Checking sequential correctness in shared-memory parallel programs

Gilbert Caplain
e-mail: caplain@cermics.enpc.fr

CERMICS Research Report 99-181
October 1999

Abstract

This research report brings a few complements to a previous paper [4] we will heavily refer to. Most of the results we derive here have been presented in an invited lecture at the *Eighth International Colloquium on Numerical Analysis and Computer Science with Applications* (Plovdiv, Bulgaria, Aug.1999) [3].

1 Introduction

We are interested in a property of *correctness* of programs written in a parallel language. This property is a semantic equivalence between the parallel program and its *sequential version*, that we define. In the framework we are considering, a parallel program is viewed as a parallelization of some *given* sequential program – basically, this parallelization will consist in parallelizing loops and introducing event synchronizations – and we require that the results of any run of the parallel program be identical to those of the given sequential program. In this context, the improvement sought through the parallelization lies only in the ability to run the program faster, by allowing several parts of it to be executed simultaneously, on several available processors. Considering this property is relevant in many numerical analysis applications.

We consider the *shared-memory* programming model, but programs written in our language may be fruitfully compiled towards *distributed-memory* machines as well.

The main result we have obtained is a theorem which ensures this correctness property under some hypotheses, mainly a condition of *preservation of data dependences.* These hypotheses involve the semantics of the sequential version only: in other words, due to our result, checking the correctness of *any run* of some parallel program boils down to checking some properties of its sequential version only. This theorem is presented and derived in [4].

This theorem could be used in the design of a tool to *statically* check the correctness of parallel programs, i.e. perform a check on the program source. A preliminary application, dealing with a subset of our language, is developed in [5, 8].

In some specific cases, in which a parallel program is dependent on input data in some way, a certain *observation* of one sequential run allows to derive whether the program is sequentially correct for any values of the input data. This possibility of *dynamic checking* is explained in Section 3.

In Section 4, we present a possible extension of the event synchronizations considered previously.

In Section 5, we present an extension of our theorem, by introducing *critical sections*, then considering a weakened (i.e. generalized) version of our correctness property.

2 The semantic equivalence theorem

Some of the basic concepts we consider have been introduced in [1]. In the development of shared-memory parallel programs, the most difficult challenge is to avoid *data races*, a circumstance corresponding to *data dependences*. A data dependence links two accesses to the same variable (memory location) when at least one of these accesses is a write. In order to ensure that the parallelized program meets the semantic equivalence requirement, it must be checked that every data dependence is *preserved*, i.e. that the two corresponding accesses operate in the right order (this is the *dependence implies precedence* requirement).

This *dependence preservation* paradigm is fairly well-known, to the point that some of its intricacies may escape at first sight. What does "dependence implies precedence" precisely mean? At first sight, it should be interpreted as "dependence (defined in the sequential version) implies precedence (ensured during a run of the parallel version)". But, what if some statements executed in the sequential version are not executed in some parallel run? or if their execution does not involve the same variables (as input or output) in both runs? Such circumstances may arise since, in the language we are considering, loop bounds and subscript expressions in arrays may contain variables: in other words, we do not limit ourselves to *static control* programs, a limitation adopted very often in the literature. As long as we have not proved the semantic equivalence we are considering, therefore, such intricacies imply that a "dependence implies precedence" requirement *has no well-defined meaning* at this point.

In order to deal with such intricacies, in our search for a static correctness criterion, the semantic equivalence theorem we outline here states sufficient correctness conditions *which refer to the semantics of the sequential version only, i.e. involve predicates that are defined on the sequential version.* (For instance, we address the possibility of execution faults in parallel runs. Only the sequential version is presupposed not to produce execution faults.) Due to our theorem, checking the semantic equivalence between *any* run of some parallel program and the sequential run of this program, boils down to checking some predicates defined on the *sequential* version of this program. Especially, this opens the possibility to use all resources of *dataflow analysis* – which is inherently adapted to the study of sequential programs – in the process of applying our theorem to check the correctness of a parallel program.

The language

The language we consider allows for **parameters**, i.e. "variables" that get a value "once and for all" when the program starts. Thus, in our framework, a *program* in fact represents a "class of programs" differing from one another by the values of parameters. A **program instance** is obtained from a program by assigning constant values to its parameters.

Apart from usual sequential constructs (`IF-THEN-ELSE` conditionals, static `DO` and dynamic `WHILE` loops), the language we consider provides for parallel loops (denoted `PDO`) and event `POST/WAIT`

```
a0:   A(0)=...                  A(0)=...
p0:   post(E(0))                continue
      pdo I=1,N                 do I=1,N
        ...                       ...
w:      wait(E(I-1))              continue
b:      ...=A(I-1)                ...=A(I-1)
c:      A(I)=...                  A(I)=...
p:      post(E(I))                continue
        ...                       ...
      endpdo                    enddo
```

Figure 1: An example of a parallel loop with synchronizations (left) and its sequential version (right)

synchronizations[1]. Figure 1 shows an example of a parallel loop with event synchronizations. The sequential version, on the right, sets the intended semantics: the output `A(I-1)` of statement instance `c(I-1)` is used as input in statement instance `b(I)`, as specified by the sequential order of execution of the DO loop. The synchronization statements `w` and `p` have been introduced in the parallel version, in order to preserve this execution order once the DO is parallelized into a PDO: the WAIT statement `w` at iteration `I` waits for the event `E(I-1)` to have received the value *posted* through the POST statement `p` at iteration `I-1` (or the statement `p0` for `I=1`). This exemplifies the semantics of synchronization statements: a variable type **event** is considered, with two values: *cleared* (the initial value at type declaration) and *posted*. A POST (resp. a CLEAR, not exemplified here) sets value *posted* (resp. value *cleared*) to the event variable it refers to. A WAIT reads the event it refers to: if this event is *posted*, the WAIT continues; if this event is *cleared*, the WAIT *waits* and tries again later. Thus, a WAIT is led to *wait for* the event to be *posted* by the execution of some POST statement, as was intended[2]. Of course, the events `E(I)` must not have been *posted* before this loop without having been *cleared* meanwhile, nor have been *posted* elsewhere in parallel, in case this loop is nested in a larger parallel construct (More on this later).

A *waiting deadlock* occurs when a WAIT waits in vain for an event which never gets posted. In Figure 1, for instance, omitting the statement `p0` leads to a deadlock on statement instance `w(1)` because `E(0)` does not get *posted*.

The *sequential version* of a given parallel program is obtained by transforming PDO into DO and *disabling* the POST, WAIT and CLEAR statements: by "disabling", we mean that, in the sequential version defined here, they are converted into CONTINUE statements which do nothing, but we retain the possibility, in the following developments, to keep track of event references, allowing ourselves to consider what occurs to these references as though they were indeed addressed in a sequential run.

Further developments about the language we consider and about the execution model are provided in [4, Section 2]. Regarding the execution model, let us just mention the fact that, due to

[1]Our language also provides for *parallel sections*, a construct specifying that several sections of code may execute in parallel. We will not consider them specifically here.

[2]The execution of POSTs and WAITs also involves *variable updatings*: variables shared in the parallel construct are updated. In the example shown here, this makes sure that the output `A(I-1)` of `c(I-1)` is indeed available as input in `b(I)`, in the local memory of iteration `I`. We do not address this point here, for the sake of brevity; see [4, §2.2].

resource limitations, iterations of a parallel loop may not *all* execute in parallel at once, but be *loaded* on parallel *processes*, each process having possibly several iterations in charge (which it then executes sequentially). Moreover, within our framework, we will require that the sequential correctness be ensured independently of the number of processes which will turn out to be available for some run of the program – and even in the extreme case when there is only one process available, in which case the parallel program will, in fact, run sequentially (*ordered single process run*). It is easy to derive that the ordered single process run is identical to the run of the sequential version up to a waiting deadlock if any. This is why, in the theorem to follow, considering the *no-waiting-deadlock* hypothesis, the reference to the sequential version takes the form of a reference to the ordered single process run.

Dependences and precedences

Let us briefly outline a few notions which are further developed in [4].

Notion of statement instance [4, §2.1].

Classically (see e.g. [10]), since a statement within a loop may execute several times, each of these executions is termed as a *statement instance*. This usual point of view brings a difficulty in our framework: since our language allows for variables in DO and PDO loop bounds and (obviously) in WHILE test expressions, the set of instances generated by one statement will generally not be known statically. Thus, we have been led to introduce a different definition of a statement instance.

To every statement in the program, is associated a (possibly empty) **index vector**, recursively defined as follows. Let a be a statement. If a is not contained in a loop, its index vector is empty: then, a generates one statement instance. Otherwise, we consider the innermost loop containing a. Let c be the header of this innermost loop, and $\mathbf{i}$ be the (possibly empty) index vector of c. The index vector of a is then obtained as the concatenation of $\mathbf{i}$ and a component j, denoted $\mathbf{i}::j$. Every (executed or not) instance $c(\mathbf{i})$ generates an infinity of instances $a(\mathbf{i}::j)$, as follows: j takes rational integer values if the loop is a DO or PDO (j then corresponds to the iteration index of the loop[3]); j takes positive integer values if the loop is a WHILE.

Through this formalism, a statement contained in a loop generates a countable infinity of statement instances but, in any given run not leading to an infinite loop, only a finite number of them will come to be executed.

Sequential execution predicate $\mathrm{Exe}^{\mathsf{s}}$.

We introduce a predicate, denoted $\mathrm{Exe}^{\mathsf{s}}$, expressing the condition for a statement instance to be executed in the sequential version. Under the extra condition that the sequential program terminates, i.e. does not enter an infinite WHILE loop, $\mathrm{Exe}^{\mathsf{s}}$ is well-defined and its expression is rather straightforward for the language we are considering.

For a statement a and an index vector $\mathbf{i}$ such that the instance $a(\mathbf{i})$ is executed in the sequential version, we may consider the environment in which the execution of $a(\mathbf{i})$ takes place. For any expression *exp* which happens to be evaluated through the execution of $a(\mathbf{i})$, its value is defined in this environment: it will be denoted $[\![exp]\!]@a(\mathbf{i})$. We must emphasize that $[\![exp]\!]@a(\mathbf{i})$ is undefined whenever $a(\mathbf{i})$ is *not* executed. This leads us, in the expressions of $\mathrm{Exe}^{\mathsf{s}}$, to make use of the *sequential conjunction*, denoted &, which differs from the *logical conjunction*, denoted $\wedge$, as follows: if A and

[3]In our language, DO and PDO loops are *normalized*: the increment is set to 1.

B are boolean expressions (taking values *true*, *false* or *undefined*), A & B is false whenever A is false, even if B is undefined, whereas $A \wedge B$ is undefined in this case.

Let a be a statement. $\mathrm{Exe}^{\mathsf{s}}(a)$ essentially depends on the nesting of a in a loop or IF construct. If a is not nested, $\mathrm{Exe}^{\mathsf{s}}(a) = \mathit{true}$. If a is nested, we consider the innermost DO, PDO, IF or WHILE a is nested in.

If the innermost nesting of a is in a DO or PDO loop of header c, with lower and upper bound expressions c.LB and c.UB respectively, let $\mathbf{i}$ denote the (possibly empty) index vector of c and $\mathbf{i}::j$ denote the index vector of a. We then have:
$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i}::j)) = \mathrm{Exe}^{\mathsf{s}}(c(\mathbf{i}))$ & $([\![c.\text{LB}]\!]@c(\mathbf{i}) \leq j \leq [\![c.\text{UB}]\!]@c(\mathbf{i}))$

Expressions of $\mathrm{Exe}^{\mathsf{s}}$ in the other cases – when the innermost nesting of a is in a WHILE or in a IF – are just as straightforward. They are provided in [4, §3.1].

Dependence Dep [4, §3.2].

Considering two statements a and b, indexed by $\mathbf{i}$ and $\mathbf{j}$ respectively, the predicate $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$ expresses that: "in case $a(\mathbf{i})$ and $b(\mathbf{j})$ are both executed in the sequential version, in this order, then they both access one same memory location (not corresponding to an event variable – event references are dealt with differently), at least one of them writing it".

Let $a(\mathbf{i})$ and $b(\mathbf{j})$ be two statement instances respectively involving references *expa* and *expb*, referring to variables (other than event variables), these two references *not both being input references.* In case $a(\mathbf{i})$ executes in the sequential version, $[\mathit{expa}@a(\mathbf{i})]^S$ denotes the variable *expa* refers to during that execution. The relation $\equiv$ between variables to which two distinct references are made, means that they are the same variable. $\ll$ denotes the sequential order. We can give an expression of Dep:

$$\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) = (a(\mathbf{i}) \ll b(\mathbf{j})) \wedge ([\mathit{expa}@a(\mathbf{i})]^S \equiv [\mathit{expb}@b(\mathbf{j})]^S)$$

This expression of Dep is not necessarily defined when $a(\mathbf{i})$ and/or $b(\mathbf{j})$ is not executed in the sequential version – it will then have boolean value *undefined.* This is why we will use Dep in expressions such as:

$$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j})) \ \& \ \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}))$$

making use of the sequential conjunction & introduced above.

Precedences Pre^0**,** $\mathrm{Sync}^{\mathsf{s}}$ **and** $\mathrm{Pre}^{\mathsf{s}}$ [4, §3.3 & 3.4].

The predicate $\mathrm{Pre}^{\mathsf{s}}$ expresses *the precedence relations which would apply in the parallel program, as a consequence of the execution model,* assuming *that the variables involved in the definition of these relations have their "sequential" values.* $\mathrm{Pre}^{\mathsf{s}}$ is obtained from a *control precedence* Pre^0 (the expression of which is rather straightforward, and independent of any variables [4, §3.3]) and a *synchronization precedence* $\mathrm{Sync}^{\mathsf{s}}$ (corresponding to the POST/WAIT pairs). Synchronization precedences $\mathrm{Sync}^{\mathsf{s}}$ are more intricate to consider than control precedences Pre^0. Especially, in order for POST/WAIT pairs to indeed realize precedences as required, there are some conditions to be met. *For instance*, we must avoid the case that several non mutually exclusive POST statement instances are susceptible to trigger the execution of one WAIT statement instance, in which case no synchronization would be warranted between any of these POSTs and this WAIT. Such conditions translate into *requirements about the use of* POST/WAIT/CLEAR *statements*, which are not detailed here for the sake of brevity; they are explained and formally expressed in [4, §3.4]. Let us just mention here that these requirements prescribe that some pairs of synchronization statement instances be related by a control precedence Pre^0.

The semantic equivalence theorem

Let us outline, without proof, the semantic equivalence theorem derived in [4, Section 5]:

Theorem 1 *Under the following hypotheses:*

i. The requirements about synchronizations *alluded to above* [4, §3.4] *;*

ii. No waiting deadlock in the ordered single process run [4, §4.4] *;*

iii. No infinite loop in the ordered single process run;

iv. For all statement instances $a(\mathbf{i})$ *and* $b(\mathbf{j})$*,*
$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j}))\ \&\ \mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j})) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j}))$*;*

the parallel program is semantically equivalent to its sequential version. Especially, no parallel run can deadlock, nor infinitely loop, nor produce an execution fault.

3 Dynamic program checking

The concept of dynamic program checking we will introduce now stems from an important remark about the assumptions of Theorem 1. Let us consider a parallel program instance (obtained by giving values to parameters). *We assume that we know, by any means, that the ordered single process run terminates (i.e. does not enter an infinite* WHILE *loop).* Then, it would be easy to derive that *the problem of the sequential correctness of our program instance is decidable*, by a procedure which consists in *observing* the ordered single process run (a sequential program) *"from outside"*, in order to check the hypotheses of Theorem 1 (other than the termination, assumed here).

Obviously, this observation concerning a program *instance* does not seem very interesting at first sight, since we would like to check the correctness of a parallel program instance through a procedure less costly than running its sequential version ! However, this observation might reveal interesting in the following circumstance: it most often happens that a parallel program is designed to run a great number of times, with different values of *input data*. These data are *parameters* in our sense. It is interesting to consider cases when it may be statically derived (i.e. derived through an analysis of the program source) that *the semantic equivalence property that we are considering here does not depend on the values of data.* Intuitively, these are situations when the "control variables" (loop bounds, tests in IFs and WHILEs, array subscripts) do not depend on data. Such situations can be viewed as a generalization of the notion of *static control program* (see e.g. [7])[4]. In such

[4]In [7], a *static control* program is a program in which loop bounds depend only on constants, indices of loops they are nested in, and *structure parameters*, a set of integer variables defined only once in the program, either as parameters in our sense, or from other structure parameters previously defined. Our situation is different, in the sense that "our" control variables may be modified several times during a program run, even to the point of not being knowable statically – indeed we are not considering a *static control* property, but rather a *control independent of data* one.

cases, it will be relevant to consider one program instance and endeavour to check the correctness of the parallel program by *observing a sequential run* of this instance.

We will provide sufficient conditions under which the correctness of a parallel program does not depend on data; then, we will mention possible applications; afterwards, we will outline the principles of a dynamic verification procedure.

The independence of correctness with respect to data

We consider a parallel program written in our language. We will derive that, under a few assumptions, the correctness (in our sense) of this program does not depend on the values of parameters.

Theorem 2 *We assume that there exists a set Δ of variables and parameters in the program, such that the following assumptions are met:*

i. All parameters belong to Δ ;

The three assumptions to follow refer to the sequential version: we focus on variables the references point to in statement instances executed in the sequential version.

ii. As soon as a reference to an element of Δ stands as input in a statement instance, the output references of this statement instance point to variables of Δ ;

iii. There is no reference to an element of Δ in a DO *or* PDO *loop bound expression, nor in a test of an* IF *or* WHILE *;*

iv. There is no reference to an element of Δ in an array subscript expression.

Then the assumptions of Theorem 1 do not depend on the values of parameters. Therefore, if a program instance meets the assumptions of Theorem 1, any other instance of the program whose sequential version does not produce an execution fault is semantically equivalent to its sequential version.

Proof : We have to derive that, under the assumptions we have just mentioned, those of Theorem 1 do not depend on the values of parameters. Let us first show that Δ contains all variables the values of which depend on those of parameters in the sequential run (*causal closure of* Δ), by deriving that a variable not belonging to Δ cannot be influenced *directly* by variables and parameters belonging to Δ.

During the sequential run of a program instance, let x be a variable not belonging to Δ, written by a statement instance α. There are three ways in which this computation of x might be influenced directly by elements of Δ. **(1)** Such elements might influence the execution of α, i.e. be involved in $\mathrm{Exe}^{\mathsf{s}}(\alpha)$: this is ruled out by *(iii)* [4, §3.1]. **(2)** Elements of Δ might be involved in the references of α, i.e. in the *designation* of the variable x written by α and/or of the input variables used by α. This is ruled out by *(iv)*. **(3)** At last, elements of Δ might be inputs of α. This is ruled out by *(ii)*. The causal closure of Δ is thus derived.

As a consequence, due to *(iii)*, the predicate $\mathrm{Exe}^{\mathsf{s}}$ does not depend on parameters, nor does the program termination [4, §3.1]. Due to *(iv)*, the predicate Dep does not depend on parameters either (Section 2). We know that Pre^0 is independent of any variable [4, §3.3]. Due to *(iv)*, the event references are independent of parameters; therefore, considering the fact that $\mathrm{Exe}^{\mathsf{s}}$ is independent of parameters, all the mechanism of synchronizations (considered in the sequential semantics, as required by Theorem 1), is independent of parameters: so, the requirements about synchronizations and the no-deadlock condition, involved in Theorem 1, are independent of parameters as well. ◀

Let us mention a weakness of this result: the fact that *test expressions* are required not to depend on data is certainly a limitation in many cases.

A few examples

This *"control independent of data"* property will often be present in numerical analysis applications. For instance, let us consider finite-element programs. We wish to solve a system of equations on a certain physical domain, by an approximation making use of a mesh of this domain and computing a discrete set of values associated to this mesh – for instance, values at the nodes. It will often occur that the parameters defining the mesh (number of nodes, contiguities...) are the control parameters of the program, determining the number of loop iterations and interfering in the subscripts of vectors and matrices; whereas the quantities to be computed at the nodes will be "data dependent" variables, elements of Δ in our sense: variables not involved in the control structure of the program.

Then, considering a finite-element parallel program, in such circumstances, the sequential correctness of this program will not depend (in the case, say, of time evolution) on the initial conditions nor the boundary ones, but only on the control structure of the program, related to the definition of the mesh.

We could even consider a finite-element parallel program allowing for a (continuous) deformation of the mesh, in order to adjust to physical domains of various shapes (but same topology). Under the condition that the deformation and adjustment parameters are "data-dependent" in our sense, the sequential correctness of the parallel program will be an invariant of these deformations. However, obviously, a change in the equations, for instance in order to solve a different physical problem using the same mesh, may modify the dependence relations Dep, thus creating a situation in which the sequential correctness property is not invariant.

Dynamically checking a program instance

In light of the above considerations, in many cases, it may be relevant to try a *dynamic* checking of a program instance, through the observation of a sequential run (or, more accurately, of an ordered single process run).

Let us consider a parallel program the sequential version of which meets the assumptions of Theorem 2 and is guaranteed to terminate. This is the situation when it may be relevant to endeavor a dynamic checking procedure. This procedure can be described as follows.

First, we make a static analysis of the program source, in order to try to check the assumptions of Theorem 1. We keep track of those assumptions which we do not succeed to check at this point – e.g. possible dependences which do not seem to be preserved by precedences, or an uncertainty about the no-deadlock condition.

We then consider "valid" values of parameters, i.e. values such that the sequential version of the corresponding program instance does not produce an execution fault; and we consider the corresponding ordered single process run. *Or rather, alternatively,* if we can, we will prefer to consider an *ordered single process run with dummy parameters*, i.e. a run during which, in the succession of executed instances, the computations of the variables belonging to Δ are not performed, in order to save time. This is possible in principle, due to the assumptions of Theorem 2.

For instance, let us consider the statement:

```
A(I,J+1) = B(I,J-N(I))*N(I-1)
```

where `A()` and `B()` are variable arrays of Δ, `N()` is an array of "control" variables, not belonging to Δ, and `I` and `J` are indices of the two loops this statement is nested in. If e.g. the instance `I=2,J=3` of this statement is executed, a *partial evaluation*, of the control expressions, is performed. First the values of the loop indices are considered, which leads to instanciate the statement:

```
A(2,3+1) = B(2,3-N(2))*N(2-1)
```

Then, the control variables are computed – let us assume that, when this instance is executed, we have got for example: `N(1)=3` and `N(2)=6` ; computing the subscript expressions, we get:

```
A(2,4) = B(2,-3)*3
```

At this point, we refrain from evaluating the variables of Δ, and we keep track of the fact that this statement involves:

- as output: `A(2,4)`
- as input: `B(2,-3), N(2), N(1)`

During this single process run, the following observations are performed:

- A deadlock situation on a WAIT is observable – the ordered single process run just stops at that point. (If we are rather observing the sequential version, in which the synchronizations are "disabled" but still examined, it is easy to check, for every visited WAIT instance, that the involved event is "posted" at that point.)
- Along the run, the event references appearing successively are recorded, which will permit to determine the synchronizations pairs Sync^{s} as well as the pairs of synchronization instances between which a precedence Pre^0 will have to be checked (Section 2 and [4, §3.4]). The search for these pairs of instances involves binary comparisons between references; therefore, its complexity is polynomial in the number of instances examined.
- As the execution proceeds through the parallel constructs that the static analysis did not succeed to "validate", variable references (other than event ones) are recorded, as we have seen in the example above, in order to record the dependence relations Dep the preservation of which could not be derived statically. More precisely, the static analysis detected and recorded all statement pairs (a, b) possibly involved in a possibly unpreserved dependence $\text{Dep}(a, b)$; then, the dynamic analysis records pairs (α, β) of instances of a and b respectively, indeed involved in a dependence $\text{Dep}(\alpha, \beta)$. The search for these pairs (α, β) involves binary comparisons between references; therefore, its complexity is polynomial in the number of instances examined thus.

Once these steps are performed, we have to solve two problems involving graphs.

- check whether the Dep edges recorded belong to the transitive closure of the graph formed with the Sync^{s} edges recorded and the Pre^0 edges (straightforwardly representable) (with the restriction that the paths considered do not end with Sync^{s} edges: for this technicality, see [4, §3.3]). This is a rather well-known graph problem, involving *breadth-first* search algorithms. See e.g. [6, 9]. The complexity of these algorithms is polynomial in the number of nodes, which is (in the worst cases) the number of executed statement instances located in the parallel constructs that the static analysis could not "validate".
- considering the synchronization instance pairs (θ, γ) for which a control precedence Pre^0 is required, check that we indeed get $\text{Pre}^0(\theta, \gamma)$ or $\text{Pre}^0(\gamma, \theta)$. This graph problem is easier than the previous one, due to the simplicity of the expressions of Pre^0.

The dynamic verification procedure outlined here has an algorithmic complexity which, in the worst cases, is polynomial in the number of executed statement *instances* – hence, this complexity will be "big" in many interesting cases. This is why it will be important to perform static analysis first, in order to focus the dynamic checking procedure on the difficult points; and to apply such a procedure only to programs designed to run a sufficient number of times.

4 Event synchronizations with guards clauses

In our execution model [4, §2.2], we have mentioned that, whenever a POST or a WAIT executes in a parallel construct, variables shared in this construct are updated. In order to avoid useless updates, it may be of interest to limit these updates to one or several references specified in the POST or WAIT statement (the specification of such references to be updated is classically termed as a *guards clause*; such references will be said to be *guarded* by the synchronization statement). Here we will adopt the following syntax:

```
POST  (<event>, <reference-list>)
WAIT  (<event>, <reference-list>)
CLEAR (<event>)
```

The `reference-list` following the event reference in the argument list of the POST and WAIT statements, contains the guarded references: the references (in the execution environment) of the variables which will be updated. Figure 2 shows an example.

In this example, the synchronization aims at ensuring that the value of `A(I-1)` used as input by statement instance `b(I)` is the value computed by statement instance `a(I-1)` (or, for `I=1`, by statement `a0`) ; this time, the precedence ensured thus will not involve any other variables. It is possible to introduce such synchronizations with guards clause in our formalism: our results remain basically unchanged.

In such an extension, our predicates Dep, Sync^{s} and Pre^{s} would be modified as follows.

The dependences Dep (Section 2) will now involve three arguments: the two statement instances as previously, *and* the variable reference involved in the dependence, in the sequential version:

```
a0:     A(0)=...
p0:     post (E(0),A(0))
        pdo I=1,N
          ...
w:        wait (E(I-1),A(I-1))
b:        ...=A(I-1)
a:        A(I)=...
p:        post (E(I),A(I))
          ...
        endpdo
```

Figure 2: An example of post/wait synchronization with guards clause

$$\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}), u) = (a(\mathbf{i}) \ll b(\mathbf{j})) \wedge ([\mathit{expa}@a(\mathbf{i})]^S \equiv [\mathit{expb}@b(\mathbf{j})]^S \equiv u)$$

The predicate $\mathrm{Sync}^{\mathsf{s}}$, too, will involve three arguments: there again, the third argument is a variable reference involved (in the sequential version) in the guards clauses of the POST and the WAIT instances of the synchronization pair. The requirements about the use of synchronization statements (Section 2 and [4, §3.4]) remain unchanged (for instance, we must still rule out the possibility that two non mutually exclusive POST statement instances be able to trigger one WAIT statement instance, even though different references would be guarded in these two instances). Under these assumptions, a predicate $\mathrm{Sync}^{\mathsf{s}}(\pi, \gamma, u)$ expresses that we have $\mathrm{Sync}^{\mathsf{s}}(\pi, \gamma)$ in the previous sense – i.e. assuming the sequential semantics, the execution of π triggers the execution of γ: (π, γ) is a synchronization pair – and furthermore, that u is a variable pointed to by a guarded reference of π *and* by a guarded reference of γ, under the sequential semantics.

The predicate $\mathrm{Pre}^{\mathsf{s}}$ will be obtained from the control precedence Pre^0 (which remains unchanged) and the "guarded" synchronizations $\mathrm{Sync}^{\mathsf{s}}$, by a transitive closure, as previously [4, §3.3], through relations such as:

$$\begin{aligned}&\mathrm{Pre}^0(\alpha, \pi_1) \wedge \mathrm{Sync}^{\mathsf{s}}(\pi_1, \omega_1, u) \wedge \mathrm{Pre}^0(\omega_1, \pi_2) \wedge \\ &\qquad \ldots \wedge \mathrm{Sync}^{\mathsf{s}}(\pi_n, \omega_n, u) \wedge \mathrm{Pre}^0(\omega_n, \beta) \quad \Rightarrow \quad \mathrm{Pre}^{\mathsf{s}}(\alpha, \beta, u)\end{aligned}$$

Then, in Theorem 1, the dependence preservation assumption becomes:

For all statement instances $a(\mathbf{i})$ and $b(\mathbf{j})$ and for all variables u,

$\mathrm{Exe}^{\mathsf{s}}(a(\mathbf{i})) \wedge \mathrm{Exe}^{\mathsf{s}}(b(\mathbf{j}))$ & $\mathrm{Dep}(a(\mathbf{i}), b(\mathbf{j}), u) \Rightarrow \mathrm{Pre}^{\mathsf{s}}(a(\mathbf{i}), b(\mathbf{j}), u)$

Adapting the proof of Theorem 1 [4, Section 5] to such extensions is rather straightforward (though tedious).

5 Introducing critical sections

Critical sections are a parallel structure that we have not considered yet. Let us consider the simple example of Figure 3.

The sequential loop on the left is designed to compute the $A(I)$'s and provide their sum, in T. Here $A()$ has a numeric type (integer or real), the same as T. The sum T obtained at the end of the loop does not depend on the order in which the $A(I)$'s are added[5]. Assuming that the computations of $A(I)$ are independent and can therefore be performed in parallel, it may be interesting to parallelize the loop in such a way that the order in which the instances of `s` execute is not constrained. *However* we must ensure that these instances *do not execute simultaneously*, and that the intermediate values obtained for T are transmitted between these consecutive instances. This is the function of the *critical section* here introduced in the parallel version of the loop (right of Figure 3).

The variable V standing in the statement `critical section` is a **lock**, the possible values of which are *locked* and *unlocked*. One lock variable V must be shared by all critical section instances which, precisely, should not execute at the same time. The variables mentioned after V in the argument list of the `critical section` statement – in the example, variable T – are those variables *guarded* in the critical section

In order to ensure the right behavior, the execution model of critical sections is as follows.

- Initially, the value of the lock V is *unlocked*.
- Whenever a process reaches a CRITICAL SECTION instance, it tests the value of V. If this value is *locked*, the process *waits* and performs the same test again later (similarly to what a WAIT does when the involved event is *cleared*) – presumably, another critical section instance sharing the same lock is running right at that moment. If this value is *unlocked*, the process sets value *locked* to V, inputs from the shared memory the value(s) of the guarded variable(s), and continues execution (thus *entering* the critical section).
- Whenever a process reaches a END CRITICAL SECTION instance, it outputs to the shared memory the value(s) of the guarded variable(s), then sets value *unlocked* to V (which had value *locked* at that moment), then continues execution (thus *exiting* the critical section).

Introducing critical sections here, constitutes an enlargement of our previous framework, in the following sense. Previously, we were interested in a property of *semantic equivalence* between a parallel program and its sequential version: a requirement that both programs perform the same computations, producing the same outputs, not necessarily in the same order. In the contrary, considering critical sections, the idea is *not to require that some intermediate values (of T, in the above example) be identical to those obtained in the sequential version*, provided that the *final* value of T, obtained at the end of the loop, is the same in both versions.

Let us specify how the previous results could be extended so as to allow the use of critical sections in the programs considered here.

[5]Here, we *neglect* the well-known problems arising from the fact that e.g. addition of reals, as implemented, is not necessarily commutative nor associative – due to approximations in the representation of reals.

```
        T=0                       T=0
        do I=1,N                  pdo I=1,N
          ...                       ...
          A(I)=...                  A(I)=...
                                    critical section(V,T)
s:        T=T+A(I)                    T=T+A(I)
                                    end critical section(V,T)
          ...                       ...
        enddo                     endpdo
```

Figure 3: An example of a critical section. The sequential version is shown on the left.

We will not deal with critical sections in their most general form, but we will study a fairly general scheme, represented in Figure 4. The element of program displayed here may be itself nested in a larger parallel construct.

Here, $A()$ and T have not necessarily a numerical type (integer or real), and may correspond to more complex structures. We will focus on variable T, which is both an input and output variable in the critical section. (Such a variable is termed as a *reduction variable.*) It would be straightforward to show that this scheme can represent cases when there are several statements in the critical section – in such cases, these statements are *condensed into* the function *func()* – provided that we correlatively condense into variable T any memory location which is both read and written by *func()*, and into variable $A(I)$ any memory location which is just read by *func()*. (The underlying idea here, is that T must contain *everything which is transmitted from a critical section instance to the next.*)

Using this critical section pattern may reveal useful whenever the computation of $A(I)$ by `a` is time consuming whereas the call to *func* is relatively fast.

Before mentioning the validity conditions of this parallelization, let us give an example: *constrained optimization.* Every loop iteration I searches for an admissible solution to some constraint problem, accompanied with a *cost.* Both the admissible solution (if any) and its cost are assigned to an adequately typed variable $A(I)$. If iteration I finds no solution, $A(I)$ is assigned a value denoted $\perp$. The variable T, typed similarly to $A(I)$, is initialized at $\perp$. *func*(,) is a function whose two arguments have that same type, and which returns as output the argument which is "minimal" for the ordering defined as follows: the order of the associated costs, completed by the data that $\perp$ is "greater" than the other values. At the end of the loop, T contains $\perp$ whenever no admissible solution has been found, and a solution with minimal cost in the contrary case. In such a situation, and provided that no two solutions have equal minimal costs (more on this in a moment), this parallel loop returns the same result as its sequential counterpart.

In this critical section scheme, it is straightforward to see that the (weakened) semantic equivalence holds under the two following conditions:

- All dependences which stand between iterations of this loop, or between any one of them and the rest of the program, are preserved by precedences, with the exception of a dependence $\mathrm{Dep}(s,s)$ involving the variable `T` (the expression of which is: $\mathrm{Dep}(s(i),s(j)) = (i < j)$ –

```
          T=eps
          ...
          pdo I=1,N
            ...
a:          A(I)=...
            critical section(V,T)
s:            T=func(A(I),T)
            end critical section(V,T)
            ...
          endpdo
```

Figure 4: A critical section scheme

remember the reference to the sequential version in the definition of dependences). As a consequence, in the loop, `T` is neither read nor written elsewhere than in statement `s`.

- The function *func()* has a property of *iterative symmetry*, which means that the final result T of the application of *func()* to the N values $A(I)$ (T being initialized at *eps*), does not depend on the order in which these N values are dealt with.

Indeed, due to the first condition, the intermediate values of T, which are not necessarily the same in some parallel run **P** being considered, as in the sequential run **S**, are not used elsewhere that in the successively executed instances of s. The critical section ensures that, in **P**, the instances of s execute consecutively and that the intermediate values of T are duly transmitted from every instance to the next one executed. Finally, due to the second condition, these different computations of T provide the same value at the loop exit.

Weakening the semantic equivalence requirement

In the constrained optimization problem we have just considered as an example, we have mentioned the limitation that no two admissible solutions of minimal cost should exist. If this limitation does not stand, the function *func()* has not the iterative symmetry property. This simple remark suggests some possibilities to extend our framework a bit further.

In [4] and in the previous sections of this paper, we have studied a *semantic equivalence* requirement, according to which all variables should undergo the same computations, and therefore receive the same values, in any parallel run **P** as in the sequential run **S**. We have just introduced critical sections, as a means to weaken our semantic equivalence requirement in the following way: some intermediate computations are no longer required to be identical in the two runs **S** and **P**, provided that, due to an *iterative symmetry* property, the "final" results of these intermediate computations are still the same in both runs.

We could consider a further generalization of this semantic equivalence requirement, along the following line: we still prescribe that the sequential run **S** and any parallel run **P** share some property pertaining to the computations they make, this common property not necessarily being the semantic equivalence we have considered before (the "maximal" requirement which may be prescribed in this context).

In the example previously mentioned, the constrained optimisation with possibility of equal-cost solutions, we obtain some weakened semantic equivalence, which can be expressed as follows: "whenever **S** finds that there is no admissible solution, so does **P**; whenever **S** finds an optimal solution, **P** too finds an optimal solution, *which is not necessarily the same.*" It may be the case that this weaker equivalence fits the needs of the programmer. It could be interesting to find out under which conditions such generalized properties of *weakened semantic equivalence* could be derived through a process of "recurrence along a parallel execution" analogous to the one followed in the derivation of Theorem 1 [4, Section 5].

References

[1] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, march 1990. ACM Press.

[2] G. Caplain. *Propriétés de correction séquentielle dans un langage parallèle à mémoire partagée.* PhD thesis, Ecole Nationale des Ponts et Chaussées, septembre 1998. http://cermics.enpc.fr/theses/.

[3] G. Caplain. Checking sequential correctness in shared-memory parallel programs. In *The Eighth International Colloquium on Numerical Analysis and Computer Science with Applications*, Plovdiv, Bulgaria, August 1999. Proceedings to be published.

[4] G. Caplain. Correctness properties in a shared-memory parallel language. *Submitted for publication*, 1999. CERMICS research report 99-180. http://cermics.enpc.fr/reports/.

[5] G. Caplain, R. Lalement, and T. Salset. Checking the serial correctness of control-parallel programs. In *Parallel Architectures and Languages Europe*, pages 741–744, Athens, Greece, July 1994. Springer Verlag, LNCS 817.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1991.

[7] P. Feautrier. Dataflow analysis of array and scalar references. *Int. Journ. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[8] T. Salset. *Correction séquentielle de programmes parallèles dans le modèle asynchrone et mémoire partagée.* PhD thesis, Ecole Nationale des Ponts et Chaussées, juillet 1997. http://cermics.enpc.fr/theses/.

[9] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[10] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.