the value 3 inside the memory location. This would reveal the abstraction as unqualified, but in practice this case does not occur because the memory hardware writes values not bitwise but cell wise. A cell is, for example, 32 bit on a 80386 processor or 64 bit on modern processors. The bits that the threads want to write are contained in the same cell. Therefore, the memory hardware completes one access before the other takes effect. In this case the memory hardware performs the atomicity and interleaving.

For our approach it is important to understand that the atomic statements of each individual thread are interleaved arbitrarily. For example, let us consider a parallel region  $P$  that consists of the sequence of statements  $S = (s_1; \dots; s_q)$ . The atomic statements considered during the execution are in case of thread 0  $(s_1^0; \dots; s_q^0)$ , for thread 1 they are  $(s_1^1; \dots; s_q^1)$ , and so on. The permutations where the order does not reflect the order of the statements inside of a particular thread are not valid. For example, an interleaving beginning with  $s_1^1 s_2^0 s_2^1 s_1^0$  is not possible, since the second statement of thread 0 is executed before the first statement  $s_1^0$  of thread 0 appears in the interleaving.

The set of variables used in a parallel region  $P$  is denoted by  $VAR_P$ , the set of variables occurring in statement  $s$  is denoted by  $VAR_s$ . Most of the time we will discuss about comparing references instead of comparing variables. Therefore, we

need a mapping from the set of variables to the memory address space. This is done by the operator  $\&$ :

$$\& : VAR_P \rightarrow MemLoc \subset \mathbb{N}.$$

The following example uses a statement where multiple threads increment a shared variable to clarify the term of a critical reference and the LCR property.

**Example 13.** Definition 34 can be explained by a source statement  $s$  represented by

$$s = \quad n \leftarrow n + 1 \quad ,$$

where  $n$  is a shared variable. This example is similar to the example in [5, p. 27]. Depending on the set of statements contained in the assembly language of a given architecture, this statement can be compiled into an atomic increment instruction or into a sequence of assembly statements whose semantic are the same as the source statement. What one can expect when considering Figure 2.1 is that the number of interleavings is different when we consider, on the one hand, a program consisting of one atomic increment instruction, or on the other hand, a program that consists of a sequence of assembly statements. The behavior can be different opposed to the case that the sequence of assembly statements would be evaluated atomically. Nevertheless, the abstraction of atomic source statements can be used when each statement fulfills the LCR property. To clarify this, we split the statement  $s$  into<sup>1</sup>

$$\begin{aligned} s_1 &= \quad a \leftarrow n \\ s_2 &= \quad n \leftarrow a + 1 \quad , \end{aligned}$$

where  $a$  is a private (thread local) variable.

We assume that the statement  $s$  and the statements  $s_1; s_2$  are executed by thread  $t$  and by thread  $t'$ . According to statement  $s$ , thread  $t$  executes at runtime the SPMD instance

$$s^t = \quad n^t \leftarrow n^t + 1$$

and thread  $t'$  executes

$$s^{t'} = \quad n^{t'} \leftarrow n^{t'} + 1$$

---

<sup>1</sup>Please note that there is an error in [5, p. 27] that we reported the author. It should be similar as presented here as pointed out on the errata website (<http://www.weizmann.ac.il/sci-tea/benari/books/pcdp2-errata.html>).

where we use a superscript index to indicate the thread that accesses the variable  $n$ . The memory locations of  $n$  concerning thread  $t$  is  $\&n^t$ , thread  $t'$  uses  $\&n^{t'}$ . Since  $n$  is a shared variable, the reference of both threads is the same, namely  $\&n^t = \&n^{t'}$ .

When we consider the reference  $\&n$  occurring in statement  $s$  on the left-hand side, we conclude that this reference is critical. It is critical by Definition 34 (1) because it occurs in thread  $t$  as left-hand side reference and in thread  $t'$  as right-hand side reference. Let us now consider the reference  $\&n$  on the right-hand side of statement  $s$ . It is critical by Definition 34 (2) since it is read by thread  $t$  and is assigned to in thread  $t'$ . Therefore,  $s$  has two critical references and does not fulfill the LCR property.

Now, we consider the code where we split  $s$  into  $s_1$  and  $s_2$ . This code is semantically equivalent to  $s$  but differs according to the number of critical references.

$$\begin{array}{ll} s_1^t = & a^t \leftarrow n^t \\ s_2^t = & n^t \leftarrow a^t + 1 \end{array} \qquad \begin{array}{ll} s_1^{t'} = & a^{t'} \leftarrow n^{t'} \\ s_2^{t'} = & n^{t'} \leftarrow a^{t'} + 1 \end{array}$$

Variable  $a$  is a thread local variable is therefore noncritical. Statement  $s_1^t$  reads the shared reference  $\&n^t$  and assigns it to the thread local reference  $\&a^t$ . But because of  $\&n^t = \&n^{t'}$ , this reference is also assigned to in thread  $t'$  as  $\&n^{t'}$  in  $s_2^{t'}$ . Therefore, reference  $\&n^t$  is a critical reference in statement  $s_1$  due to Definition 34 (2). The reference  $\&n^t$  is assigned to in  $s_2^t$  and is read in  $s_1^{t'}$ . Thus, the occurrence of variable  $n$  in  $s_2$  is also critical because of (1) in Definition 34. In summary,  $s_1$  and  $s_2$  both fulfill the LCR property whereby  $s$  does not. This means that the approach of interleavings of atomic statements can be applied to the code  $s_1; s_2$  but not to  $s$ .

Just to be clear, the LCR property does not ensure the correct evaluation of the sequence, it only ensures that this abstraction shows the same behavior as an architecture with an atomic load and store. If we abstract  $s_1; s_2$  by interleavings, then we get different results for  $n$  due to the fact that  $n$  is critical. The following table shows that only two of the possible six interleavings result in the correct result of two increment operations which results in being  $n + 2$ . The remaining four interleavings all end up with setting  $n$  to a value of  $n + 1$ . The column titled with  $a^0$  shows the value of variable  $a$  that belongs to thread 0, whereby  $a^1$  belongs to thread 1.

Interleaving	$a^0$	$a^1$	$n$ (after execution)	Correct
$s_0^0; s_1^0; s_0^1; s_1^1;$	$n$	$n + 1$	$n + 2$	true
$s_0^0; s_0^1; s_1^0; s_1^1;$	$n$	$n$	$n + 1$	false
$s_0^1; s_0^0; s_1^1; s_1^0;$	$n$	$n$	$n + 1$	false
$s_0^0; s_0^1; s_1^1; s_1^0;$	$n$	$n$	$n + 1$	false
$s_0^1; s_0^0; s_1^1; s_1^0;$	$n$	$n$	$n + 1$	false
$s_0^1; s_1^1; s_0^0; s_1^0;$	$n + 1$	$n$	$n + 2$	true

In contrast to the above, let us reconsider the statement  $s$  and the possible interleavings

$$s^0; s^1$$

and

$$s^1; s^0.$$

Both interleavings lead to  $n$  having the value  $n + 2$ , which is the correct result but shows that in this case our mathematical abstraction would be wrong, since the associated interleaved assembly instructions could lead to the result  $n + 1$ .

The above table shows why the runtime situation of a code where a critical reference is involved is called a race condition in the domain of shared memory parallel programming. The finishing value in the critical reference depends on the thread that performs the last store to the reference.  $\triangle$

If we consider synchronization methods in parallel programming, then certain interleavings are not possible. Therefore, we define an order of interleavings. Let us suppose that two statements  $s_i$  and  $s_j$  are executed by the threads  $t$  and  $t'$ . The order of  $s_i^t$  and  $s_j^{t'}$  inside of  $I$  is denoted by

$$s_i^t \prec s_j^{t'}$$

when  $s_i^t$  is evaluated prior to  $s_j^{t'}$ , or in other words, in the interleaved sequence  $s_i^t$  is before  $s_j^{t'}$ . Please note that, when considering statements executed by two different threads, this order is independent of the order of statements that each thread executes. For example let  $s_1$  and  $s_8$  be two statements inside of a sequence of statements  $S = (s_1; s_2; \dots; s_8)$ . Suppose that  $S$  is executed by two threads identified by 0 and 1. If we consider only thread 0, then it can only hold

$$s_1^0 \prec s_8^0$$

because  $s_8^0 \prec s_1^0$  is a contradiction to the order of  $(s_1; s_2; \dots; s_8)$ . In case that we consider different threads, for example 0 and 1, then both

$$s_1^0 \prec s_8^1$$

and

$$s_8^0 \prec s_1^1$$

are possible because thread 0 could execute  $s_8$  prior as thread 1 executes  $s_1$ .

The following definition describes the set of possible interleavings for a given parallel region  $P$  with  $q$  statements that is executed with  $p$  threads. This set contains all the possible interleavings and it may possibly have infinite size, for example, when at least one thread executes an infinite loop.

**Definition 35** (set of possible interleavings). Suppose a parallel region  $P$  with  $q$  statements, this is  $P = S$  with  $S = (s_1; \dots; s_q)$ ,  $q \in \mathbb{N}$ , and  $P$  is executed by  $p$  threads. The set of possible interleavings is denoted by

$$\mathcal{I}(P, q, p)$$

□

**Lemma 36.** *Suppose a parallel region  $P$  is executed by  $p$  threads and consists of the sequence  $S$  with  $q$  straight-line code statements. Then the set of possible interleavings is finite and is defined by*

$$\mathcal{I}(P, q, p) = \left\{ I \mid I = (s_i^t)_{i=1, \dots, q}^{t \in \{0, \dots, p-1\}}, \text{ with } s_i^t \prec s_j^t \right\}$$

where  $s \in S$ ,  $i, j \in \{1, \dots, q\}$ ,  $t \in \{0, \dots, p-1\}$ .  $I$  is a sequence of instances of statements in execution. This sequence has a length of  $p \cdot q$  instances. The restriction  $s_i^t \prec s_j^t$  means that the ordering of the statements from  $S$  concerning the same thread must be preserved. The size of  $\mathcal{I}(P, q, p)$  is

$$|\mathcal{I}(P, q, p)| := \frac{(q \cdot p)!}{(q!)^p}$$



*Proof.* We show this by induction over  $q$ , the number of straight-line code statements. Assume we only have one statement in the parallel region. The parallel region is executed by  $p$  threads resulting in  $p$  statements inside of a possible interleaving. The  $p$  statements can be scheduled in any order. Therefore, we have  $p!$  possible interleavings.

Assuming that the number of possible interleavings for  $(q - 1)$  statements in a parallel region executed by  $p$  threads is

$$\frac{((q - 1) \cdot p)!}{((q - 1)!)^p}, \quad (2.1)$$

then we have to show that the set of possible interleavings is

$$\frac{(q \cdot p)!}{(q!)^p}$$

when the number of statements in the parallel region is  $q$ . A parallel region with  $q$  statements executed by  $p$  threads results in  $q \cdot p$  statements. We can illustrate this by an urn problem with  $qp$  balls where  $q$  balls have one of  $p$  colors. Therefore, we have  $qp$  possibilities to choose the first statement in the interleaving and  $qp - 1$  possibilities to choose the second statement, and so forth. For the  $p$ -th statement we have  $qp - p + 1$  possibilities. We concern about the statements coming from a group of  $p$  threads or as shown with the urn problem; the fact that we have  $p$  different colors must be taken into account. Hence, the number of possibilities to choose the first  $p$  statements is

$$\frac{qp \cdot (qp - 1) \cdot \dots \cdot (qp - p + 1)}{\underbrace{q \cdot q \cdot \dots \cdot q}_{p\text{-times}}}.$$

According to the prerequisites, the number of possibilities to choose the remaining  $(q - 1)p$  statements is (2.1). Therefore, the number of interleavings is

$$\frac{qp \cdot (qp - 1) \cdot \dots \cdot (qp - p + 1)}{\underbrace{q \cdot q \cdot \dots \cdot q}_{p\text{-times}}} \cdot \frac{((q - 1) \cdot p)!}{((q - 1)!)^p} = \frac{(q \cdot p)!}{(q!)^p}$$

□

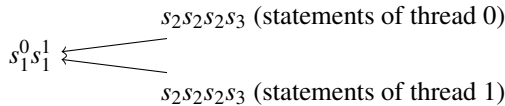
To get familiar with the above formalism, we present in the following several examples where we apply our notation to some OpenMP code examples. These examples contain different OpenMP constructs as the barrier or the master construct.

However, these examples serve only as proof of concept and from the next section on, we consider parallel regions such that these regions only consist of code and not further pragmas. The barrier and master constructs are topic of Chapter 4.

The OpenMP standard provides a loop construct for sharing loop iterations among a group of threads, see Section 1.3. To see the difference between a regular loop statement and a worksharing loop, let us consider the following example where a loop consist of three iterations.

**Example 14** (Loop Worksharing Construct of OpenMP). For simplicity we define a group of threads of size two and we assume that all the statements  $s_1, s_2, s_3$  in the parallel region are straight-line code statements. In OpenMP this can be realized by defining `omp_set_num_threads(2)` before the execution encounters the parallel region.

```
omp_set_num_threads(2);
#pragma omp parallel
{
     $s_1$ 
    for (  $i \leftarrow 0; i < 3; i \leftarrow i + 1$  )
         $s_2$ 
     $s_3$ 
}
```



Above we see that thread 0 and thread 1 have three instances of statement  $s_2$  waiting for the scheduler to be put into execution. The first statement of each thread ( $s_1^0$  and  $s_1^1$ ) is already executed at the current point in time. Next, we show the same code structure but with a preceding worksharing construct.

```
omp_set_num_threads(2);
#pragma omp parallel
{
     $s_1$ 
    #pragma omp for
    for (  $i \leftarrow 0; i < 3; i \leftarrow i + 1$  )
         $s_2$ 
     $s_3$ 
}
```

Algorithmic Differentiation of Pragma-Defined Parallel  
Regions

Differentiating Computer Programs Containing OpenMP

Förster, M.

2014, XI, 405 p. 41 illus., Softcover

ISBN: 978-3-658-07596-5