Design and Verification of Distributed Phasers

Karthik Murthy, Sri Raj Paul, Kuldeep S. Meel,
Tiago Cogumbreiro, and John Mellor-Crummey

Rice University

Abstract. A phaser is an expressive barrier-like synchronization construct that supports dynamic task membership. Each task can participate in a phaser as a signaler, a waiter, or both. In this paper, we present a highly concurrent and scalable design of phasers for a distributed memory environment. Our design for a distributed phaser employs a pair of concurrent skip lists augmented with the ability to collect and propagate synchronization signals. To enable a high degree of concurrency, the addition and deletion of participant tasks are performed in two steps: a "fast single-link-modify" step followed by multiple hand-over-hand "lazy multi-link-modify" steps. We verify our design for a distributed phaser using the SPIN model checker. We employ a novel "message-based" model checking scheme to enable a non-approximate complete model checking of our phaser design. We guarantee the correctness of phaser semantics by ensuring that a set of linear temporal logic formulae are valid during model checking. We also present complexity analysis of the cost of synchronization and structural operations.

1 Introduction

Power consumption is now considered to be a very important parameter in the design of future HPC systems. Dynamic voltage and frequency scaling is an essential tool required to operate parallel systems within a tight energy envelope [10]. As a consequence, dynamic task-based programming models are gaining attention as an alternative to static SPMD models. Synchronization between tasks in the dynamic task-based programming models is becoming increasingly important, as noted in the report "Software Challenges in Extreme Scale Systems"[9].

Phasers are a general barrier-like synchronization primitive that supports dynamic registration of tasks. Each task has a choice of participation modes: signal-only, wait-only, and signal-wait. To date, the only phaser design available is for shared memory systems [11,12]. In this paper, we present a highly concurrent and scalable design of phasers for distributed memory parallel systems.

Recent designs for phaser-like synchronization include Alting barriers in Communicating Sequential Processes for Java (JCSP) [13] and Clocks in X10 [8]. While Clocks have been implemented for distributed memory environments, they use a non-scalable design in which a single `root` task collects information from all the participants [7]. Alting barriers similarly maintain global state in a centralized fashion. In contrast, our phaser design uses a scalable distributed protocol.

Synchronization protocols that take time linear in the number of participating tasks are not scalable. Protocols with sub-linear growth in time complexity are necessary. Skip lists [6] have long been used in shared memory environments, providing an expected time complexity of $O(\log n)$ for operations on a skip list containing n items. We make use of a pair of distributed concurrent skip lists as the backbone for a distributed phaser. Insert and delete operations on the skip lists enable a task to dynamically join or abandon a phaser. Additional operations on the skip lists support propagation of synchronization signals.

Proving the correctness of distributed protocols is difficult. The manual enumeration of communication interleavings is infeasible and writing formal proofs is error prone. For these reasons, we employ automated formal verification known as model checking to verify our design. We check whether our design satisfies the required phaser semantics with a quorum of Linear Temporal Logic (LTL) formulae. Model checkers explore all possible paths of execution, verifying the input LTLs at each point along these paths. During this process, the size of the state space needed to completely model check the operations on a distributed phaser is significantly more than a terabyte. However, we employ a novel "message-based" divide-and-conquer strategy to reduce the state space and provide a non-approximate complete model checking of our design. To the best of our knowledge, we are the first to employ a message-based scheme for a non-approximate model checking to prove the correctness of a distributed synchronization protocol.

In this paper, we explore the design of a distributed phaser, complexity of operations and its correctness. Our contributions are as follows:

- We describe a design for distributed phasers that employs a scalable decentralized event-driven approach to synchronize dynamic tasks.
- We prove livelock- and deadlock-freedom, semantic properties about synchronization and structural-modification operations through a novel "message"-based model checking scheme.
- We analyze the time and message complexity of operations on distributed phasers.

Section 2 introduces distributed phasers. Section 3 details the design and operations. Section 3.4 verifies our design using model checking. Section 4 derives the complexity of phaser operations. Section 5 discusses related work. Section 6 presents conclusions.

2 Distributed Memory Phasers

A phaser is a flexible, barrier-like primitive used to synchronize a group of parallel tasks [11]. A phaser enables each task to participate in one of three modes: signal-only, wait-only, signal-wait. This flexibility lets a phaser be used in a spectrum of synchronization patterns ranging from a barrier to a producer-consumer pattern.

A phaser supports five operations: `create`, `register`, `drop`, `signal`, and `wait`. `create` is a collective among a team of tasks that creates a phaser.

`register` adds a task as a participant, while `drop` lets a task remove its membership. The only way to invoke `register` is when a task spawns another: the spawner registers the spawnee. Operations `create` and `register` indicate whether a task participates as a signaler (signal-only), a waiter (wait-only), or both signaler and waiter (signal-wait). The participation mode affects the two remaining phaser operations `signal` and `wait`, explained next.

A phaser synchronization maintains a monotonically increasing global event counter called *phase*. To increment the counter, all signalers that have not dropped from the phaser must invoke `signal` exactly once. A waiter issues a `wait` to block until the phaser reaches a certain phase i, effectively observing the i-th collective event. Any task that is both a signaler and a waiter must always signal before waiting. A wait-only task will observe but not affect synchronization. In contrast, a signal-only task contributes to advancing phase, but waits for no other, *e.g.,* a producer in a producer-consumer pattern.

On distributed systems, tasks participating in a phaser may reside on different compute nodes and must interact with each other through messages[1]. Below, we detail the challenges of designing phasers for a distributed memory model and introduce our solutions to address these challenges.

1) Efficient creation, signal aggregation and diffusion among participants Communication costs are significantly higher than computation costs in a distributed memory model. Centralized algorithms lack scalability. Decentralized algorithms that grow sub-linearly in the number of communication interactions among participant tasks to perform phaser operations are necessary.

Skip lists [6] have long been used in shared memory environments, providing an expected time complexity of $O(\log n)$ for operations on a skip list containing n items. The items in a skip list participate at one or more levels. Every item participates at level 0. An item at level k participates at level $k+1$ with probability p. A skip list does not require rebalancing after insertion/removal of items to maintain expected logarithmic time complexity for all operations.

Intuitively, determining the phase of a phaser is equivalent to retrieving the phase information resident on signalers organized as members of a skip list while performing a min-reduce of the phase information along the retrieval path.

2) Efficient integration of dynamically created participants The expected cost of including a task into a distributed phaser should be cost effective in terms of the number of communication interactions needed.

In our design, the number of communication interactions to either `register` or `drop` a task is sub-linear in the number of phaser participants.

3) Concurrent synchronization and structural modifications A distributed phaser design needs to provide a separation of concerns by allowing synchronization signals to propagate through the underlying data structure while structural modifications (adding or deleting a task) are in progress.

We achieve this concurrency by factoring a register/drop into a sequence of sub-operations that can be interleaved with signaling operations. In particular,

[1] Our design of distributed phasers is idempotent to whether messages are one-sided (i.e., RDMA) or two-sided.

we factor every register/drop into a "fast single-link-modify" step followed by a "lazy multi-link-modify" step similar to the one presented by Crain et al. [3] to support higher levels of concurrency in a distributed memory environment.

3 Distributed Phaser Design

Our design for a distributed phaser employs a pair of distributed skip lists. Signalers self organize into a signal collection skip list (referred to as *SCSL*), which is used to aggregate signals to a designated signaler at the head of the list. Waiters self organize into a signal notification skip list (referred to as *SNSL*) that is used to diffuse phase information from the head of the list to all the waiters.

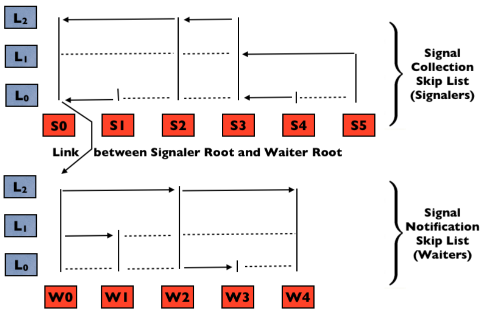

Fig. 1: Phaser synchronization achieved through signal collection and notification skip lists.

In a synchronization round, phase aggregation occurs in a right to left sweep with each signaler communicating the minimum phase of itself and its right neighbors to its highest level left neighbor in the *SCSL*. The designated signaler at the head of the *SCSL* conveys the aggregated phase to the designated waiter at the head of the *SNSL*, who then initiates a left to right diffusion of the phase to all the waiters.

To support non-blocking `signal` operations, we separate the implementation of a task into actions by a computation and communication thread. The computation thread executes the task, informs the communication thread at `signal`, and proceeds without blocking. The communication thread interacts with other such threads to perform the required *SCSL* actions. All the task actions described in this paper are those of the communication thread. We explicitly refer to the computation thread where necessary. In this section, we present detailed design descriptions of the creation and operations on *SCSL*. Managing the signal notification skip list - *SNSL* is similar, but simpler compared to *SCSL*. For lack of space, we omit the design of *SNSL*.

3.1 Distributed Skip Lists Creation

`Create` is a collective operation among a set of tasks that is used to create a phaser. Each task can specify whether it wants to participate in the phaser and if so, its participation mode. Invoking the `create` operation leads to the creation of both *SCSL* and *SNSL*, for which we employ the $O(\log n)$-based recursive doubling algorithm developed by Egecioglu et al. [4] without wrap-around. The algorithm proceeds in $\log n$ rounds of communication. In each round `i`, a task communicates with its hypercube neighbors at 2^i links away and accumulates left and right "frontiers" that indicate visible neighbors at each level.

3.2 Synchronization Signal Aggregation

Definition 1. `Local_phase` is the number of calls to `signal` by a signaler's computation thread. `Subtree_phase` at a signaler is the minimum `subtree_phase` across all the right neighbors connected to the signaler in the *SCSL* at their highest level and `local_phase` at the signaler itself. These right neighbors are referred to as `from_neighbor`s. Messages informing the `subtree_phase` at a signaler to the left neighbor at its highest level is called a synchronization signal. The left neighbor at the highest level is referred to as the `to_neighbor`.

For example, in Fig. 1, s_3 is the `from_neighbor` of s_2 and also the `to_neighbor` for both s_4 and s_5.

Single round of signal aggregation In a round of signal aggregation, each signaler issues a request (SRQ) to its `from_neighbor`s querying whether they can participate in the next phase, i.e., `subtree_phase`+1. On receiving an SRQ, a signaler waits for responses from all its `from_neighbor`s and waits for `local_phase` to equal the requested phase. After the wait conditions are satisfied, the signaler increments its `subtree_phase` and sends a response (SRP) to its `to_neighbor` and forming a request-response chain, which begins at the designated `root` signaler task in the *SCSL*. The request-response chain might, however, result in the ripple of requests from the `root` to the farthest signaler participating in the *SCSL*, and the ripple of responses back to the `root` in every synchronization round. To mitigate the latency of such ripples, we require that a signaler issues SRQ for the subsequent synchronization round immediately after sending a response to its `to_neighbor`.

Definition 2 (Synchronization signal invariant)**.** The `to_neighbor` of any signaler aggregates signals for the same or lower synchronization round than the signaler itself, formally:

$$\forall s \in SCSL, s.\texttt{subtree\_phase} \geq s.\texttt{to\_neighbor}.\texttt{subtree\_phase}$$

Since every signaler is transitively connected to the `root` in the *SCSL*, this invariant ensures that an increment of the `subtree_phase` at the `root` occurs only after all signalers in the *SCSL* have signaled for that round.

3.3 Registration of a Signaler

Our design supports the dynamic addition of a task into a phaser. Similar to phasers in shared memory, only a task currently participating in a phaser, referred to as parent, registers new tasks, referred to as children, into the phaser. By doing so, we provide the guarantee that a child begins participating in the same phase (`local_phase`+1) as that of its parent.

Inserting a child into the SCSL is decomposed into multiple steps to enhance concurrency. In each step, the granularity of locking is limited to a maximum of two links at two adjacent levels of the *SCSL*. First, a parent eagerly inserts a child into a link at the lowest level of the *SCSL*, i.e., L_0. Next, the child initiates a lazy hand-over-hand climb from one level to the next until its final level in the list; decision to move to level $k{+}1$ from k is based on probability p.

There are two modes of signal propagation for a child. After eager insertion, a child still needs the parent to propagate its signals since it might be at a lower phase than the `subtree_phase` of the left neighbor at L_0 of the *SCSL*. We refer to this state as a *transient* state. Once it reaches the `subtree_phase` of its `left_neighbor` at L_0, then it functions as a typical task in the *SCSL* and propagates its signals through its left neighbor. We refer to this state as a *normal* state. In the next two sections, we describe the two steps of eager- and lazy-insertion in detail.

3.3.1 Eager Single-link Modify Here, we describe in detail the pattern of communications between tasks in the *SCSL* needed to register a child task with the phaser. When a parent registers a child, the parent's computation thread blocks until the child is linked into the *SCSL* at L_0. The blocking ensures that no signals of the child are lost. To insert a child at L_0 of the *SCSL*, the first step is to find the location where the child should be linked. To do so, we employ the logical rank of the compute node on which a child will execute as a key.

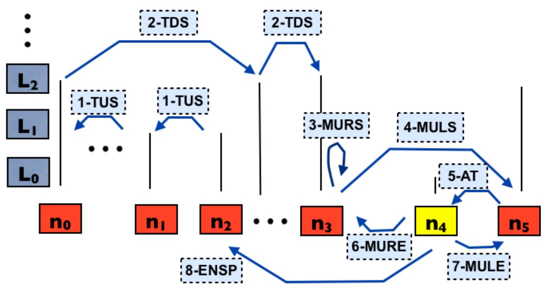

Fig. 2: Message sequence for addition of a task.

TUS	Travel Upstream during Spawn
TDS	Travel Downstream during Spawn
MURS/E	Moving Up of Right Neighbor Started/Ended
MULS/E	Moving Up of Left Neighbor Started/Ended
AT	Attach
ENSP	End Of Spawn

Fig. 2 illustrates the message sequence for linking a child to L_0 of the *SCSL* based on the child's key. In the figure, n_2 links its child n_4 into the *SCSL*. The first step in this process is to find neighbors n_3 and n_5 such that n_4's key lies between n_3 and n_5. To do so, n_2 initiates an upstream message chain, 1-TUS, that hops from a task to its `left_neighbor` at its highest level terminating at a task n_0, such that n_4's key lies between n_0 and its `right_neighbor` or the highest level of the *SCSL* is reached and n_0's `right_neighbor`'s key is less than the child's key. n_0 then initiates a downstream message chain, 2-TDS, that hops from a task to its `right_neighbor` until it ends at the L_0 link where the child should be linked. The `left_neighbor` of this L_0 link, n_3, enqueues 3-MURS on itself because another inclusion/drop might occur concurrently preventing n_3 from handling the 3-MURS immediately. Once, n_3 dequeues 3-MURS, it verifies whether the link with its current `right_neighbor`, n_5, remains valid for the child's inclusion. If so, n_3 proceeds to lock the link to prevent other structural changes and informs n_5 of the child through 4-MULS. n_5 sets its `to_neighbor` to n_4, locks its left neighbor at L_0 and sends 5-AT to n_4. n_4 sets its left and right neighbors at L_0 to n_3 and n_5 respectively and sends 6-MURE, 7-MULE and 8-ENSP. n_4 starts in the same phase as n_2. If `subtree_phase` of n_3 is higher than that of n_2, then n_4 needs to send signals to n_2 till it catches up with the synchronization round of n_3, i.e., *transient* state. On receipt of 6-MURE, n_3 sets its `right_neighbor` at L_0 to n_4 and unlocks it. On receipt of 7-MULE, n_5 sets its `left_neighbor` at L_0 to n_4 and unlocks it. The `right_neighbor` of n_3 and `left_neighbor` of n_5 are set at the end to ensure that search messages such as 1-TUS and 1-TDS are never blocked and go through a transient task only after its completely linked at L_0. On receipt of 8-ENSP, n_2 determines whether to maintain a signaling link with child task (n_4) and notifies its blocked computation thread to proceed.

3.3.2 Lazy Multi-link Modify The lazy hand-over-hand movement of a child to its final height in the *SCSL* does not begin until the child completes transition from *transient* state, i.e., signals through its parent, to *normal* state, i.e., signals through its `left_neighbor` at L_0 in the *SCSL*. The transition to *normal* state occurs once the child reaches the `subtree_phase` of its `left_neighbor` at L_0. In *normal* state, at each level *k*, the child decides to move to level *k+1* based on probability p until it reaches its final height. To move to level *k+1*, it needs to determine its neighbors at level *k+1*. Using a message chain similar to 1-TUS, the first neighbor on the left of the child with a height of *k+1* is determined. This neighbor, its `right_neighbor` at level *k+1*, and the child interact in a hand-shake message sequence exactly like the one for eager insertion to move the child to level *k+1*.

For lack of space, we do not provide details about the `drop` operation. The message exchanges are similar to the inclusion except the signaler is moving lazily from *k+1* to *k* before delinking itself from the *SCSL* completely.

3.4 Verification of *SCSL*

In this section, we show the correctness of *SCSL* operations with model checking [2]. In model checking, given a system (specified as a *configuration*) and some properties, a model checker tests these properties in all possible execution paths of the system. The goal of the *SCSL* verification is to show that the signal aggregated at the root is inclusive of signals from all registered signalers who haven't `drop`'ed; we call this property *root aggregation correctness.* To this end, we define a set of linear temporal logic (LTL) formulae that capture the root aggregation property. We check whether these formulae are satisfied during model checking. We employ a "message"-based strategy that consists of model-checking LTLs against a different configuration for each message type, say 1-TUS. We do so because a naive process-based model checking strategy required more than 1TB of RAM in our experiments.

We realize our verification using the state-of-the-art model checker Spin [5].The complete set of LTLs and configurations is available online at: http://goo.gl/ypuhaq

3.4.1 Root Aggregation Correctness We introduce three categories of properties: synchronization signal, structural consistency, and progress.
Synchronization signal Every signaler signals to its `to_neighbor` only after its `from_neighbors` and itself have signaled, i.e., *SCSL* maintains the synchronization signal invariant at all times. The synchronization signal invariant, Definition 2, guarantees the integrity of the phase aggregated at the `root` of the *SCSL*. The LTLs that capture this invariant are as follows:

– $\Box(\forall i,\ (!\ \texttt{is\_transient}(n_i) \implies (n_i.\texttt{subtree\_phase} \geq n_i.\texttt{to\_neighbor.subtree\_phase}))$
– $\Box(\forall i,\ (\texttt{is\_transient}(n_i) \implies n_i.\texttt{left\_neighbor}[\texttt{cur\_height}].\texttt{subtree\_phase} > n_i.\texttt{subtree\_phase}))$

Structural consistency Every signaler is transitively connected to the root, i.e., *SCSL* maintains structural consistency at all times. Every signaler has a single `to_neighbor` whose identifier is lesser than its own, and every signaler has at most one `from_neighbor` at each level of *SCSL*. This prevents any independent clusters in the *SCSL* and guarantees eventual connectivity to the root of the *SCSL*, thereby, ensuring that no signal from a signaler is lost.

– $\Box(\forall i, \forall L,\ (n_i.\texttt{left\_neighbor}[L] < n_i < n_i.\texttt{right\_neighbor}[L]))$ states that for every signaler, its identifier is always between its `left_neighbor` and `right_neighbor` at every level in which it participates. This monotonically increasing task-to-`to_neighbor` chain ensures that there are no independent loops of signalers that are not attached to the *SCSL*.
– $\Box(\forall i, \forall L,\ (n_i == n_i.\texttt{left\_neighbor}[L].\texttt{right\_neighbor}[L]))$ states that for every signaler, the `right_neighbor`'s `left_neighbor` is the signaler itself.
– $\Box(\forall i,\ n_i == n_i.\texttt{to\_neighbor.from\_neighbor}[\text{height}(n_i)]))$ states that every signaler always has a `to_neighbor` and that the `from_neighbor` of the `to_neighbor` at the height of the signaler is always the signaler itself.

Progress *SCSL* is deadlock- and livelock-free. This requirement ensures progress.

3.4.2 Message-based Verification Every phaser operation in our design is implemented as a series of message exchanges in the *SCSL*, where every message is handled atomically and terminates with the initiation of the next message needed for the operation. For example, if a task processes a 1-TUS then it either sends a 1-TUS or initiates the 2-TDS and does so atomically. Therefore, if each message of an operation can be processed correctly under any possible structural change and every message completes by starting the next message needed for the operation, then the operation is guaranteed to function correctly.

Message-based Modeling and Model Checking Our scheme uses a quorum of processes, signalers in our case, to undergo structural changes that challenge the successful completion of a single message in an operation on the *SCSL*. The structural changes include the source of the message delinking from the *SCSL* or moving lazily to a higher level, the destination of the message delinking from the *SCSL* or moving lazily to a higher level, and a new signaler linking between the source and destination and later delinking itself. These processes also have to complete a specific set of synchronization rounds. In the presence of such structural changes, if a message successfully completes, the LTL constraints are satisfied, and the specific number of synchronization rounds are complete, then we conclude that the handling of that message is correct.

Verifying 1-TUS message Consider the 1-TUS message in Fig. 2. n_2 initiates a 1-TUS to n_1 in the *SCSL*. The following structural changes can occur: n_1 can move down from L_1 to L_0, and n_1's new neighbor at L_0, say n_{01}, can drop out of the *SCSL*. To ensure the successful handling of 1-TUS message in these scenarios, we model check a configuration of 6 signalers $n_{0,01,1,2,3,4}$ such that n_2 inserts n_4, n_1 and n_{01} undergo structural changes as mentioned above. This configuration along with others needed to verify eager insertion are present in Table 1. In Table 1, column 1 describes the message while column 2-6 lists configurations of 5 tasks; the root n_0 participates at all levels, does not undergo structural changes, and hence, omitted from the table. Column 7 specifies the memory consumed and Column 8 specifies the number of states explored. A configuration of the task is specified as L:X*, where L indicates the initial level and X* is the sequence of operations comprising of D (drop), M(lazy move up), E[i] (eager insertion with parent task i).

Message	n_{01}	n_1	n_2	n_3	n_4	Mem (GB)	States
TUS	L_0:D	L_1:D	L_1	L_2	:E(2)	135	1.1e10
TDS	L_1:D	L_0:D	:E(0)	L_1	-	23	1.7e9
MURS	:E(0)D	:E(0)	L_0:D	-	-	10	5.6e8
MULS-1	L_0	L_0	:E(01)	:E(0)	-	78	7.4e9
MULS-2	L_0	L_0	:E(01)	L_0:M	-	86	6.7e9
MULS-3	L_0	L_0	:E(01)	L_1:D	-	50	4.3e9
AT	L_1	:E(0)	:E(0)MDD	-	-	6	3.1e8
ENSP	L_1	:E(0)	-	-	-	1	5.4e7

Table 1: Configurations used to model check the eager insertion of a signaler.

A Model of SCSL in PROMELA The input specification to Spin is the *SCSL* implemented in PROMELA along with the LTLs. We implement the *SCSL* as a group of processes (`proctype`s), one for each signaler. These signalers interact with each other using `channel`s; a `channel` holds messages sent from one process to another. Every signaler is configured to perform a specific number of phase advancements and its probabilistic height is decided a priori based on the configuration needed to verify a specific message. Every signaler executes a message-driven progress engine, which on receipt of a specific message responds with messages as specified in previous sections. We model check our configurations on a POWER7 compute node with 256GB RAM. A few experiments that needed more memory than 256GB were run on NERSC's Carver system, which had 1TB RAM. In total, we employed 23 configurations to verify all the messages in all operations on the *SCSL*.

Design Influenced by Model Checking: Tagging Messages with Link-sequence Numbers Monotonically increasing unique integral identifiers are assigned to links between tasks in the *SCSL* and messages are tagged with them. This design feature avoids problems due to stale messages. Consider the scenario in which n_3 initiates a move into the link between n_2 and n_4 at L_i. Concurrently, n_4 also decides to move into the next level, i.e., L_i to L_{i+1}, and issues an 1-LLNL to n_2. Before the 1-LLNL is processed at n_2, the following events occur: n_3 moves into the link between n_2 and n_4, n_3 processes the move up of n_4, n_3 drops out of the phaser, n_4 drops a level relinking itself to n_2, and n_2 processes the 1-LLNL issued by n_4 prior to these events. Processing the stale 1-LLNL leads to n_2 locking the link n_2-n_4 without n_4 having any intention of moving to the upper level. This led to the introduction of link identifiers.

4 Complexity Analysis

In this section, we present complexity analysis of synchronization and structure modification operations on the signal collection skip list - *SCSL*.

Complexity of Signal Aggregation The expected critical path length in a skip list from any task to the `root` is logarithmic in the number of tasks in the skip list. Hence, the expected time complexity taken by a signal from any participant in the *SCSL* to reach the designated `root` is $O(\log n)$, where n is the total number of signalers. The expected time complexity to aggregate signals from all the signaler tasks is also $O(\log n)$ since the aggregation occurs in parallel across all such chains.

Complexity of Participant Addition Here, we present complexity analysis of the expected number of message hops, i.e., pairwise communications, needed to insert a task to the *SCSL*. Eager insertion requires a skip list search, $O(\log n)$, to find the position to attach and a constant number of operations to finalize attach. Hence, eager insertion has a time and message complexity of $O(\log n)$. The rest of this discussion derives the complexity for moving a task lazily from L_0 to its eventual height.

Let there be a group of tasks that are lazily moving up to the higher levels between two stable tasks; stable tasks are those that have already reached their final height. We use K_i^j to indicate the j^{th} task at L_i and use $|K_i^j|$ to represent the distance between the left stable task and K_i^j. To this end, we abstract our model by making the following assumptions: (1) When considering the movement of tasks from L_i to L_{i+1}, there is a uniform probability distribution over the orders

in which they move up. For example, if tasks K_i^1, K_i^2, K_i^3 are moving up, then any of the 6 possible orders are equally likely. (2) The number of hops required for task K_i^j is
(a) $|K_i^j|$, if there is no task $|K_i^l| < |K_i^j|$ that moves to L_{i+1} before K_i^j, and
(b) $|K_i^j| - |K_i^l|$, if K_i^l moves to L_{i+1} before K_i^j and there is no other task K_i^t that reaches before K_i^j and $|K_i^t| > |K_i^l|$.

The key idea in our complexity analysis is to compute the expected number of messages for an arbitrary link, L_i. We then sum up the number of messages across levels and divide by the total number of inserted tasks to obtain per inserted task analysis. Before stating the main result, we prove three helper lemmas. Let m_i denotes the total number of intervals at L_i and m_T denote the total number of intervals at L_0.

Lemma 1. *Let C be the interval contention at L_0 in the SCSL and let the interval contention at L_i be denoted by C^i. Then $C * p^i \leq E[\mathsf{C}^i] \leq C$.*

Proof. Let X be the number of newly inserted tasks that move to L_i and Y is the number of stable tasks excluding root that are present at L_i. X and Y are independent of each other and are binomially distributed with probability p^i. m_T is the total number of intervals at L_0. By definition, $\mathsf{C}^i = X/(Y+1)$ and hence, $E[\mathsf{C}^i] = E[X/(Y+1)]$. Since X and Y are independent and binomially distributed with probability p^i, $E[X] = m_T C p^i$ and $E[1/(Y+1)] = \frac{(1-(1-p^i)^{m_T})}{m_T p^i}$. Since, $E[\mathsf{C}^i] = E[X] * E[1/(Y+1)]$, we have $C * p^i \leq E[\mathsf{C}^i] \leq C$. □

Lemma 2. *Let $K_i = \{K_i^1, \cdots, K_i^{n_i}\}$ be the tasks that move up from L_i to L_{i+1}, then the expected value of total number of hops for K_i, denoted by $E[\mathsf{Cost}(K_i)]$, is $\Sigma_{j=1}^{n_i} \frac{|K_i^j|}{n_i+1-j}$.*

Proof. We first note that $E[\mathsf{Cost}(K_i)] = \Sigma_{j=1}^{n_i} E[\mathsf{Cost}(K_i^j)]$. To compute, $E[\mathsf{Cost}(K_i^j)]$, we further partition the space of different configurations based on the order in which K_i^j moves up and use $M({K_i}^j, r)$ to denote the event that K_i^j is r^{th} task to reach the level $i+1$. Note that $\mathsf{Cost}(M({K_i}^j, r))$ depends only on the largest $K_i^l < K_i^j$ that reaches L_{i+1} before K_i^j. To this end, we use $MO(K_i^j, r, l)$ to denote the event that K_i^j is r^{th} task to reach L_{i+1} and K_i^l reaches before K_i^j and there is no other task K_i^t that reaches before K_i^j and $|K_i^t| > |K_i^l|$. We use $MO(K_i^j, 1, 0)$ to denote the event when K_i^j is the first task to reach L_{i+1}. Therefore, $E[\mathsf{Cost}(K_i)] = \Sigma_{j=1}^{n_i} \Sigma_{r=1}^{n_i} \Sigma_{l=0, l\neq j}^{n_i} E[\mathsf{Cost}(MO(K_i^j, r, l))]$. The rest of the proof is completed by first computing $E[\mathsf{Cost}(MO(K_i^j, r, l))]$ and then applying algebraic simplifications to compute $E[\mathsf{Cost}(K_i)]$.

To compute $E[\mathsf{Cost}(MO(K_i^j, r, l))]$, we first note that $E[\mathsf{Cost}\ (MO(K_i^j, r, l))] = Pr(MO(K_i^j, r, l)) \times \mathsf{Cost}(MO(K_i^j, r, l))$. Next, $Pr(MO(K_i^j, r, l))$ is (a) $\frac{1}{n_i} \prod_{t=1}^{n_i - l} (\frac{n_i - r - t - 1}{n_i - t})$ for $r \neq 1, j > l-1$, (b) 0 for $r \neq 1, j <= l-1$ and (c) 1/n for $r = 1$. Also, $\mathsf{Cost}(MO(K_i^j, r, l)) = |K_i^j| - |K_i^l|$ if $r \neq 1$, $|K_i^j|$ otherwise. Therefore, $E[\mathsf{Cost}(K_i^j)] = |K_i^j| - \Sigma_{t=1}^{j-1} \frac{|K_i^{j-t}|}{t(t+1)}$. Summing up over j, we obtain

$E[\mathsf{Cost}(K_i)] = \Sigma_{j=1}^{n_i} \frac{|K_i^j|}{n_i+1-j}$. To simplify this cost expression, we use the following lemma. □

Lemma 3. *Let* $|K_i^*| = \min_{j=1}^{n_i/2} \frac{|K_i^j|+|K_i^{n_i+1-j}|}{2}$, *then* $E(|K_i^*|) \geq \frac{p}{4}\mathsf{C}^i$.

Proof. $|K_i^*| = \min_{j=1}^{n_i/2} \frac{|K_i^j|+|K_i^{n_i+1-j}|}{2} \geq \min_{j=1}^{n_i/2} \frac{|K_i^j|}{2} + \min_{j=1}^{n_i/2} \frac{|K_i^{n_i+1-j}|}{2}$. Therefore, $|K_i^*| \geq \frac{1}{2} + \frac{|K_i^{n_i/2}|}{2} \geq \frac{|K_i^{n_i/2}|}{2}$. Since $E(|K_i^{n_i/2}|) \geq \frac{p}{2}\mathsf{C}^i$, $E(|K_i^*|) \geq \frac{p}{4}\mathsf{C}^i$. □

Theorem 1. *Let* $E[H_C]$ *be the expected number of hops consumed by a task inserted at* L_0 *to reach stable state, then* $\Omega(p^3 \log(Cp^3)) \leq E[H_C] \leq \mathcal{O}(\frac{p}{1-p} \log(C\frac{p}{1-p}))$.

Proof. To compute expected number of hops per task, we take the ratio of expected number of hops for all tasks inserted at L_0, denoted by $E[H_C^T]$ and the total number of tasks at L_0. Let $H_C^{T,i}$ denote the total number of hops consumed by tasks moving from L_{i-1} to L_i, then $E[H_C^T] = \Sigma_i E[H_C^{T,i}]$. From Lemma 2, we have $E[H_C^{T,i}] = E[m_i \Sigma_{j=1}^{n_i} \frac{|K_i^j|}{n_i+1-j}]$. Using Lemma 3 and $\forall j, K_i^j < K_i^{n_i}$, we have $E[m_i \Sigma_{j=1}^{n_i} \frac{|K_i^*|}{n_i+1-j}] \leq E[H_C^{T,i}] \leq E[m_i \Sigma_{j=1}^{n_i} \frac{|K_i^{n_i}|}{n_i+1-j}]$. From the proof of Lemma 1, we know that $E[n_i] = E[\mathsf{C}^i]p$. Similarly, following the proof of Lemma 1, we have $E[m_i] = m_T p^i$. Since $\Omega(\log n_i) \leq \Sigma_{j=1}^{n_i} \frac{1}{n_i+1-j} \leq \mathcal{O}(\log n_i)$. Next, $E[K_i^{n_i}] \leq C$ and noting the random variables m_i, n_i, K_i are independent, we have $m_T p^i \frac{p}{2} E[\mathsf{C}^i]\Omega(\log E[n_i]) \leq E[H_C^{T,i}] \leq m_T p^i C \mathcal{O}(\log E[n_i])$. Hence, $m_T p^i C \frac{p^{i+1}}{4} \Omega(\log(Cp^{i+1})) \leq E[H_C^{T,i}] \leq m_T p^i C \mathcal{O}(\log Cp)$. Therefore, $m_T C p^3 \Omega(\log(Cp^3)) \leq E[H_C^T] \leq m_T C \frac{p}{1-p} \mathcal{O}(\log(C\frac{p}{1-p}))$. Noting that the total number of tasks inserted at L_0 is $m_T C$ we have,
$\Omega(p^3 \log(Cp^3)) \leq E[H_C] \leq \mathcal{O}(\frac{p}{1-p} \log(C\frac{p}{1-p}))$ □

5 Related Work

Agarwal et al. present a distributed version of X10 clocks [1]. In this protocol, each task consults a local snapshot to determine the participant tasks and to make a decision about moving to the next phase. Processes add or drop themselves from these local snapshot. The authors, however, do not depict how this information is exchanged and state that in a basic implementation, one would require $O(n^2)$ messages. Our protocol describes the complete set of actions needed to ensure a total of $O(n)$ messages and $O(\log n)$ time complexity for synchronization using distributed skip lists.

In the non-blocking skip list protocol presented by Crain et al. [3], changes to the skip list structure are divided into two stages: eager abstract modification and lazy structural adaptation. They employ a single adaptive thread with global information to perform the structural changes based on neighborhood information. Our protocol is similar with two stages for insertion and deletion, but does not rely on an adaptive thread to perform the structural changes.

6 Conclusions

In this paper, we present a design for phasers, a general barrier-like synchronization construct that supports dynamic addition and deletion of parallel tasks, for a distributed memory-environment. Our design is based on a pair of distributed concurrent skip lists augmented with the ability to aggregate and diffuse phaser synchronization signals. By employing eager- and lazy-strategies while performing structural operations, our distributed phaser design supports a high-degree of concurrency. We employ a novel "message-based" model checking scheme to prove the correctness of our design. We derive the expected cost of signal aggregation, i.e., $\log n$ and cost for inclusion of a new task in the presence of interval contention `C`, i.e., $\Omega(p^3 \log(Cp^3)) \leq E[H_C] \leq \mathcal{O}(\frac{p}{1-p} \log(C\frac{p}{1-p}))$.

Acknowledgment We would like to thank Moshe Y. Vardi for the insightful discussions regarding verification. This research was supported in part by the DOE Office of Science Advanced Scientific Computing Research program through collaborative agreement DE-FC02-12ER26105. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. S. Agarwal, S. Joshi, and R. K. Shyamasundar. Distributed Generalized Dynamic Barrier Synchronization. In *Proc. of ICDCN*, pages 143–154, 2011.
2. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
3. T. Crain, V. Gramoli, and M. Raynal. No Hot Spot Non-blocking Skip List. In *Proc. of ICDCS*, pages 196–205, 2013.
4. O. Egecioglu, C. K. Koc, and A. J. Laub. A Recursive Doubling Algorithm for Solution of Tridiagonal Systems on Hypercube Multiprocessors. *Journal of Computational and Applied Mathematics*, 27(1):95–108, 1989.
5. G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
6. W. Pugh. Skip lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.
7. V. Saraswat et al. Source Distribution of X10 V2.5.1, 2014. `http://sourceforge.net/projects/x10/files/x10/2.5.1`.
8. V. Saraswat et al. X10 Language Specification Version 2.5, 2014. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.
9. V. Sarkar, W. Harrod, and A. E. Snavely. Software challenges in extreme scale systems. *Journal of Physics: Conference Series*, 180(1):12, 2009.
10. R. Schöne et al. Tools and Methods for Measuring and Tuning the Energy Efficiency of HPC Systems. *Scientific Programming*, 22(4):273–283, 2014.
11. J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A Unified Deadlock-free Construct for Collective and Point-to-Point Synchronization. In *Proc. of ICS*, pages 277–288. ACM, 2008.
12. J. Shirako and V. Sarkar. Hierarchical Phasers for Scalable Synchronization and Reductions in Dynamic Parallelism. In *Proc. of IPDPS*, pages 1–12, 2010.
13. P. Welch et al. Alting Barriers: Synchronisation with Choice in Java Using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, 2010.