Four barrier algorithms verified

Wim H. Hesselink, whh598

September 24, 2023

Abstract

Keywords: concurrency, barrier

1 Introduction

Barrier synchronization is a classical synchronization task in concurrency, see, e.g., [1, 3.4], [2].

When a task is distributed over several threads, it is often the case that, at certain points in the computation, the threads must wait for all other threads before they can proceed with the next part of the computation. This is called *barrier synchronization*. We model the problem by letting each thread execute the infinite loop

(0) **loop of thread** p **is**
    $NCS(p)$;
    $Barrier(p)$
end loop .

Here, *NCS* stands for a terminating noncritical section, a program fragment that eventually always terminates, but that in all other aspects is irrelevant for the problem at hand.

The threads may only pass the barrier when all of them have completed the last NCS. On the other hand, when all of them have completed the last NCS, they all must pass the barrier and start the next NCS.

In this paper four barriers are discussed. First, a symmetric barrier, i.e., a barrier in which all threads are treated in the same way. This barrier was published first by Lamport in [3, Fig. 7]. Its performance is better than one might expect.

Second, a FAI barrier. Here FAI stands for the "hardware instruction" fetch-and-increment. This one seems to have the best performance.

Third, a ring barrier. This solution is not symmetric, but every thread has the same amount of work and waiting to perform. Each thread waits two times in every call of the barrier. The idea may be elegant, but the performance is bad.

Fourth, a tree barrier. Here the root of the tree has a special responsibility. Yet the total amount of waiting is roughly the same as in the ring barrier. One may expect better performance than the ring barrier, but this depends on the shape of the tree (and the ordering of the children).

1.1 Formalization

To formalize the concept of barrier, assume each thread p counts the barriers that it has executed by a private ghost variable cnt_p which is initially 0 and is incremented in every call of $barrier(p)$ by one.

The barrier is now specified by the requirement that a thread that runs ahead with cnt, must not execute *NCS* but wait. This is expressed in the barrier condition that, for all threads q and r,

$$BC: \quad q \textbf{ in } NCS \Rightarrow cnt_q \leq cnt_r \ .$$

Justification: condition BC prohibits any thread q to enter a new NCS unless the other threads r have exited the old NCS and reached the barrier. Conversely, every barrier in the informal sense satisfies the barrier condition. This is shown by defining cnt_p as the number of times thread p has entered its barrier. Then BC is an invariant of the system, because it can only be invalidated by thread q when it exits the barrier with $cnt_r < cnt_q$. Then q has called the barrier more often than r. Therefore, q needs to wait for r to reach the barrier.

Note that cnt is a ghost variable that can be incremented anywhere in the barrier, possibly in an atomic command that modifies a shared variable. As suggested by the above analysis, the natural location for the incrementation of cnt_p is at the start of $Barrier(p)$. Yet, in all four correctness proofs of barriers below, it is more convenient to combine this incrementation with one of the critical assignments of the algorithm.

There is of course also a progress requirement: when each thread has terminated its NCS, then eventually all threads pass the barrier.

1.2 Testing safety of a barrier

The safety of a barrier implementation can be tested by declaring a new shared integer array `test`[], initially all zeros, and including at the end of $NCS(p)$ a testing procedure

> *testing*(*thread* p) =
>   **for each thread** k **do** `assert`(`test`[p] $\le$ `test`[k]) **endfor** ;
>   `test`[p] := `test`[p] + 1 .

Indeed, every barrier passes this test, i.e., never causes an assert failure. This is because `test`[q] $= cnt_q$ when q is in NCS, while $cnt_r \le$ `test`[r] always holds.

On the other hand, if a potential barrier *bar* always passes this test, the combinations

> `test`(p)`++`; *bar*(p)

satisfies the barrier condion when `test` is regarded as the ghost variable cnt. This shows that *bar* is a barrier, provided it satisfies the progress requirement.

2 Some implementations of barriers

In all cases the system has N threads, numbered from 0 upto $N-1$. Each thread p has a persistent ghost variable cnt_p, which is initially 0. In all four cases, the proof of correctness consists of a proof of the barrier condition BC by means of invariants, followed by a proof of absence of deadlock.

2.1 A symmetric implementation

Perhaps the simplest solution is to introduce a shared array `tag` such that `tag`[p] indicates the value of the unbounded integer cnt_p. This solution was described by Lamport in [3, Fig. 7]. I had found this solution around the year 2000, and had presented in a course on concurrent programming, that I gave in the years 2001-2005. See Lamport's Writings [4, nr. 164].

> *int* `tag`[N] = ([N] 0) ;

In $Barrier(p)$, the value of `tag`[p] is incremented with 1, modulo some constant $R > 2$. Then thread p waits until all other threads have done the same incrementation.

> *Barrier*(p) :
>   *int old* := `tag`[p] ;
>   `tag`[p] := (*old* + 1) **mod** R ;
> cnt_p`++` ;
>   **for each thread** kk **do**
>   **await** (`tag`[kk] $\ne$ *old*)
>   **endfor**
> **end** *Barrier* .

The **for** loop is written in such a way that each thread can read the numbers $\texttt{tag}[k]$ in its own order. Indeed, to avoid memory contention, it seems to be advantageous that the orders of inspection of the threads differ. One can e.g. give each thread a fixed permutation of the thread identifiers to specify its order of inspection.

To prove the correctness of this barrier, we include it in the loop (0) to get

```
        loop of thread p is
11          NCS(p) ;
12          old_p := tag[p] ;
            tag[p] := (old_p + 1) mod R ;
            cnt_p++ ;
            lis_p := allthreads ;
13          while exists kk_p ∈ lis_p do
14              await (tag[kk_p] ≠ old_p) ;
                remove kk_p from lis_p
            endwhile ;
        end loop .
```

Here the index p is attached to the local variables of thread p. Line numbers are introduced, starting with 11 for the ease of using query-replace in the PVS proof script. The private variable pc_p gives the line number of the command that thread p has to execute next. The four assignments of line 12 can be atomically combined because the share variable $\texttt{tag}[p]$ is written only by thread p, and old_p, cnt_p, and lis_p are local variables of thread p.

The set lis_p holds the threads for which thread p still has to inspect the $\texttt{tag}$. The local variable kk_p is used to hold the index thread p will be inspecting. In line 14, the removal can be atomically attached to the waiting, because kk_p and lis_p are local variables of thread p.

The initial condition of the transition system is

$$\forall\, q : pc_q = 11 \ \wedge\ cnt_q = 0 \ \wedge\ \texttt{tag}[q] = 0 \ .$$

As *NCS* is at line 11, the barrier condition *BC* is implied by the predicate

$$\mathit{Iq1}: \qquad q \in [11, 12] \ \Rightarrow\ cnt_q \le cnt_r \ .$$

A complete family of invariant predicates is constructed to prove that *Iq1* is invariant.

Predicate *Iq1* is threatened only by step 13 when lis_p is empty and thread p jumps to 11. It has the remedy

$$\mathit{Iq2}: \qquad q \in [13, 14] \ \Rightarrow\ r \in lis_q \ \vee\ cnt_q \le cnt_r \ .$$

Predicate *Iq2* is threatened only by step 14. It has the remedy

$$\mathit{Iq345}: \qquad q \in [13, 14] \ \wedge\ cnt_r < cnt_q \ \Rightarrow\ \texttt{tag}[r] = old_q \ .$$

Predicate *Iq345* is logically implied by the three predicates

$$\begin{array}{ll} \mathit{Iq3}: & q \in [13, 14] \ \Rightarrow\ \texttt{tag}[q] = (old_q + 1) \ \mathbf{mod}\ R \ , \\ \mathit{Iq4}: & \texttt{tag}[q] = cnt_q \ \mathbf{mod}\ R \ , \\ \mathit{Iq5}: & cnt_q \le cnt_r + 1 \ . \end{array}$$

This implication is proved as follows:

$$\begin{array}{ll} & q \in [13, 14] \ \wedge\ cnt_r < cnt_q \\ \Rightarrow & \{\mathit{Iq5}\} \quad q \in [13, 14] \ \wedge\ cnt_r + 1 = cnt_q \\ \Rightarrow & \{\mathit{Iq4}\} \quad q \in [13, 14] \ \wedge\ (\texttt{tag}[r] + 1) \ \mathbf{mod}\ R = \texttt{tag}[q] \\ \Rightarrow & \{\mathit{Iq3}\} \quad (\texttt{tag}[r] + 1) \ \mathbf{mod}\ R = (old_q + 1) \ \mathbf{mod}\ R \\ \Rightarrow & \{\texttt{tag}[q] < R \wedge old_q < R\} \quad \texttt{tag}[r] = old_q \ . \end{array}$$

The predicates *Iq3* and *Iq4* are inductive. Predicate *Iq5* is threatened only by step 12. It has *Iq1* as remedy. This concludes the construction of a complete family of invariant predicates. Therefore the algorithm datisfies the barrier condition *BC*.

We still have to prove liveness, that is absence of deadlock. Let a state be called a *deadlock state* iff no thread can do a step. As the loop in the barrier is bounded by N, absence of deadlock states is enough to infer deadlock freedom.

Theorem 1 *Assume that* $R > 2$. *Then deadlock states are not reachable.*

Proof. Assume that the state is in a reachable deadlock state. As the state is reachable, all invariants are applicable. Every thread is in $[11, 14]$. Every thread in $[11, 13]$ can do the step of its line number. Therefore, all threads are at line 14. Let p be a thread with the smallest counter, i.e., with $cnt_p \leq cnt_q$ for all threads q. As thread p is blocked at line 14, it satisfies $\mathtt{tag}[kk_p] = old_p$. Put $r = kk_p$. The invariant *Iq3* then implies $\mathtt{tag}[p] = (\mathtt{tag}[r] + 1)$ **mod** R.

On the other hand, by *Iq5* and minimality of cnt_p, one has $cnt_p \leq cnt_r \leq cnt_p + 1$. It follows that $cnt_p = cnt_r$ or $cnt_p + 1 = cnt_r$. Finally, application of *Iq4* on both p and r gives a contradiction with $R > 2$. □

This symmetric implementation has the disadvantage that all threads have to pass N (or $N-1$) await statements.

2.2 Using fetch and increment

The next barrier is essentially the same as the sense-reversing centralized barrier of [5, Fig. 8]. It uses the special atomic instruction fetch-and-increment that increments an integer variable and returns its previous value. Here, the shared integer variable `count` is used to count the number of threads that have arrived at the barrier. The threads at the barrier wait for a shared boolean `sense`.

```
   int count := 0;
   bool sense := true;

Barrier(p):
   bool nef := not sense;
   if fetch_and_increment(count, 1) = N - 1 then
      count := 0;
      sense := nef
   else
      await (sense = nef)
   endif
```

In comparison with [5], here, the value of `count` is replaced by $N -$ `count`, and the persistent private variable *local-sense* is replaced by a local variable *nef*.

In compile-time, this barrier is just as symmetric as the barrier of Section 2.1: all threads are treated in the same way. At at runtime, however, the symmetry is broken: there is a unique thread that increments `count` to N and toggles the shared variable `sense`.

For the sake of the proof, two ghost variables are introduced. The ghost variable `guests` holds the set of threads that are blocked at the final await statement. The ghost variable `butler` is introduced to express that there is at most one thread in the then-branch of the conditional. If there is such a thread, it is the `butler`; otherwise `butler` $= N$.

```
Transition system
ghost variables
    butler: [0..N]
    guests: set of thread

Loop of thread p:
```

```
11   NCS
12   nef_p := not sense;
13   temp := count;
     count++; cnt_p++;
     if temp < N - 1 then add p to guests
     else butler := p endif;
     if temp = N - 1 then
14      count := 0;
15      sense := nef_p;
        butler := N ; guests := emptyset
     else
16      await (sense = nef_p)
     endif
endloop
```

The initial condition is

$$\texttt{count} = 0 \ \wedge \ \texttt{butler} = N \ \wedge \ \texttt{guests} = \emptyset \\ \wedge \quad \forall q : q \in [11] \ \wedge \ cnt_q = 0 \ .$$

The properties of the `butler` are expressed by the invariants

Iq1 : $\texttt{butler} < N \ \Rightarrow \ \texttt{butler} \in [14, 15]$,
Iq2 : $q \in [14, 15] \ \Rightarrow \ q = \texttt{butler}$.

Predicate *Iq1* is inductive. Predicate *Iq2* is threatened only by step 13. It has the remedies

Iz1 : $q \in [13] \ \Rightarrow \ \texttt{count} < N$,
Iq3 : $\texttt{butler} < N \ \wedge \ q \neq \texttt{butler} \ \Rightarrow \ q \in [16]$.

Predicate *Iz1* is implied by *Iq1*, *Iq3*, and

Iq4 : $\texttt{butler} = N \ \Rightarrow \ \texttt{count} = \#\texttt{guests}$,
Iq5 : $q \in \texttt{guests} \ \Rightarrow \ q \in [16]$.

Predicate *Iq3* is threatened only by the steps 13 and 16. At step 13, it has the remedies *Iq5* and

Iz2 : $q \in [13] \ \wedge \ \texttt{count} = N - 1 \ \Rightarrow \ \{q\} \cup \texttt{guests} = \mathit{allthreads}$.

This predicate is implied by *Iq1*, *Iq3*, *Iq4*, and *Iq5*. At step 16, the predicate *Iq3* has the remedy

Iq6 : $\mathit{nef}_q = \texttt{sense} \ \Rightarrow \ \texttt{butler} = N$.

Predicate *Iq4* is threatened only by the steps 13, 14, and 15. At the steps 13 and 14, it has the respective remedies *Iq5* and *Iq2*. At step 15, it has the remedy

Iq7 : $q \in [15] \ \Rightarrow \ \texttt{count} = 0$.

Predicate *Iq5* is threatened only by step 16. It has the remedy

Iq8 : $q \in \texttt{guests} \ \Rightarrow \ \mathit{nef}_q \neq \texttt{sense}$.

Predicate *Iq6* is threatened only by step 13. It has the remedies *Iq8* and

Iq9 : $q \in [13] \ \Rightarrow \ \mathit{nef}_q \neq \texttt{sense}$.

Predicate *Iq7* is threatened only by step 13. It has the remedies *Iq2* and *Iq3*.
Predicate *Iq8* is threatened only by step 13. It has the remedy *Iq9*.
Predicate *Iq9* is threatened only by step 15. It has the remedies *Iq2* and *Iq3*.
This concludes the proofs of the invariants *Iq1* up to *Iq9*.

For the proof of the barrier condition, we need invariants about the variables cnt_q. The barrier condition itself is generalized to

$Jq1: \quad q \in [11,13] \;\Rightarrow\; cnt_q \le cnt_r$.

It is threatened only by steps 16 and 15, and has the respective remedies

$Jq2: \quad q \in [16] \;\wedge\; nef_q = \texttt{sense} \;\Rightarrow\; cnt_q \le cnt_r$,
$Jq3: \quad q \in [14,15] \;\Rightarrow\; cnt_r = cnt_q$.

Predicate *Jq2* is threatened only by the steps 13 and 15. At step 13 it has the remedy *Iq9*, at step 15 the remedy *Jq3*.

Predicate *Jq3* is threatened only by step 13. At step 13 one uses the remedies *Iq2* and *Iq3* to handle the case that the acting thread goes to line 16. If the acting thread goes to line 14 and $r \neq q$, then *Iz2* implies that $r \in \texttt{guests}$. By *Iq5* and *Iq8*, it follows that r is in line 16 and $nef_r \neq \texttt{sense}$. Finally, one uses the new predicate

$Jq4: \quad q \in [11,13] \;\wedge\; r \in [16] \;\wedge\; nef_r \neq \texttt{sense} \;\Rightarrow\; cnt_q + 1 = cnt_r$.

Predicate *Jq4* is threatened only by the steps 13, 16, and 15. At step 13, it has the remedy *Jq1*. At step 15, it has the remedies *Iq2*, *Iq3*, and *Iq6*. At step 16, it has the remedy

$Jq5: \quad q \in [16] \;\wedge\; nef_q = \texttt{sense} \;\wedge\; r \in [16] \;\wedge\; nef_r \neq \texttt{sense} \;\Rightarrow\; cnt_q + 1 = cnt_r$.

Predicate *Jq5* is threatened only by the steps 13 and 15. At step 13, it has the remedies *Iq9*, *Jq1*, and *Jq2*. At step 15, it has the remedies *Iq2* and *Iq6*. This concludes the proof of the invariants *Jq1* up to *Jq5*, and thus of the barrier condition.

Two more invariants are needed for the proof of absence of deadlock.

$Kq1: \quad q \in [16] \;\wedge\; nef_q \neq \texttt{sense} \;\Rightarrow\; q \in \texttt{guests}$,
$Kq2: \quad \texttt{count} < N \;\vee\; (\texttt{butler} < N \;\wedge\; \texttt{butler} \in [14])$.

Predicate *Kq1* is threatened only by steps 13 and 15. At step 13, it has the remedies *Kq2* and *Iq3*, at step 15 the remedies *Iq2* and *Iq6*. Predicate *Kq2* is threatened only by step 15. It has the remedy *Iq2*.

This concludes the proofs of *Kq1* and *Kq2*. Note that *Kq1* is the converse of *Iq5* and *Iq8*. As for *Kq2*, it is possible that all threads are at line 16, but then $\texttt{count} < N$ and some threads are not in the set `guests`. Now for the proof of deadlock freedom.

Theorem 2 *Deadlock is not reachable in the FAI barrier.*

Proof. Assume that deadlock has been reached. Then every thread q is blocked at line 16 with $nef_q \neq \texttt{sense}$. The invariant *Kq1* then implies that all threads are in the set `guests`. This implies that $\#\texttt{guests} = N$. On the other hand, *Iq1*, *Iq4*, and *Kq2* imply that $\texttt{butler} = N$ and $count = \#\texttt{guests}$ and $\texttt{count} \neq N$. This gives a contradiction. □

2.3 The ring barrier

In the ring barrier, the threads are arranged in a directed ring. Every thread waits twice in the barrier. Thread p waits only for the value of the boolean $\texttt{tog}[p]$. Initially, $\texttt{tog}[p] = \mathit{false}$ for all p. The ring barrier is almost symmetric, but thread 0 has a special role: it negates the Boolean value that is sent forward to the next thread.

```
Barrier(thread p):
   int next = (p+1 < N ? p+1 : 0) ;
   bool nz = (p > 0) ; // nz: nonzero
   await (tog[p] = nz) ;
   tog[next] := true ; cnt_p++ ;
   await (tog[p] != nz) ;
   tog[next] := false
end Barrier
```

The algorithm is like a token ring, in which a message is sent around the ring twice, the first time as a token, the second time as an acknowledgement. For the sake of the proof, we therefore introduce a shared ghost variable `loc` to hold the location and the meaning of the message. The message is the token iff `loc` $< N$, the location of the message is `loc` **mod** N. Initially `loc` $= 0$. It is updated in the atomic commands that modify `tog`. The incrementation of cnt_p is combined atomically with the first modification of `tog`. In this way, we arrive at the following transition system, where we regard the constant local variables $\mathtt{nz}_p$ and $\mathtt{next}_p$ as abbreviations.

```
Transition system
 loop of thread p:
11   NCS ;
12   await (tog[p] = nz_p) ;
13   tog[next_p] := true ;
     cnt_p++ ; loc++ ;
14   await (tog[p] != nz_p) ;
15   tog[next_p] := false ;
     loc := (loc+1 < 2*N ? loc+1 : 0) ;
 endloop
```

In line 13, the incrementations can be atomically combined with the assignment to `tog` because cnt_p and `loc` are ghost variables. The same holds for line 15.

The initial condition is

$$\mathtt{loc} = 0 \ \wedge\ \forall\, q : pc_q = 11 \ \wedge\ \mathtt{tog}[q] = \mathit{false} \ \wedge\ cnt_q = 0 \ .$$

The relationship between the algorithm and the variable `loc` is captured in the four invariants

Iq1 : $\quad \mathtt{tog}[q] \ = \ (nz_p = (q \le \mathtt{loc} < q + N))$,
Iq2 : $\quad q \in [14, 15] \ \equiv\ (q < \mathtt{loc} \le q + N)$,
Iq3 : $\quad q \in [13] \ \Rightarrow\ q = \mathtt{loc}$,
Iq4 : $\quad q \in [15] \ \Rightarrow\ q + N = \mathtt{loc}$.

For *Iq1*, note that the equality operator (=) for booleans is associative (as emphasized by Dijkstra), i.e, $a = (b = c)$ is the same as $(a = b) = c)$ for booleans a, b, c.

Predicate *Iq1* is threatened only by the steps 13 and 15. It has the respective remedies *Iq3* and *Iq4*. The same holds for the predicate *Iq2*. Predicate *Iq3* is threatened only by the steps 12, 13, and 15. At step 12, it has the remedies *Iq1* and *Iq2*. At 13 and 15, it has the remedies *Iq3* and *Iq4* as before. Predicate *Iq4* is threatened only by the steps 13, 14, and 15. At step 14, it has the remedies *Iq1* and *Iq2*. At 13 and 15, it has the remedies *Iq3* and *Iq4* as before. This concludes the proof of the invariants *Iq1* up to *Iq4*.

An invariant about *cnt* is needed to prove the barrier condition *BC*. There is one thread that holds the least value of *cnt*, and other threads may have the same value. This is postulated in the invariant

Iq5 : $\quad cnt_q = cnt_{N-1} + (q < \mathtt{loc} < N \ ?\ 1 : 0)$.

It follows from *Iq2* and *Iq5* that

$$q \in [11, 13] \ \Rightarrow\ cnt_q = cnt_{N-1} \ .$$

Therefore, again using *Iq5*, the barrier condition is implied.

Predicate *Iq5* is threatened only by the steps 13 and 15. At these steps it has the respective remedies *Iq3* and *Iq4*.

Absence of deadlock means that the message can always be sent forward to the next thread in the ring. The proof needs the additional invariant

Iq6 : $\quad \mathtt{loc} < 2 \cdot N$,

which is proved by means of the remedy *Iq3*.

Theorem 3 *In the ring barrier, deadlock is not reachable.*

Proof. Assume the state is in deadlock. Then all threads are blocked, waiting in the lines 12 or 14. If $\texttt{loc} < N$, the invariants *Iq2* and *Iq1* imply that thread $q = \texttt{loc}$ is not at line 14, and hence at line 12, and that $\texttt{tog}[q] = nz_q$ holds, so that q is enabled. On the other hand, if $N \leq \texttt{loc} < 2 \cdot N$, the same invariants imply that thread $q = \texttt{loc} - N$ is not at line 12, and hence at line 14, and that $\texttt{tog}[q] \neq nz_q$ holds, so that q is enabled. □

2.4 A new general tree barrier

The tree barriers of, e.g., [2, 5] work with a prescribed tree. Here, one can use an arbitrary rooted tree with a one-to-one correspondence between the nodes and the threads. One thread is the root of the tree, and every thread q is a node and therefore has a set of children *children*(q). The algorithm uses a shared array `aa` of booleans, which are toggled twice in every call of the barrier. The boolean `aa`[`root`] of the root is ignored. Every shared variable `aa`[q], $q \neq \texttt{root}$, is read and written only by thread q and its parent in the tree.

```
bool aa[N] := ([N] false) ;

barrier(thread p):
(*)  for all children kk of p do
        await (aa[kk]) endfor ;
     if p != root then
        aa[p] := true ;
        await (not aa[p]) ;
     endif ;
(**) for all children kk of p do
        aa[kk] := false endfor ;
```

The barrier works as follows. Each node $q \neq \texttt{root}$ set its value $\texttt{aa}[q] := \textit{true}$, when its children have done so. As the leaves of the tree have no children, the process begins at the leaves, and ends with the root. When the root observes that all its descendants have set `aa`, it spreads the message *false* over the descendants. Upon reception of *false*, the descendants can proceed. Note that the same tree must be used for collection and for the backward broadcast.

Four special cases are considered here.

Case A, the flat tree. All nonroot threads are leaves of the tree and children of the root. Loop (*) can therefore be implemented by

```
(A)  if p = root then
        for all threads kk != root do
           await (aa[kk]) endfor
     endif
```

and similarly for loop (**)

Case B, the linear tree. Let the threads be $0, \ldots, N-1$. The root is 0, thread $N-1$ is the only leaf, every thread $q < N-1$ has single child $q+1$. Loop (*) is implemented by

```
(B)  if p < N-1 then
        await (aa[p+1] != aa[p])
     endif
```

It is not likely that this tree performs well.

Case C, the binary tree. Again $0, \ldots, N-1$ are the threads. Every thread q has at most two children, viz. $2*q+1$ and $2*q+2$, provided these are less than N. Loop (*) is implemented by

```
(C)  if 2*p + 2 < N then
        await (aa[2*p+2] != aa[p]) endif ;
     if 2*p + 1 < N then
        await (aa[2*p+1] != aa[p]) endif ;
```

If the children of a node in the tree have different heights, the order of inspection in the loop over the children matters for performance. It is best to begin with the children with the smallest height, because they are likely to be the first to toggle `aa`. For this reason, in loop C the higher node is inspected first This suggests that it is not necessarily optimal to use a balanced tree.

Case D. Indeed, the (unbalanced) tree of [2] may be quite adequate. In this case, the **for** loop (*) over the children of p can be described by

```
(D)   int pow := 1, pd := p;
      while pd mod 2 = 0 and p + pow < N do
         await (aa[p+pow] != aa[p]) ;
         pow := 2 * pow ; pd := pd / 2 ;
      endwhile
```

In this tree, the parent of a nonzero node q is determined as follows. As $q > 0$, there is a highest power 2^d that divides q. Then thread $q - 2^d$ is the parent of q.

2.4.1 Correctness of the tree barrier

The greatest problem for correctness is that some threads can remain waiting for $\neg\,$`aa`$[p]$ while other threads have executed loop (**) and loop (*), and have reached the waiting position for $\neg\,$`aa`$[p]$ again. To distinguish the two cases, a shared set-valued ghost variable `rear` is introduced, which is set by the root when it passes the conditional statement. For simplicity, the threads are represented by numbers 0 up to $N-1$ in such a way that 0 is the root, and that $q < r$ holds if q is the parent of r.

The transition system for the tree barrier

```
  loop of thread p:
11   NCS ;
12   lis_p := children(p) ;
13   while exists kk_p in lis_p do
14      await (aa[kk_p]) ;
        remove kk_p from lis_p
     endwhile ;
15   if p != 0 then aa[p] := true
     else rear := allthreads endif ;
     cnt_p ++ ;
16   await (p = 0 or not aa[p]) ;
17   lis_p := children(p) ;
18   while exists k in lis_p do
        aa[k] := false ; remove k from lis_p
     endwhile ;
     remove p from rear
  end loop .
```

Note that the loop variables of the two loops are treated in a different way. In the loop at line 18, the variable k need not be recorded in the state because one atomic statement can contain the choice of k, the assignment `aa`$[k] := \mathit{false}$, and the removal of k from lis_p. In the first loop, however, thread p has to remember the chosen value kk_p while it waits at location 14.

The initial condition is

$$\mathit{Init}: \qquad \texttt{rear} = \emptyset \ \wedge\ \forall\, q : pc_q = 11 \ \wedge\ \texttt{aa}[q] = \mathit{false} \ \wedge\ \mathit{cnt}_q = 0\ .$$

The discussion of the ghost variables `rear` and cnt_p is postponed. First some invariants concerning array `aa` and the tree structure.

For nonroot nodes q, the setting of $\mathtt{aa}[q]$ in line 15 is followed by testing $\neg\,\mathtt{aa}[q[$ in line 16. We therefore have the invariant

$$Iq1: \qquad q \neq 0 \ \wedge\ \mathtt{aa}[q] \ \Rightarrow\ q \in [16]\ .$$

Indeed, predicate *Iq1* is inductive.

There are two predicates about `aa` related to the tree:

$$Iq2: \qquad q \neq 0 \ \wedge\ \mathtt{aa}[q] \ \wedge\ r \in children(q) \ \Rightarrow\ \mathtt{aa}[r]\ ,$$
$$Iq3: \qquad q \in [15] \ \wedge\ r \in children(q) \ \Rightarrow\ \mathtt{aa}[r]\ .$$

Predicate *Iq2* is threatened only by the steps 15 and 18. It has the remedy *Iq3* at line 15. At line 18, it has the remedies *Iq1* and

$$Iq4: \qquad q \in [13, 18] \ \Rightarrow\ lis_q \subseteq children(q)\ .$$

Predicate *Iq3* is threatened only by the steps 13 and 18. It has the remedy *Iq4* at step 18. At step 13, it has the remedy

$$Iq5: \qquad q \in [13, 14] \ \wedge\ r \in children(q) \ \Rightarrow\ r \in lis_q \ \vee\ \mathtt{aa}[r]\ .$$

Predicate *Iq4* is inductive. Predicate *Iq5* is threatened only by step 18. It has the remedy *Iq4*. The invariants *Iq1* up to *Iq5* enable us the first global conclusion: if the root is at line 15, all other threads q have $\mathtt{aa}[q]$, and are thus at line 16 because of *Iq1*.

$$Crit1: \qquad 0 \in [15] \ \wedge\ q \neq 0 \ \Rightarrow\ \mathtt{aa}[q] \ \wedge\ q \in [16]\ .$$

This is proved as follows. Assume that $0 \in [15]$. Let q be the least number of a thread $q \neq 0$ with $\neg\,\mathtt{aa}[q]$ (if such exist). As $q \neq 0$, it has a parent p with $p < q$. Minimality of q implies that $p = 0$ or $\mathtt{aa}[p]$. If $p = 0$, predicate *Jq3* implies that $\mathtt{aa}[q]$ holds. Otherwise, *Jq2* does this. In either case, this gives a contradiction. Therefore such threads q do not exist. For the second conjunct, use *Iq1*.

The next invariant describes the relation between `rear`, the location of the thread and the value of `aa`: every thread q is always in *Bottom*, *Middle*, or *Top*.

$$\begin{array}{lll} Iq6: & q \in Bottom \ \vee\ q \in Middle \ \vee\ q \in Top\ , \text{ where} & \\ & \quad q \in Bottom \ \equiv\ q \neq 0 \ \wedge\ q \in \mathtt{rear} \ \wedge\ \mathtt{aa}[q] \ \wedge\ q \in [16]\ , & \\ & \quad q \in Middle \ \equiv\ q \in [16, 18] \ \wedge\ q \in \mathtt{rear} \ \wedge\ (q = 0 \ \vee\ \neg\,\mathtt{aa}[q])\ , & \\ & \quad q \in Top \ \equiv\ q \in [11, 16] \ \wedge\ q \notin \mathtt{rear} \ \wedge\ (q \in [16] \ \Rightarrow\ q \neq 0 \ \wedge\ \mathtt{aa}[q])\ . & \end{array}$$

A case distinction is used to prove this invariant. First, if $q \in Bottom$, any step of the algorithm keeps $q \in Bottom$ or transfers q to *Middle*. Similarly, if $q \in Middle$, any step of the algorithm keeps $q \in Middle$ or transfers q to *Top*. If q is in *Top*, any step of the algorithm, except step 15 of the root, keeps q in *Top*. In the case of step 18, this is proved with the new invariant

$$Iq7: \qquad q \in [18] \ \wedge\ r \in lis_q \ \Rightarrow\ r \in Bottom\ .$$

If $q \in Top$, step 15 of root 0 transfers q to *Bottom* or *Middle* because of *Crit1*.

Before proceeding to prove *Iq7*, we first note that predicate *Iq6* has the following easy consequences:

$$Iq6A: \qquad q \in [17, 18] \ \Rightarrow\ q \in \mathtt{rear}\ ,$$
$$Iq6B: \qquad q \in \mathtt{rear} \ \Rightarrow\ q \in [16, 18]\ ,$$
$$Iq6C: \qquad q \neq 0 \ \wedge\ q \in [16] \ \Rightarrow\ q \in \mathtt{rear} \ \vee\ \mathtt{aa}[q]\ .$$

Predicate *Iq7* is threatened only by the steps 17 and 18. At step 18, it has the remedy *Iq4*. At step 17, it has the remedies *Iq6A* and

$$Iq8: \qquad q \in [16, 17] \ \wedge\ q \in \mathtt{rear} \ \wedge\ r \in children(q) \ \Rightarrow\ r \in Bottom\ .$$

Predicate *Iq8* is threatened only by the steps 15 and 18. At step 15, it has the remedies *Iq6B* and *Crit1*. At step 18, it has the remedy *Iq4*. This concludes the formation and proof of the first batch of invariants *Iq**.

For the proof of the barrier condition, we postulate

$$Jq1: \qquad cnt_q = cnt_0 + (q \in [16] \ \wedge \ q \notin \texttt{rear}\ ?\ 1 : 0)\ .$$

Indeed, this predicate clearly implies the barrier condition

$$BC: \qquad q \in [11] \ \Rightarrow \ cnt_q \leq cnt_r\ .$$

Predicate *Jq1* is threatened only by the steps 15 and 16. At step 16, it has the remedy *Iq6C*. At step 15, it has the remedies *Crit1*, *Iq6B*, and

$$Crit2: \qquad 0 \in [15] \ \Rightarrow \ q \notin \texttt{rear}\ .$$

In order to prove *Crit2*, one postulates the invariant

$$Jq2: \qquad r \in children(q) \ \wedge \ \texttt{aa}[r] \ \wedge \ r \in \texttt{rear} \ \Rightarrow \ q \in \texttt{rear}\ .$$

Indeed, using *Jq2*, predicate *Crit2* can be proved as follows. Assume $0 \in [15]$. If there is a thread in `rear`, there is a smallest one, say r. As $0 \in [15]$, predicate *Iq6B* implies that $r \neq 0$. Predicate *Crit1* implies that $\texttt{aa}[r]$ holds. Let q be the parent of r. Then *Jq2* implies $q \in \texttt{rear}$. On the other hand, $q < r$ because q is the parent of r. This contradicts the minimality of r.

Predicate *Jq2* is threatened only by the steps 15 and 18. At step 15, it has the remedy *Iq6B*. At step 18, it has the remedy

$$Jq3: \qquad q \in [18] \ \wedge \ r \in children(q) \ \wedge \ \texttt{aa}[r] \ \wedge \ r \in \texttt{rear} \ \Rightarrow \ r \in lis_q\ .$$

Predicate *Jq3* is threatened only by step 15. It has the remedies *Crit1* and *Iq6B*. This concludes the proof of the barrier condition *BC*.

It remains to prove absence of deadlock. This is rather easy. Assume that all threads are blocked. Then they are all waiting at one of the lines 14 and 16. Thread 0, the root, need not wait at line 16 and is therefore waiting at line 14. Let q be the thread with the greatest number that is waiting at line 14. As q is blocked at 14, it has $\neg\,\texttt{aa}[r]$ for thread $r = \texttt{kk}_q$. Now, r is a child of q and therefore $q < r$. By maximality of q, thread r is not waiting at line 14. Therefore r is waiting at line 16 and $\texttt{aa}[r]$ holds. This is a contradiction. This proves

Theorem 4 *In the tree barrier, deadlock is not reachable.*

References

[1] G.R. Andrews. *Foundations of multithreaded, parallel, and distributed Programming.* Addison Wesley, Reading, etc., 2000.

[2] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17:1–17, 1988.

[3] L. Lamport. Implementing dataflow with threads. *Distributed Computing*, 21:163–181, 2008. See also Lamport's Writings (nr. 164).

[4] L. Lamport. My writings.
http://research.microsoft.com/en-us/um/people/ lamport/pubs/pubs.html, 2010.

[5] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.