# Distributed Generalized Dynamic Barrier Synchronization

**Conference Paper** · January 2011

DOI: 10.1007/978-3-642-17679-1_13 · Source: DBLP

CITATIONS
2

READS
104

**3 authors:**

Shivali Agarwal
IBM
**10** PUBLICATIONS **145** CITATIONS
SEE PROFILE

Saurabh Joshi
Indian Institute of Technology Hyderabad
**27** PUBLICATIONS **185** CITATIONS
SEE PROFILE

R. K. Shyamasundar
Indian Institute of Technology Bombay
**291** PUBLICATIONS **1,587** CITATIONS
SEE PROFILE

# Distributed Generalized Dynamic Barrier Synchronization

Shivali Agarwal Saurabh Joshi R.K. Shyamasundar

**Abstract**

Barrier synchronization is widely used in shared-memory parallel programs to synchronize between phases of data-parallel algorithms. With proliferation of many-core processors, barrier synchronization has been adapted for higher level language abstractions in new languages such as X10 wherein the processes participating in barrier synchronization are not known a priori, and the processes in distinct "places" don't share memory. Thus, the challenge here is to not only achieve barrier synchronization in a distributed setting without any centralized controller, but also to deal with dynamic nature of such a synchronization as processes are free to join and drop out at any synchronization phase. In this paper, we describe a solution for the generalized distributed barrier synchronization wherein processes can dynamically join or drop out of barrier synchronization; that is, participating processes are not known a priori. Using the policy of permitting a process to join only in the beginning of each phase, we arrive at a solution that ensures (i) Progress: a process executing phase k will enter phase k + 1 unless it wants to drop out of synchronization (assuming the phase execution of the processes terminate), and (ii) Starvation Freedom: a new process that wants to join a phase synchronization group that has already started, does so in a finite number of phases. The correctness of the solution is formally established. From the perspective of a global observer, our protocol guarantees a bound of at most two phases from the phase a process had registered it's intention to join. We show how the testing by each of the processes with all the other processes can be short circuited leading to efficient synchronization. The above protocol is further generalized to multiple groups of processes (possibly non-disjoint) engaged in barrier synchronization.

## 1 Introduction

Synchronization and coordination play an important role in parallel computation. Language constructs for efficient coordination of computation on shared memory multi-processors, and multi-core processors are of growing interest. There are a plethora of language constructs used for realizing mutual exclusion, point-to-point synchronization, termination detection, collective *barrier* synchronization etc. Barrier [8] is one of the important busy-wait primitives used to ensure that none of the processes proceed beyond a particular point in a computation until all have arrived at that point. A software implementation of the barrier using shared variables is also referred to as phase synchronization [1, 7]. The issues of remote references while realizing barriers has been treated exhaustively in the seminal work [3]. Barrier synchronization protocols, either centralized and distributed, have been proposed earlier for the case when processes that have to synchronize are given a priori [7][5][14][6][15]. With the proliferation of many-core processors, barrier synchronization has been adapted for higher level language abstractions in new distributed shared memory based

languages such as X10 [21] wherein the processes participating in barrier synchronization are not known a priori. Some of the recent works that address dynamic number of processes for barrier synchronization are [25][23][26]. More details on existing work on barrier synchronization can be found in section 6. Surprisingly, a distributed solution to the phase synchronization problem in such dynamic environments has not yet been proposed. In this paper, we describe a distributed solution to the problem of barrier synchronization used as an underlying synchronization mechanism for achieving phase synchronization where processes are dynamically created (in the context of nested parallelism). The challenge arises in arriving at the common knowledge of the processes that want to participate in phase synchronization for every phase in a de-centralized manner such that there are some guarantees on the progress and starvation freedom properties of the processes in addition to the basic correctness property.

### 1.1 Phase Synchronization Problem

The problem of phase synchronization [7] is described below:

Consider a set of asynchronous processes where each process executes a sequence of phases; a process begins its next phase only upon completion of its previous phase (for the moment let us ignore the constitution of a phase). The problem is to design a synchronization scheme which guarantees the following properties:

1. No process begins it's $(k+1)^{th}$ phase until all processes have completed their $k^{th}$ phase, $k \geq 0$.
2. No process will be permanently blocked from executing it's $(k+1)^{th}$ phase if all processes have completed their $k^{th}$ phase, $k \geq 0$.

The set of processes that have to synchronize can be either given a priori which remains unchanged or the set can be a dynamic set in which new processes may join as and when they want to phase synchronize or existing processes may drop out of phase synchronization.

In this paper, we describe a distributed solution for the dynamic barrier synchronization in the context of phase synchronization, wherein processes can dynamically join or drop out of phase synchronization. Using the policy of permitting a process to join in a phase subsequent to the phase of registration, we arrive at a solution that ensures

(i) *Progress*: a process executing phase $k$ will enter phase $k+1$ unless it wants to drop out of synchronization (assuming the phase execution of the processes terminate), and

(ii) *Starvation Freedom*: a new process that wants to join a phase synchronization group that has already started, does so in a finite number of phases. Our protocol establishes a bound of at most *two* phases from the phase it registered it's intention to join[1] the phase synchronization. The lower bound is *one* phase.

The correctness of the solution is formally established. The dynamic barrier synchronization algorithm is further generalized to cater to groups of barrier synchronization processes.

[1]Starvation Freedom is guaranteed only for processes that are registered

$< Program >$ ::=$< async\ Proc > || < async\ Proc >$
$< Proc >$ ::=$< clockDec >; < stmtseq > | clocked < clock\text{-}id >< stmtseq >$
$< clockDec >$ ::= $new\ clock\ c1, c2..$
$< stmtseq >$ ::=$< basic\text{-}stmt > | < basic\text{-}stmt >< stmtseq >$
$< basic\text{-}stmt >$ ::= $async\ Proc|atomic\ stmt|seq\ stmt|c.register|c.drop|next$
$clock\text{-}id$ ::= $c1, c2, ...$

Figure 1: Abstract Clock Language for Barrier Synchronization

# 2 Barrier synchronization with dynamic set of processes

We consider a distributed system which gets initialized with a non-empty set of processes. New processes can join the system at will and existing processes may drop out of the system when they are done with their work. They carry out individual computations in phases and synchronize with each other at the end of each phase. Since it is a distributed system with no centralized control and no a priori knowledge of number of processes in the system, each process has to dynamically discover the new processes that have joined the system in such a manner that the new process can start synchronizing with them in finite amount of time. The distributed barrier synchronization protocol described below deals with this issue of including new processes in the ongoing phase synchronization in a manner that progress of existing as well as newly joined processes is ensured. It also handles the processes that drop out of the system so that existing processes know that they do not have to wait on these for commencing the next phase.

Note that there is no a priori limit on the number of processes. The abstract linguistic constructs for registration and synchronization of processes is described in the following.

## 2.1 Abstract Language

We base our abstract language for the barrier synchronization protocol on X10. The relevant syntax is shown in figure 1 and explained below:

• *Asynchronous activities*: The keyword to denote asynchronous processes is *async*. The async is used with an optional place expression and a mandatory code block. A process that creates another process is said to be the parent of the process it creates.

• *Clock synchronization*: Special variables of type *clock* are used for barrier synchronization of processes. A *clock* corresponds to the notion of a barrier. A set of processes registered with a clock synchronize with each other w.r.t. that clock. A barrier synchronization point in a process is denoted by *next*. If a process is registered on multiple clocks, then *next* denotes synchronization on all of those. This makes the barrier synchronization deadlock-free. The abstraction of phase synchronization through clocks enables to form groups of processes such that groups can merge or disjoin dynamically for synchronization. Some important points regarding dynamic joining rule for phase synchronization are:

- A process registered on clock $c$, can create a child process synchronizing on $c$ via *async clocked*

$c$ {*body*}. The child process joins the phase synchronization in phase $k+1$, if the parent is in phase $k$ while executing *async*.

- A process can register with a clock $c$ using $c.register$. It will join the phase synchronization from phase $k+1$ or $k+2$ if the clock is in phase $k$ at the time of registration.

Some important points regarding dropping out of phase synchronization are:
- A process that drops in phase $k$ is dropped out in the same phase and is not allowed to create child processes that want to join phase synchronization in that phase. Note that this does not restrict the expressiveness of the language in any way and ensures a clean way of dropping out.
- A process that registers after dropping loses the information of parent and is treated as a process whose parent is not clocked on $c$.
- An implicit $c.drop$ is assumed when a process registered on clock $c$ terminates.

We now provide a solution in the form of a protocol for distributed dynamic barrier synchronization problem that provably obeys the above mentioned dynamic joining rules.

# 3 Distributed barrier synchronization solution

The distributed barrier synchronization protocol for a single clock is given in figure 2. The figures respectively describe the protocol for barrier operations like initialization, synchronization and drop. The notations used in the solution are described below.
**Notation**:
- We denote $i^{th}$ process by $A_i$ (also referred as process $i$).
- Phases are tracked in terms of $k-1, k, k+1, \cdots$.
- We use the guarded command notation [2] for describing our algorithm as it is easy to capture interleaving execution and termination in a structured manner. The main commands are explained in appendix in brief.

**Assumption**: The processes do not have random failures and will always call $c.drop$ if they want to leave the phase synchronization.

## 3.1 Correspondance between protocol steps and clock operations

The correspondence of the clock operations with the protocol operations is given below:

- **new clock** $c$: Creation of a clock(barrier) corresponds to creation of a special process $A_c$ that executes as follows where the code blocks `INIT_c`, `SYNC` and `CONSOLIDATE` are shown in figure 2.

```
INIT_c;
while (true) {
   SYNC;
   CONSOLIDATE;
}
```

*Note on $A_c$*: It is a special process that exists till the program terminates. It acts like a point of contact for processes in case of explicit registration through *c.register* as seen below without introducing any centralized control.

- **next**: A process $A_i$ already in phase synchronization performs a `next` for barrier synchronization. A *next* corresponds to:

```
SYNC;  CONSOLIDATE;
```

- **Registration through *clocked***: A process $A_i$ can either register through *clocked* at time of creation in which case it gets into the list $A_j.registered$ of its parent process, $A_j$. In this case, $A_i$ joins from next phase. The specific code that gets executed in the parent for a clocked process is:

```
INIT_i;
A_j.registered:=A_j.registered+A_i;
```

  The code that gets executed in $A_i$ is:

```
while (!A_i.proceed);
```

- **Registration through *c.register***: If $A_i$ registers like this, then it may join phase synchronization within atmost next *two* phases. Following code gets executed in $A_i$:

```
INIT_i;
A_c.registered:=A_c.registered+A_i;
while (!A_i.proceed);
```

- **c.drop** : Process $A_i$ drops out of phase synchronization through *c.drop* (see DROP in fig. 2). The code that gets executed is:

```
DROP;
```

*Note*: 1) Though we assume $A_c$ to exist throughout the program execution, our algorithm is robust with respect to graceful termination of $A_c$, that is, it terminates after completing `CONSOLIDATE` and there are no processes in $A_c.registered$ upon consolidation. The only impact on phase synchronization being that no new processes can register through *c.register*. 2) The assignments are done atomically.

### 3.2 How the Protocol Works

The solution achieves phase synchronization by ensuring that the set of processes that enter a phase is a common knowledge to all the processes. Attaining common knowledge of the existence of new processes and the non-existence of dropped processes in every phase is the non-trivial part of the phase synchronization protocol in a dynamic environment. The machinery built to solve this problem is shown in figures 2 and described below in detail.

**Protocol for Process** $i$

`[`$A_c$`]::` **(* Initialization of clock process *)**

`INIT_c:` $A_c.executing,\ A_c.next, A_c.Iconcurrent,\ A_c.registered,\ A_c.proceed,\ A_c.drop$
$:=\ 0, 1, A_c, \emptyset, true,\ false;$

`[`$A_i$`]::` **(* Initialization of $i$ that performs a registration *)**

`INIT_i:` $A_i.proceed\ :=\ false;$

`SYNC :` **(*CHECK COMPLETION of $A_i.executing$ by other members*)**

$A_i.newsynchproc := A_i.registered;$
$A_i.newIconcurrent :=\ A_i.newsynchproc + A_i;$
$A_i.next := A_i.next + 1;\ A_i.checklist := \emptyset;$
`do`
$A_i.Iconcurrent\ \neq\ \emptyset\ \wedge\ A_j\ \in\ A_i.Iconcurrent \wedge\ i\ \neq\ j\ \wedge$
$A_i.next\ \leq\ \ A_j.next \rightarrow$
$A_i.Iconcurrent\ :=\ A_i.Iconcurrent\ -\ \{A_j\};$
$A_i.newIconcurrent\ :=\ A_i.newIconcurrent + A_j.newsynchproc\ +;$
$\{A_j\};\ A_i.checklist :=\ A_i.checklist\ +\ A_j$
$[\!]\ A_i.Iconcurrent\ \neq\ \emptyset\ \wedge\ A_j.drop$
$\rightarrow\ A_i.Iconcurrent\ :=\ A_i.Iconcurrent\ -\ \{A_j\};$
$[\!]\ A_i.Iconcurrent\ \neq\ \emptyset\ \wedge\ A_j\ \in\ A_i.Iconcurrent\ \wedge$
$A_i.next >\ A_j.next$ `(* no need to check` $i\ \neq\ j$ `*)` $\rightarrow\ skip;$
`od;`
$A_i.executing\ :=\ A_i.executing + 1;$ `(* Set the current phase *)`
`do (* Check for completion of phase in other processes *)`
$A_i.checklist \neq \emptyset\ \wedge\ j \in\ A_i.checklist\ \wedge$
$A_i.executing == A_j.executing$
$\rightarrow\ A_i.checklist := A_i.checklist\ -\ \{A_j\}$
`od;`

`CONSOLIDATE:` **(* CONSOLIDATE processes for the next phase *)**

$A_i.Iconcurrent$ `:=` $A_i.newIconcurrent;$
$A_i.registered\ :=\ A_i.registered\ -\ \ A_i.newsynchproc;$
`for all` $j\ \in\ A_i.newsynchproc$ `do`
$A_j.executing, A_j.next, A_j.Iconcurrent,\ A_j.registered,\ A_j.drop$
$:= A_i.executing, A_i.next, A_i.Iconcurrent, \emptyset,\ false;$
$A_j.proceed\ :=\ true;$

`DROP :` **(* Code when Process $A_i$ calls c.drop*)**

$A_i.proceed\ :=\ false;$
$A_i.drop\ :=\ true;$

Figure 2: Action of processes in phase synchronization

Protocol Variables: $A_i.Iconcurrent$ denotes the set of processes that $A_i$ is synchronizing with in a phase. This set may shrink or expand after each phase depending on if an existing process drops or a new process joins respectively. The variable $A_i.newsynchproc$ is assigned to the set that process $A_i$ wants the other processes to include for synchronization from next phase onwards. The variable $A_i.newIconcurrent$ is used to accumulate the processes that will form $A_i.Iconcurrent$ in next phase. $A_i.executing$ denotes the current phase of $A_i$ and $A_i.next$ denotes the next phase that the process will move to.

`INIT_c`: This initializes the special clock process $A_c$ that is started at the creation of a clock $c$. Note that the clock process is initialized to itself for the set of initial processes that it has to synchronize with. $A_c.proceed$ is set to $true$ to start the synchronization.

`INIT_i`: When a process registers with a clock, $A_i.proceed$ is set to $false$. The newly registered process waits for $A_i.proceed$ to be made $true$ which is done in `CONSOLIDATE` block of the process that contains $A_i$ in its $registered$ set. Rest of the variables are also set properly in this `CONSOLIDATE` block. In the following, we explain the protocol for `SYNC` and `CONSOLIDATE`.

`SYNC`: This is the barrier synchronization stage of a process and performs the following main functions: 1) checks if all the processes in the phase are ready to move to next phase; $A_i.next$ is used to denote the completion of phase and check for others in the phase, 2) informs the other processes about the new processes that have to join from next phase, 3) establishes if the processes have exchanged the relevant information so that it can consolidate the information required for the execution of next phase.

The new processes that are registered with $A_i$ form the set $A_i.newsynchproc$. This step is required to capture the local snapshot. Note that for processes other than clock process, $A_i.registered$ will be same as $A_i.newsynhproc$. However for the special clock process $A_c$, $A_c.registered$ may keep on changing during the `SYNC` execution. Therefore, we need to take a snapshot so that consistent set of processes that have to be included from the next phase can be conveyed to other processes that are present in the synhcronization.

The increment of $A_i.next$ denotes that effectively the process has completed the phase and is preparing to move to the next phase. Note that after this operation the difference between $A_i.next$ and $A_i.executing$ becomes 2 denoting the transition. The second part of `SYNC` is a do-od loop that forms the crux of barrier synchronization. There are $three$ guarded commands in this loop which are explained below.

1. The first guarded command checks if there exists a process $j$ in $A_i.Iconcurrent$ that has also reached barrier synchronization. If the value of $A_j.next$ is greater or equal to $A_i.next$, then it implies that $A_j$ has also reached the barrier point. If this guard is evaluated true, then that process is removed from $A_i.Iconcurrent$ and the new processes that registered with $A_j$ are added to the set $A_i.newIconcurrent$.

2. The second guard checks if any process in $A_i.Iconcurrent$ has dropped out of synchronization and accordingly the set $A_i.newIconcurrent$ is updated.

3. The third guard is true if the process $j$ has not yet reached the barrier synchronization point. The associated statement with this guard is a no-op. It is this statement which forms the waiting part for barrier synchronization.

By the end of this loop, $A_i.Iconcurrent$ shall only contain $A_i$. The current phase denoted by $A_i.executing$ is incremented to denote that process can start with the next phase. However, to ensure that the local snapshot captured in $A_i.newsynchproc$ is properly conveyed to the other processes participating in phase synchronization, another do-od loop is executed that checks if processes have indeed moved to next phase by incrementing $A_i.executing$.

`CONSOLIDATE`: After ensuring that $A_i$ has synchronized on the barrier, a final round of consolidation is performed to prepare $A_i$ for executing in next phase. This phase consolidation is described under label `CONSOLIDATE`. The set of processes that $A_i$ needs to phase synchronize are in $A_i.newIconcurrent$, therefore, $A_i.Iconcurrent$ is assigned to $A_i.newIconcurrent$. All the new processes that will join from $A_i.executing$ are signalled to proceed after initializing them properly. The set $A_i.registered$ is updated to ensure that it has only those new processes that got registered after the value of $A_i.registered$ was last read in `SYNC`. This is possible because of the explicit registration that is allowed through the special clock process.

`DROP`: $A_i.drop$ is set to $true$ so that the guarded command in `SYNC` can become true appropriately. The restriction posed on a drop command ensures that $A_i.registered$ will be empty and thus the starvation freedom guarantee is preserved.

An illustrative description of the protocol is provided in the appendix.

# 4 Correctness of the Solution

In this section, we provide a proof in semi-formal way in the style of [1]. The proof obligations for synchronization and progress are given below. The symbol '$\mapsto$' denotes *leads to*.

- Synchronization

  We need to show that the postcondition of `SYNC;CONSOLIDATE;` (corresponding to barrier synchronization) for processes that have *proceed* set to *true* is :
  $\{\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow A_i.executing = A_j.executing)\}$

- Progress
  Property 1: The progress for processes already in phase synchronization is given by ($k$ is used to denote current phase) the following property which says that if all the processes have completed phase $k$, then each of the processes move to a phase greater than k if they do not drop out.
  P1: $\{\forall i(A_i.drop = false \wedge \forall j(A_j.drop = false \Rightarrow A_j.executing \geq k) \mapsto (A_i.executing \geq k + 1))\}$
  Property 2: The progress for new processes that want to join the phase synchronization is given by the following property which says that a process that gets registered with a process involved in phase synchronization will also join the phase synchronization.
  P2: $\{\exists i((A_i.proceed = false \wedge \exists j(i \in A_j.registered)) \mapsto A_i.proceed = true)\}$

The proof details can be found in Appendix C.

**Complexity Analysis**: The protocol in it's simplest form has a remote message complexity of $O(n^2)$ where $n$ is the upper bound on the number of processes that can participate in the barrier

synchronization in any phase. This bound can be improved in practice by optimizing the testing of the completion of a phase by each of the participating processes. The optimization is briefly explained in the following. When a process $A_i$ checks for completion of phase in other process, say $A_j$, and it finds that $A_i.executing < A_j.executing$, then it can actually come out of the do-od loop by copying $A_j.newIconcurrent$ that has the complete information about the processes participating in next phase. This optimization can have the best case of $O(n)$ messages and is very straightforward to embed in the proposed protocol. Note that in any case, atleast $n$ messages are always required to propagate the information.

# 5 Generalization: Multi-Clock Phase Synchronization

In this section, we outline the generalization of the distributed dynamic barrier synchronization for multiple clocks.
1) There is a special clock process for each clock.
2) The processes maintain protocol variables for each of the clocks that they register with.
3) A process can register with multiple clocks through *C.register*, where *C* denotes a set of clocks, as the first operation before starting with phase synchronized computation. The notation C.register denotes that for all clocks c such that c∈C, perform c.register. The corresponding code is:

```
for each c in C
     INIT_i_c;
     A_c.registered:=A_c.registered+A_i;
     while (!A_i_c.proceed);
```

Some important restrictions to avoid deadlock scenarios are:
i) C.register, when C contains more than one clock, can only be done by the process that creates the clocks contained in C.
ii) If a process wants to register with a single clock c that is *in use for phase synchronization by other processes*, it will have to drop all it's clocks and then it can use c.register to synchronize on the desired clock. Note that the clock c need not be re-created.
iii) Subsequent child processes should use *clocked* to register with any subset of the multiple clocks that the parent is registered with.
This combined with (iv) below avoids the deadlock scenarios of the likes of mobile barriers [25].
iv) For synchronization, the process increments the value of $A_i.next$ for each registered clock, then executes the guarded loop for each of the clocks before it can move to `CONSOLIDATE` stage. The `SYNC` part of the protocol for multi-clock is very similar to single-clock except for an extra loop to run the guarded command loop for each of the clocks.

A process clocked on multiple clocks results in synchronization of all the processes that are registered with these clocks. This is evident from the second do-od loop in `SYNC` part of the barrier synchronization protocol. For example, if a process $A_1$ is synchronizing on c1, $A_2$ on c1 and c2 and $A_3$ on c2, then $A_1$ and $A_3$ also get synchronized as long as $A_2$ does not drop one or both of the clocks. These clocks can be thus thought of as forming a group. Let the processes that are clocked on multiple clocks and result in a group formation of clocks be referred as pivot processes.

If c1,c2 form a group and c2,c3 another, then if the pivot process is same for the two groups, then transitivity holds as a result of which c1,c2,c3 also forms a group. If the pivot processes are different but the groups have a common clock, then transitivity does not hold and this manifests as a phase differenece of *one*, at the most, between processes synchronizing on non-common clocks of the two groups having a common clock. Note that a group can be broken into subgroups if the pivot process decides to drop out. In case of multiple pivot processes, sub groups are formed when the last pivot process drops out.

Each group can be thought of as an abstract single clock such that a new process that joins any of the constituent clocks effectively joins the group. Thus, the proofs of progress and starvation freedom for a single clock can be directly applied to a group in the multi-clock synchronization. In the following, we state the guarantees of synchronization provided by the protocol:
1) A process that synchronizes on multiple clocks can move to the next phase only when all the processes in the group formed by the clocks have also completed their current phase.
2) Two clock groups that do not have a common pivot process but have a common clock may differ in phase by atmost *one*. Note that it cannot exceed *one* because that would imply improper synchronization between processes clocked on same clock which as has been proved above to be impossible in our protocol.
3) A new process registered with multiple clocks starts in the next phase (from the perspective of a local observer) w.r.t. each of the clocks individually.

## 6 Comparing with other Dynamic Barrier Schemes

The clock syntax resembles that of X10 but differs in the joining policy. An X10 activity that registers with a clock in some phase starts the synchronization from the same phase. The advantage of our dynamic joining policy (starting from next phase) is that when a process starts a phase, it exactly knows the processes that it is synchronizing with in the phase. This makes it simpler to detect the completion of a phase in a distributed set-up. Whether to join in same phase or next phase is more a matter of semantics rather than expressiveness. If there is a centralized manager process to manage phase synchronization, then the semantics of starting a newly registsered activity in same phase is feasible. However, for a distributed phase synchronization protocol with dynamically joining processes, the semantics of starting from next phase is more efficient.

The other clock related works [24], [23] are directed more towards efficient implementations of X10 like clocks rather than dealing with synchronization in a distributed setting. Barriers in JCSP [26] and occam-pi [25] do allow process to dynamically join and resign from barrier synchronization. Because the synchronization is barrier specific in both JCSP ( using `barrier.sync()` ) and occam-pi ( using `SYNC barrier`, it is a burden on the programmer to write a deadlock free program which is not the case here, as the use of *next* achieves synchronization over all registered clocks. JCSP and occam-pi barriers achieve linear time synchronization due to centralized control of barriers which is also possible in the optimized version of our protocol.

Previous work on barrier implementation has focussed on algorithms that work on pre-specified number of processes or processors. The Butterfly barrier algorithm [9], Dissemination algorithm [10][5],

Tournament algorithm [5], [4] are some of the earlier algorithms. Most of them emphasized on how to reduce the number of messages that need to be exchanged in order to know that all the processes have reached the barrier. Some of the more recent works on barrier algorithms in software are described in [6][11][13][18][15],[3].

As contrasted to the literature, our focus has been on developing algorithms for barrier synchronization where processes dynamically join and drop out; thus, processes that can be in a barrier synchronization need not be known a priori.

# 7 Conclusions

In this paper, we have described a solution for distributed dynamic phase synchronization that is shown to satisfy properties of progress and starvation freedom. To our knowledge, this is the first dynamic distributed multi-processor synchronization algorithm wherein we have the established properties of progress, starvation freedom and shown the dependence of the progress on the entry strategies (captured through process registration). A future direction is to consider fault tolerance in the context of distributed barrier synchronization for dynamic number of processes.

# References


[1] K.M. Chandy and J. Misra, Parallel program Design: A Foundation, Reading, Mass., Addison Wesley, 1988.

[2] E. W. Dijkstra, *Guarded commands, non-determinacy and formal derivation of programs*, Communications of the ACM, 18(8), August 1975.

[3] J.M. Mellor-Crummey,and Michael L. Scott, *Algorithms for scalable synchronization on shared -memory multiprocessors*, ACM TOCS 9, 1, 1991, 21-65.

[4] Anja Feldmann, Thomas Gross, David OHallaron and Thomas M. Stricker, *Subset barrier synchronization on a private-memory parallel system*, SPAA, 1992.

[5] Debra Hensgen, Raphael Finkel, and Udi Manbet, *Two algorithms for barrier synchronization*, International Journal of Parallel Programming, 17(1):117, 1988.

[6] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[7] J. Misra, *Phase Synchronization*, Notes on Unity, 12-90, U. Texas at Austin, 1990.

[8] P.Tang and P.C. Yew, *Processor Self scheduling for multiple-nested parallel loops*, Proc. ICPP, pp. 528-535,August, 1986.

[9] Eugene D. Brooks III, *The butterfly barrier*, International Journal of Parallel Programming, 15(4), 1986.

[10] Y. Han and R. Finkel, *An Optimal Scheme for Disseminating Information* , In Proc. of 17th International Conference on Parallel Processing, 1988.

[11] Michael L. Scott and Maged M. Michael, *The Topological Barrier: A Synchronization Abstraction for Regularly-Structured Parallel Applications* TechReport TR605, 1996, Univ. of Rochester.

[12] Rajiv Gupta, *The fuzzy barrier: a mechanism for high speed synchronization of processors*, ACM SIGARCH Computer Architecture News, 17(2), April 1989.

[13] Rajiv Gupta and Charles R. Hill, *A scalable implementation of barrier synchronization using an adaptive combining tree*, International Journal of Parallel Programming, 18(3), 1990.

[14] Mike Livesey, *A Network Model of Barrier Synchronization Algorithms*, International Journal of Parallel Programming, 20(1), February, 1991.

[15] Hong Xu, P.K. McKinley and L.M.Ni, *Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputers*, Proceedings of the 12th International Conference on 9(12) Jun 1992.

[16] Henry G. Dietz1, A. Zaafrani1 and M. T. O'Keefe *Static scheduling for barrier MIMD architectures*, The Journal of Supercomputing, 5(4), February, 1992.

[17] Shisheng Shang and Kai Hwang, *Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters*, IEEE Transactions on Parallel and Distributed Systems, 6(6), June 1995.

[18] Jenq-Shyan Yang and Chung-Ta King, *Designing Tree-Based Barrier Synchronization on 2D Mesh Networks*, IEEE Transactions on Parallel and Distributed Systems, 9(6), 1998.

[19] William E. Cohen, David W. Hyde and Rhonda K. Gaede, *An Optical Bus-Based Distributed Dynamic Barrier Mechanism*, IEEE Transactions on Computers, 49(12), December 2000.

[20] T.A. Johnson, R. R. Hoare, *Cyclical cascade chains: A dynamic barrier synchronization mechanism for multiprocessor systems*, 15th International Parallel and Distributed Processing Symposium (IPDPS'01) Workshops, 2001

[21] Vijay Sarawat and Radha Jagadeesan, *Concurrent clustered programming*, CONCUR 2005: Concurrency Theory.

[22] *Unified Parallel C Language*, http://www.gwu.edu/ upc/

[23] J. Shirako, M.D. Peixotto, V. Sarkar and Scherer, N. William, *Phasers: a unified deadlock-free construct for collective and point-to-point synchronization*, ICS, 2008, pp. 277-288.

[24] N. Vasudevan, O. Tardieu, J. Dolby, S.A. Edwards, *Compile-Time Analysis and Specialization of Clocks in Concurrent Programs*, CC, 2009, pp. 48-62

[25] P. Welsch, F. Barnes, *Mobile Barriers for occam-pi: Semantics, Implementation and Application*, Communicating Process Architecture, 2005.

[26] P. Welsch, N. Brown, J. Moores, K. Chalmers, B.H.C. Sputh, *Integrating and Extending JCSP*, CPA, 2007, pp. 349-370.

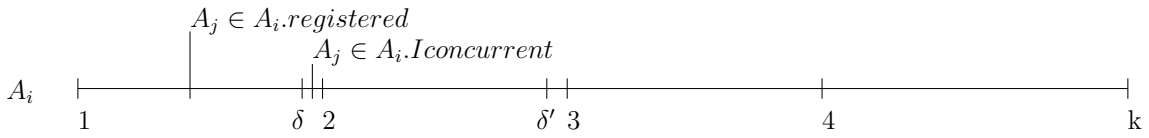


Figure 3: Dynamic Phase Synchronization

# A Brief description of Guarded Commands

1. (Multiple Assignment) x,y:= e1,e2 - denotes concurrent assignment of e1 and e2 to x and y respectively.

2. (Guard) g - denotes quantifier free boolean expression.

3. (Guarded Selection) $if\ g_1 \rightarrow S_1 [\!] \cdots [\!] g_n \rightarrow S_n\ fi$ - denotes nondeterministic selection. First, $g_i$'s, called guards, are computed at the entry and one of the branches corresponding to the guard is true. is selected and starts the execution of $S_i$. The command is said to terminate when $S_i$ terminates.

4. (Guarded Iteration) $do\ g_1 \rightarrow S_1 [\!] \cdots [\!] g_n \rightarrow S_n\ od$ - denotes the continued execution of the underlying guarded selection till at least one of the guards is true. Thus, when all the guards are false, the command terminates.

5. For simple iteration, we use the classical while loop: `while b do S endwhile` for the sake of easy understanding.

# B An Illustrative Description of the Solution

The solution achieves phase synchronization by ensuring that the set of processes that enter a phase is a common knowledge to all the processes. This set is denoted by the variable $A_i.Iconcurrent$ in the protocol and the processes in it continue to work in phase till they drop from the synchronization. After a process in $Iconcurrent$ starts the phase, a fresh process wanting to join the phase synchronization has to wait at least till the current phase is complete and at most could miss one phase . Figure 3 illustrates that if $A_j$ got included in $A_i.registered$ between 1 and $\delta$, where $\delta$ denotes the point where local snapshot is taken and the phase numbers $1, 2, 3 \ldots$ denote the completion of `CONSOLIDATE`, then $A_j$ can start from phase 2, otherwise anything registered between $\delta$ and $\delta'$ will start in phase 3. (Due to the asynchronous nature of processes, a global observer could observe that a process that intended to register in phase $k$ via process $j$ actually entered the synchronization only in phase $k+2$ as it could not register with process $j$ in phase $k$ itself (it could register only in phase $k+1$ as explained later)).

# C Single Clock Proof

In this section, we provide a proof in semi-formal way in the style of [1]. The proof obligations for synchronization and progress are given below. The symbol '$\mapsto$' denotes *leads to*.

- Synchronization

  We need to show that the postcondition of `SYNC;CONSOLIDATE;` (corresponding to barrier synchronization) for processes that have *proceed* set to *true* is :
  $\{\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow A_i.executing = A_j.executing)\}$

- Progress
  Property 1: The progress for processes already in phase synchronization is given by ($k$ is used to denote current phase) the following property which says that if all the processes have completed phase $k$, then each of the processes move to a phase greater than k if they do not drop out.
  P1: $\{\forall i(A_i.drop = false \wedge \forall j(A_j.drop = false \Rightarrow A_j.executing \geq k) \mapsto (A_i.executing \geq k + 1))\}$
  Property 2: The progress for new processes that want to join the phase synchronization is given by the following property which says that a process that gets registered will join the phase synchronization.
  P2: $\{\exists i((A_i.proceed = false \wedge \exists j(i \in A_j.registered)) \mapsto A_i.proceed = true)\}$

## C.1 Proof of Synchronization

The invariant that holds in the protocol is:
I1: $\{\forall i(A_i.proceed = true \Rightarrow (\forall j(A_j.proceed = true \Rightarrow (A_i.next \leq A_j.next + 1 \wedge (j \in A_i.Iconcurrent \vee A_i.next \leq A_j.next)))))\}$

It can be easily checked that each clock operation maintains this invariant. We show that for all processes involved in phase synchronization, following holds:
{I1 } `SYNC` $\{\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow (A_i.executing = A_j.executing \wedge$
$A_i.newIconcurrent = A_j.newIconcurrent))\}$
The initialization ensures that I1 is true and a process starts with the proper $Iconcurrent$ set. Let us look at the execution of the loops in the code block `SYNC`. The postcondition of the first do-od loop can be shown to be $\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow A_i.next = A_j.next)$ which satisfies the invariant. Let us see how this postcondition is obtained. The first guard ensures that another process gets removed from $A_i.Iconcurrent$ only when $A_i.next \leq A_j.next$. By symmetry, we can arrive at $A_i.next = A_j.next \vee j \in A_i.Iconcurrent$. The rest of the guards do not change this. Note that if $A_j$ drops, then $A_j.proceed$ is $false$, thus maintaining the invariant. The loop with guarded commands exits for $A_i$ when the only process left in $A_i.Iconcurrent$ is itself. Thus, eventually we will have the stated postcondition.

With $A_i.next = A_j.next$ at the end of first do-od loop, the post-condition of the second do-od loop can be shown to be:

$\{\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow (A_i.executing = A_j.executing \wedge A_i.newIconcurrent = A_j.newIconcurrent))\}$
The reasoning is simply that the guard checks for $A_i.executing = A_j.executing$ which can be true only if $A_i$ and $A_j$ have finished their first loop. At the end of the loop, $A_i.newIconcurrent = A_j.newIconcurrent$ is true because all the processes add/delete the same processes due to the common knowledge achieved by the set denoted by $Iconcurrent$. The property $A_i.newIconcurrent = A_j.newIconcurrent$ is very important for processes to dynamically discover the processes that want to join the phase synchronization and make it a common knowledge. The post-condition of second do-od is the post-condition of `SYNC`.

The post-condition of `CONSOLIDATE` can now be easily shown to be:
$\{\forall i, j((A_i.proceed = true \wedge A_j.proceed = true) \Rightarrow A_i.executing = A_j.executing)\}$
Note that this includes the new processes that got flagged to proceed. Another important property that holds after consolidatation is:
C1: A process knows the set of processes that it has to synchronize with for that phase.
This is because of the assignment $A_i.Iconcurrent := A_i.newIconcurrent$ and the fact that all the new processes in $A_i.newsynchproc$ are assigned the same values as that of $A_i$ before flagging them to proceed.

## C.2 Proof of Progress

Proof of Property 1:
We claim the following properties hold for a phase synchronization (In each of the properties, "$i$" is a free variable signifying that the property holds for all $i$). A property of the form p *ensures* q is proven for a program by showing that (1) for each statement s in the program, {p $\wedge$ $\neg$q} s {p $\vee$ q} holds, and (2) there is a statement t in the program for which {p $\wedge$ $\neg$q} t {q} holds; the statement t "establishes" the property. This notion of *ensures* has been used in [1]. A property is "stable" if it continues to be true once it becomes true.

A0: $\{\forall k(k \geq 0 \Rightarrow (A_i.executing = k \wedge A_i.next > A_i.executing))\}$
A1: $\{\forall k(k \geq 0 \Rightarrow (A_i.next \geq k+1$ stable till $A_i$ drops$))\}$
A2: $\{\forall k(k \geq 0 \Rightarrow (A_i.next \geq k+1 \wedge A_i.Iconcurrent = \{A_i\}\ ensures\ A_i.next > k+1))\}$
A3: $\{\forall k(k \geq 0 \Rightarrow ((A_i.next \geq k+1 \wedge (\forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1)) \wedge \{A_i\} \subset A_i.Iconcurrent)$
$ensures\ (A_i.next \geq k+1 \wedge (\forall j(A_j.next \geq k+1)) \wedge \{A_i\} \subseteq A_i.Iconcurrent)))\}$

It is straightforward to see that each property is suitable preserved by each statement in case of *ensures*, that is, if the precondition of *ensures* is met, then the postcondition is also met. We do not give the details for sake of brevity.

From A0-A3, we derive A4 given below for every $i$, such that $A_i.drop = false$ and $k \geq 0$.
A4: $\{(A_i.next \geq k+1 \wedge (\forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1)) \mapsto A_i.next > k+1\}$
Proof: A4 is proved by splitting it into A4.1 and A4.2 for the cases $A_i.Iconcurrent = \{A_i\}$ and $\{A_i\} \subset A_i.Iconcurrent$ respectively and then applying disjunction:

A4.1:

$$
\begin{array}{ll}
 & A_i.next \geq k+1 \wedge (\forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1)) \wedge A_i.Iconcurrent = \{A_i\} \\
\mapsto & \{\text{using A0 and A2}\} \\
 & A_i.next > k+1
\end{array}
$$

A4.2:

$$
\begin{array}{ll}
 & A_i.next \geq k+1 \wedge (\forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1)) \wedge \{A_i\} \subset A_i.Iconcurrent \\
\mapsto & \{using A3, A5 \text{and the induction rule-finite sets are well founded under subset ordering}\} \\
 & A_i.next \geq k+1 \wedge (\forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1)) \wedge A_i.Iconcurrent = \{A_i\} \\
\mapsto & \{\text{Applying transitivity with A4.1}\} \\
 & A_i.next > k+1
\end{array}
$$

The disjunction proves A4 because by protocol design $\{A_i\} \subseteq A_i.Iconcurrent$. Now, we prove the following property for all $k \geq 0$:
P1': $\{\forall i(A_i.drop = false \wedge \forall j(A_j.drop = false \Rightarrow A_j.next \geq k+1) \mapsto (A_i.next > k+1))\}$
First consider $k = 0$ for clock $c$: $A_c.next$ is initialized to 1 and $A_c.Iconcurrent = A_c$. By using A4, we can deduce that the P1'holds for k=0.
Next, consider $k > 0$: By A1, we know that $A_i.next \geq k+1$ and from I1 we know that $\forall i : A_i.next \geq k+1$ during synchronization. Using A1, I1, and A4, the P1' for $k > 0$ follows naturally for all $i$.
From A0 and A1, we deduce P1'$\Rightarrow$ P1.

Proof of Property 2 :
The second property of the proof of progress is for newly registered processes.
A5: $\{\forall i(A_i.proceed = false \wedge \exists j(i \in A_j.registered)\ ensures$
$(i \in A_j.newsynchproc))\}$
A6: $\{\forall i(A_i.proceed = false \wedge (\exists j(i \in A_j.newsynchproc))\ ensures$
$A_i.proceed = true)\}$

From A5-A6, we can deduce the progress property of a new process $i$. We first show: A7: $\{(A_i.proceed = false \wedge \exists j(i \in A_j.registered)) \mapsto A_i.proceed = true\}$

$$
\begin{array}{ll}
 & A_i.proceed = false \wedge \exists j(i \in A_j.registered) \\
\mapsto & using A5 \\
 & i \in A_j.newsynchproc \\
\mapsto & using A6 \\
 & A_i.proceed = true
\end{array}
$$

Since $A_i.proceed = false$ for new processes, the above can be written as
$\{\forall i(A_i.proceed = false \wedge \exists j(i \in A_j.registered)) \mapsto A_i.proceed = true)\}$

C2: There can be atmost a phase difference of one between registration and assignment to *newsynchproc*.
C3: There is a phase different of one from the time $A_i$ is found in $A_j.newsynchproc$ to the time that it is flagged to proceed.

Using C2 and C3, we can claim that if a process registers in phase $k$ and assigned to *newsynchproc* in $k$, then it will start the phase synchronization $k+1$. But, if it gets assigned to *newsynchproc* in $k+1$ as is possible for $A_c.registered$ set, then it will start from $k+2$.