# Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems: Formal Proof

**ABSTRACT**

The ability to consistently handle faults in a distributed environment requires, among a small set of basic routines, an agreement algorithm allowing surviving entities to reach a consensual decision between a bounded set of volatile resources. Based on communication concepts used by most parallel programming paradigms, this paper presents an algorithm that implements an Early Returning Agreement (ERA) in pseudo-synchronous systems, algorithm that optimistically allows a process to resume its activity while guaranteeing strong progress. We prove the correctness of our ERA algorithm, and expose its logarithmic behavior, an extremely desirable property for any algorithm targeted toward future exascale platforms. We detail a practical implementation of this consensus algorithm in the context of an MPI library, and evaluate both its efficiency and scalability through a set of benchmarks and two fault tolerant scientific applications.

**Categories and Subject Descriptors**

9 [System Software]: Support for Fault Tolerance

**Keywords**

MPI, Agreement, Fault-Tolerance

1. INTRODUCTION

The capacity to agree upon a common decision under the duress of failures is a critical component of the infrastructure for failure recovery in distributed systems. Intuitively, recovering from an adverse condition, like an unexpected process failure, is simpler when one can rely on some level of shared knowledge and cooperation between the surviving participants of the distributed system. In practice, while a small number of recovery techniques can continue operating over a system in which no clear consistent state can be established, like in many self-stabilizing algorithms [11], most failure recovery strategies are collective (like checkpointing [7, 14], algorithm based fault tolerance [20], etc.), or require some guarantee about the success of previous transactions to establish a consistent global state during the recovery procedure (as is the case in most replication schemes [15], distributed databases [29], etc.).

Because of its practical importance, the agreement in the presence of failure has been studied extensively, at least from a theoretical standpoint. The formulation of the problem is set in terms of a $k$-set agreement with failures [9, 29]. The $k$-set agreement problem is a generalization of the consensus: considering a system made up of $n$ processes, where each process proposes a value, each non-faulty process has to decide a value such that a decided value is a proposed value, and no more than $k$ different values are decided. In the literature, two major properties are of interest when considering a $k$-set agreement algorithm: an agreement is Early Deciding, when the algorithm can decide in a number of phases that depends primarily on the effective number of faults, and Early Stopping, when that same property holds for the number of rounds before termination. Strong theoretical bounds on the minimal number of rounds and messages required to achieve a $k$-set agreement exist [13], depending on the failure model considered (byzantine, omission, or crash).

In this paper, we consider a practical agreement algorithm with the following desired properties: 1) the unique decided value is the result of combination of all values proposed by deciding processes (a major difference with a $1$-set agreement), 2) failures consist of permanent crashes in pseudo-synchronous system (no data corruption, loss of message or malicious behavior are considered), and 3) the agreement favors the failure-free performance over the failure case, striving to exchange a logarithmic number of messages in the absence of failures. To satisfy this last requirement, we introduce a practical, intermediate property, called Early Returning: the capacity of an early deciding algorithm to return before the stopping condition (early or not) is guaranteed: as soon as a process can determine that the decision value is fixed (except if it fails itself), the process is allowed to return. However, because the process is allowed to return...
early, later failures may compel that process to participate in additional communications. Therefore, the decision must remain available after the processes returned, in order to serve unexpected message exchanges until the stopping condition can be established. Unlike a regular early stopping algorithm, not all processes decide and stop at the same round, and some processes participate in more message exchanges—depending on the location of failed processes in the logarithmic communication topology.

1.1 Use case: the ULFM Agreement

The agreement problem is a corner-stone block of many algorithms in fault-tolerant distributed computing, including the management of distributed resources, high-availability distributed databases, total order multicast, and even ubiquitous computing. In this paper, we tackle this issue from a different perspective, with the goal of improving the efficiency of the existing implementation of the fault tolerant constructs added to the Message Passing Interface (MPI) by the User-Level Failure Mitigation (ULFM) [3] proposal. Moreover, MPI being a de-facto parallel programming paradigm, one of our main concerns will be the time-to-solution, more specifically the scalability, of the proposed agreement.

The ULFM proposal extends the MPI specification by providing a well-defined flexible mechanism allowing applications and libraries to handle multiple types of faults. Two of the proposed extensions have an agreement semantic: MPIX_COMM_SHRINK and MPIX_COMM_AGREE.

The purpose of the MPIX_COMM_AGREE is to agree, between all surviving processes in a communicator (i.e. a communication context in the MPI terminology), on an output integer value and on the group of failed processes in the communicator. On completion, all living processes agree to set the output integer value to the result of a bitwise ‘AND’ operation over the contributed input values. When a process is discovered as failed during the agreement, surviving processes still agree on the output value, but the fact that a failed process’s contribution has been included is uncertain, and MPIX_COMM_AGREE raises an exception at all participating processes. However, if the failure of a process is known and acknowledged by all participants before entering the agreement, no exception is raised and the output value is simply computed without its contribution.

In more formal wording, this operation performs a non-uniform agreement in which all surviving processes must decide on 1) the same output value, which is a reduction of the contributed values; and 2) a group of processes that have failed to contribute to the output value, and the associated action of raising an exception if members of that group have not been acknowledged by all participants.

The purpose of the MPIX_COMM_SHRINK can be seen as an overload of the MPIX_COMM_AGREE, as it builds a new communicator (similar to the output integer value of the agreement), containing all processes from the original communicator that are alive at the end of the shrink operation. The output communicator is valid and consistent with all participants. Moreover, the design commands a strong process by requesting that a failed process that is acknowledged by any of the participants be excluded from the resulting communicator (preventing MPIX_COMM_SHRINK from being implemented as a MPI_COMM_DUP).

2. EARLY RETURNING AGREEMENT

In this section, we present the Early Returning Agreement Algorithm (ERA) using the guarded rules formalism: a guarded rule has two parts, 1) a guard, that is a boolean condition or that describes events as the detection of a failure or as a message reception, and 2) an action, that is a set of assignments or messages emissions. When a guard is true, or the corresponding event occurs, the process will execute all the associated actions. We assume that the execution of a guarded rule is atomic, and that the scheduler is fair (i.e., any guard that is true is eventually executed).

The algorithm is designed for a typical MPI environment with fail-stop failures: processes have a unique identifier; they communicate by sending and receiving messages in a reliable asynchronous network with unknown bounds on the message transmission delay; and they behave according to the algorithm, unless they are subject to a failure, in which case they do not send messages, receive messages, or apply any rule, ever. We also assume for simplicity that at least one process survives the execution. We assume an eventually perfect failure detector (νP in the terminology of [6, 27]) where every process has access to a distributed system that can tell if a process is suspected of being dead or not. This distributed system guarantees that only dead processes are suspected at any time (strong completeness), and all dead processes are eventually suspected of failure (eventual strong accuracy). Such a failure detector is realistic in a system with bounded transmission time (even if the bound is not known): see [10] for an implementation based on heartbeats that provide these properties with arbitrary high probability.

Last, let C be the set of all possible agreement values. All processes share an associative and commutative deterministic binary idempotent operation ⋃, such that C ⋃ ∅ with F is a monoid of identity 1.

To simplify the reading, we split the algorithm into multiple parts: ERA Part 1 presents the variables that are used and maintained by ERA: Procedures Decision, and Agreement are two procedures used by ERA; then ERA Part 3 to ERA Part 4 hold the guarded rules that are executed when some internal condition occurs (ERA Part 3), when a message is received (ERA Part 2), and when a process is discovered dead (ERA Part 4). Each of these algorithms is presented with the following notation: process p holds a set of variables for each possible agreement, νP, denoting the value of variable v for process p during the agreement a.

Algorithm ERA Part 1 presents the variables used for the algorithm. We write the algorithm without making assumptions on the number of parallel agreements, as long as each agreement is uniquely identified (by a in our notation). In MPI this unique identifier can be easily computed, and since an agreement is a collective call, all living processes in the communicator must participate in each agreement. Since communicators already have a unique identifier, deriving one for each agreement is as simple as counting the number of agreement calls in a given communicator.

Processes are uniquely identified with a natural number, and are organized in a tree. The tree is defined through two functions: 1) Parent(S, p) that returns the parent (a single process) of p, assuming that processes marked as dead in S are dead, and 2) Children(S, p), the children of p, assuming

\[ v' = \neg v \text{ [8]} \]
Procedure Decision presents the actions corresponding to deciding upon a value. For each agreement \( a \), each living process \( p \) eventually calls this procedure once, and only once. It then remembers the value decided for the agreement in \( \text{Result}_p^a \), and sends the decision to its children and all other processes that requested it (\( \text{RequestedResult}_n^a \)).

Procedure Decision\((v, a)\): Decide on \( v \) for agreement \( a \), and participate to the broadcasts of this decision.

<table>
<thead>
<tr>
<th>Input: ( v ): decision value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: ( a ): agreement identifier</td>
</tr>
<tr>
<td>( \text{Result}_p^a \leftarrow v )</td>
</tr>
<tr>
<td>for ( n \in \text{Children}(S_p, p) \cup \text{RequestedResult}_n^a ) do</td>
</tr>
<tr>
<td>( \text{Send}(\text{DOWN}(c, \text{Result}_c^a)) ) to ( n )</td>
</tr>
<tr>
<td>RequestedResult(n)(^a) (\leftarrow) \emptyset</td>
</tr>
</tbody>
</table>

ERA Part 2 describes how processes react to the reception of the different messages based on their local state. Each message handling is considered separately.

An UP message is received from a child, if the process is in a notcontributed or gathering state. In this case, the contribution of the child is taken into account, and the child is added to the contributors, potentially triggering the spontaneous rule (defined below) if it is the last child to contribute. It is also possible to receive an UP message from a child while in the broadcasting state: if a process sees its parent die before receiving the DOWN message, it will send its UP message again, even if its contribution could have already been accounted for. If the decision was already made, the process reacts by re-sending the DOWN message; otherwise it waits the decision to be taken, which will trigger the answer to the requesting child.

A DOWN message is received from a parent process if it is in the broadcasting state. It means that one of the elders \( \text{Eld}(S_p, p) \) has decided upon a pending agreement. In this case, the process also decides and broadcasts the result to its children and additional requesters (if any).

A RESULT REQUEST message is received from any process. Such a message is sent by processes when failures have changed the tree during an agreement, and a process needs to check if a previous decision was taken for that agreement. Different cases happen: if the receiving process took a decision for that agreement, it sends it back to the requester using a DOWN message; if the receiving process has not yet reached a decision, there are still two cases to consider:

- if the receiving process is in a broadcasting state, it is possible that the requester process is now a parent and the contribution of the receiving process was lost; the receiving process then sends back his saved contribution for the agreement;
- otherwise, the process remembers that the requester asked to receive the result of this agreement once reached.

ERA Part 3 presents a rule that must be executed when the process \( p \) reaches a state where it is gathering data, and all of its children and itself have contributed. In the rest of this document, we call this condition DC (for Deciding Condition). We distinguish two cases: if the process is the
ERA Part 2: Rules when a message is received

Rule \textbf{Recv}(UP(a,v)) from q \rightarrow
\begin{align*}
\text{if } & \text{Status}_p^v = \text{not contributed} \lor \\
\text{Status}_p^v &= \text{gathering then} \\
\text{Current}_p^v &\leftarrow F(\text{Current}_p^v,v) \\
\text{Contributed}_p^v &\leftarrow \text{Contributed}_p^v \cup \{q\} \\
\text{else if } & \text{Result}_p^v \neq \bot \text{ then} \\
\text{Send}(\text{DOWN}(a, \text{Result}_p^v)) \text{ to } q
\end{align*}

Rule \textbf{Recv}(\text{DOWN}(a,v)) from q \rightarrow
\begin{align*}
\text{if } & \text{Result}_p^v = \bot \land q = \text{Parent}(S_p,p) \text{ then} \\
\text{Decision}(v,a)
\end{align*}

Rule \textbf{Recv}(\text{RESULTREQUEST}(a)) from q \rightarrow
\begin{align*}
\forall r < q \text{ do} \\
\text{if } & \text{Result}_p^r \neq \bot \text{ then} \\
\text{Send}(\text{DOWN}(a, \text{Result}_p^r)) \text{ to } q \\
\text{else if } & \text{Status}_p^r = \text{broadcasting then} \\
\text{Send}(UP(a, \text{Current}_p^r)) \to q \\
\text{else} \\
\text{RequestedResult}_p^r &\leftarrow \text{RequestedResult}_p^r \cup \{q\}
\end{align*}

ERA Part 3: Spontaneous Rule

\begin{align*}
\text{Rule Status}_p^v &= \text{gathering} \land \\
\text{Children}(S_p,p) \cup \{\} \subseteq \text{Contributed}_p^v \rightarrow \\
\text{if } & \text{Parent}(S_p,p) = \bot \text{ then} \\
\text{Decision}(\text{Current}_p^v,a) \\
\text{else} \\
\text{Send}(UP(a, \text{Current}_p^v)) \to \text{Parent}(S_p,p) \\
\text{Status}_p^v &\leftarrow \text{broadcasting}
\end{align*}

ERA Part 4 describes the actions that a process takes when it discovers that another process has died. Note that the algorithm requires that the local failure detector monitors only the processes appearing in its neighborhood.

First, the \textit{Sp} array is updated to mend the tree. If the dead process was participating (or expected to participate) in an ongoing agreement, and it was the parent of the process \textit{p} that notices the failure, process \textit{p} will react: if it becomes the root of the new tree, it will start the decision process by first reentering the \textit{gathering} state, and then requesting all the processes that may have received the result of this agreement to contribute again. Otherwise, if it is not becoming root, it just sends its contribution to its new parent (\textit{UP} message), to ensure that the contribution is not lost.

If the dead process was one of the children of \textit{p}, the children of the dead process will become direct children of \textit{p} in the mended tree. They will eventually notice the death of their former parent (since they are monitoring it for a \textit{DOWN} message), and react by again sending an \textit{UP} message (see ERA Part 2). Eventually, this will trigger the rule of ERA Part 3 that makes process \textit{p} send its contribution up and wait for its parent to make a decision.

However, if \textit{p} was not the original root of the agreement, but it became root following a failure, and it discovers that one of its new children has died, then a process lower in the tree might have received the agreement decision from a previous root. Therefore, \textit{p} must request the contribution of all the children of any of its dead children (that is, grandchildren now becoming direct children).

ERA Part 4: Rule when a process is discovered dead

\begin{align*}
\text{Rule Process } q \text{ is discovered dead } \rightarrow \\
\text{if } \text{forall the a s.t. Status}_p^a = \text{broadcasting} \land \\
q = \text{Parent}(S_p,p) \text{ do} \\
\text{if } \text{Parent}(S_p,p) = \bot \text{ then} \\
\text{Contributed}_p^v &\leftarrow \{p\} \\
\text{Status}_p^v &\leftarrow \text{gathering} \\
\text{for } n \in \text{Children}(S_p,p) \text{ do} \\
\text{Send}(\text{RESULTREQUEST}(a)) \text{ to } n \\
\text{else} \\
\text{Send}(UP(a, \text{Current}_p^v)) \to \text{Parent}(S_p,p) \\
\text{for } n \in \text{Children}(S_p,q) \text{ do} \\
\text{Send}(\text{RESULTREQUEST}(a)) \to n
\end{align*}

2.1 Correctness

We define a correct agreement with the following traditional properties ([6, 27]), adapted to the MPI context. We say that process \textit{p} has contributed to value \textit{v}' if for any value \textit{v} proposed by \textit{p}, \textit{F}(v', v) = v'.

Termination Every living process eventually decides.

Irrevocability Once a living process decides a value, it remains decided on that value.

Agreement No two living processes decide differently.

Participation When a process decides upon a value, it contributed to the decided value.

\begin{theorem} \textbf{(Irrevocability).} Once a living process decides a value, it remains decided on that value. \end{theorem}
Proof. Processes decide in the procedure Agreement. When $Result^p_q$ is set to anything different from $\perp$ by the procedure Decision, the process returns. Thus, for a process, a decision is irrevocable (as it returns only once).

To prove the other properties, we introduce the following lemmas.

**Lemma 1 (Reliable failure detection).** For any process $p$, $q$, and any execution $E = C_0, C_1, \ldots$:

1. If $q$ and $p$ are alive in configuration $C_i$, $S_p[q] = \text{alive}$ in that configuration;
2. If $q$ is dead in configuration $C_i$, there is a configuration $C_{j_1} \geq i$ such that $S_p[q] = \text{dead}$ or $p$ is dead in $C_{j_1}$;
3. If $S_p[q] = \text{dead}$ in $C_i$, then $q$ is dead in $C_i$;
4. If $S_p[q] = \text{dead}$ in $C_i$, and $p$ is alive in $C_{j_2} \geq i$, then $S_p[q] = \text{dead}$ in $C_j$.

Proof. In $C_0$, all processes are alive, and $S_p[q] = \text{alive}$ for all $p$ and $q$.

No process assign $S_p[q]$ to $\text{alive}$, so once a process assigns $S_p[q]$ to $\text{dead}$, it remains so ($4$).

Processes assign $S_p[q]$ to $\text{dead}$ for two reasons: A) because the failure of $q$ was suspected in ERA Part 4. Since the failure detector is strongly complete, there is only a process that can send a message to $p$ to assign $S_p[q]$ to $\text{dead}$; B) because they receive a $\text{RESULTREQUEST}$ message from a process $r$ which is dead in ERA Part 2. We prove now by recursion that there is at most one process that can send such message in any configuration, and that this process is root. In $C_0$, only rank 0 is root. Since $p \geq 0$, rank 1 can only send $\text{RESULTREQUEST}$ messages if $0$ is suspected dead in ERA Part 4.

Assume that in $C_i$, there is only a process $r$ that is root, and that in the configuration $\text{RESULTREQUEST}$ messages come only from processes that were root in $C_{j_2} \leq i$. Let $C_{j_2}$ be the next configuration in which a failure happens. If a process receives such a message between $C_{j_2}$ and $C_i$, no process will assign $S_p[r]$ to $\text{dead}$, because $r$ is not suspected (eventual strong accuracy), and no process $t > r$ could become root to send a $\text{RESULTREQUEST}$ message.

If in $C_i$, $r$ dies, only process $p > r$ such that $p$ is the smallest alive identifier can become root. It will send a $\text{RESULTREQUEST}$ message to some processes, but that message comes from the root in $C_i$, so the property holds.

If in $C_i$, $p \neq r$ dies, $r$ remains the root, and no $\text{RESULTREQUEST}$ message is sent. Hence the property holds. By A) and B), we prove (1) and (3).

The failure detector being eventually strongly accurate, ERA Part 4 eventually triggers for any dead process, thus we deduce property ($2$).

**Lemma 2.** Eventually $\exists p$ s.t. $p$ is root and $p$ is alive.

Proof. At least one process survives the execution. Let $p = \min \{ q \text{ s.t. } q \text{ survives the execution} \}$. Because of the eventual strong completeness of $\perp$, Rule ERA Part 4 eventually triggers for all $q < p$. Thus, eventually, $\text{Arc}(S_p, p) = \emptyset$, and $\text{Eld}(S_p, p) = 0 \implies \text{Parent}(S_p, p) = \perp$.

**Lemma 3.** If $p$ is root at a given time, it remains root unless it dies.

Proof. Consider a configuration in which $p$ is root. For all processes $r$ such that $r < p$, $S_p[r] = \text{dead}$. By lemma 1, this remains true as long as $p$ lives. Thus, $\text{Parent}(S_p, p) = \perp$ as long as $p$ lives.

**Lemma 4.** A root process can only decide in ERA Part 3 after it contributed to the decided value.

Proof. The condition to decide or propagate the message $U/p$ to the parent in ERA Part 3 is $DC = (\text{Status}_p^a \neq \text{gathering}) \land (\text{Children}(S_p, p) \cup \{ p \} \subseteq \text{Contributed}_p^a)$.

If $q \in \text{Contributed}_p^a$, then there was a configuration before where $q \in \text{Children}(S_p, p)$ and $p$ received a message $U/p(a, v)$ from $q$, thus $Current^p_q$ holds $v$, the contribution of $q$ (see ERA Part 2). Since $F$ is an idempotent operation of a monoid of identity $\bot$, and the $Current^p_q$ is initialized to $\bot$, the ordering of contributions do not influence the result when all children have contributed (and the contribution of $q$ can be taken into account multiple times without changing the value of $Current^p_q$ as long as $q$ does not change its contribution, since $F$ is an idempotent operation).

If a child $q$ of process $p$ dies in configuration $C$, $DC$ remains false for $p$ until $S_p[q] = \text{dead}$ per ERA Part 4. Per lemma 1, there is a configuration $C'$ in which this is true. In that case, by definition of $\text{Children}$, all processes in $\text{Children}(S_q, q)$ in $C$ are in $\text{Children}(S_p, p)$ in $C'$. Thus, $DC$ can only be true when all processes have contributed. Since per lemma 1 those processes will eventually mark $S_p[q] = \text{dead}$, they will eventually send the $U/p$ message to $p$ (ERA Part 4).

Thus, when $DC$ is true for a process $p$, then all alive children of that process contributed. By definition of $\text{Parent}$ and $\text{Children}$, the combination of the contribution of all children of the root process is the combination of the contributions of all nodes in the tree.

**Theorem 2 (Agreement).** If process $p$ calls Decision $(v_1, a)$, and at a later time process $q$ calls Decision $(v_2, a)$, and $p$ is alive when $q$ calls the decision, then $v_1 = v_2$.

Proof. There are two ways for a process $p$ to decide:

If it is root, it may decide on $Current^p_q$, if the condition $DC$ holds. In that case, we prove first that all processes that may have decided before are dead.

Consider that a process $q$ decided $v_2$ before $p$ for the same agreement. This means that in a previous configuration $C_1$, there was a process $p'$ that was root, $p$ was not root in that configuration, and $p'$ is dead in the current configuration $C_2$ (lemma 3).

Since a root can only decide after all of its children contributed (lemma 4), $\text{Status}_p^a = \text{broadcasting}$ between $C_1$ and $C_2$. Thus, $p$ executed the $\text{Parent}(S_p, p) = \perp$ branch of ERA Part 4 to go from the status $\text{broadcasting}$ to the status $\text{gathering}$. Thus, $p$ sent a $\text{RESULTREQUEST}$ message to all $\text{Children}(S_p, p)$.

If any alive process in $\text{Children}(S_p, p)$ has received the decision of $p'$ between $C_1$ and $C_2$, then this process will answer to $p$ with a $\text{DOWN}(a, v_2)$ message (ERA Part 2), and $p$ will decide $v_2$ and the condition $DC$ cannot be true.

If no alive process in $\text{Children}(S_p, p)$ has received the decision of $p'$, then $q$ cannot have received the decision of $p'$: either $q \in \text{Children}(S_p, p)$, or $q$ was a descendant of one of the processes in $\text{Children}(S_p, p')$ in $C_1$ which is a subset of $\text{Children}(S_p, p)$ in $C_2$. Therefore $q$ has not decided.

If a process $q$ in $\text{Children}(S_p, p)$ dies before it answers to the $\text{RESULTREQUEST}$ message from $p$, its failure is eventually noticed by $p$ (because of the eventual strong accuracy of the detector, and because the condition $DC$ cannot be true unless an answer is received from $q$ or the death of $q$ is noticed by $p$). According to ERA Part 4, when $p$ notices...
the failure of \( q \), it firsts sends a `RESULTREQUEST` message to all children of \( q \) that will belong to its own children, before marking \( q \) as dead in \( S_p \), thus changing the time at which condition \( DC \) will become true.

Consider now the case of a non-root process. The only call to `Decision` is in ERA Part 2, when handling the `DOWN` message sent by \( p \), if a previous decision was not taken, and the message comes from the parent process.

By recursion on the hierarchy of the tree, any non root process takes the same decision as the root, and we proved above that if one process survives, any new root must take the same decision as this process.

**Theorem 3 (Termination).** Every living process eventually decides.

**Proof.** By lemma 2, there is eventually a root in the system. By lemma 4, the root eventually decides. We prove that if a root decides and remains alive, all processes eventually receive a `DOWN` message (triggering the decision).

When a process decides, it broadcasts the `DOWN` message to all its children (proc. `Decision`). If one of the children dies before receiving the `DOWN` message, its descendants are in the `gathering` status, thus ERA Part 4 makes them send an `UP` message to their new parent. By recursion on the topology of the tree, this parent is either the root, or a process that received the `DOWN` message, thus triggering the emission of a `DOWN` message.

**Theorem 4 (Participation).** When a process decides upon a value, it contributed to this value.

**Proof.** By theorem 3, all processes decide. By theorem 2, all decisions are equal to a decision of a root process. If a root process decides in ERA Part 2, then this decision comes from a previous root (proof of theorem 4). By recursion on the execution, all decisions originate in ERA Part 3. If a root process decides in ERA Part 3, then its decision includes the contribution of all alive nodes (lemma 4).

## 3. Complexity Analysis

**Theorem 5.** Let \( \delta \) be the maximum degree of the tree defined by the Parent/Children functions, and \( n \) the number of nodes in the tree: all processes decide in at most \( 2 \log_2 n \) parallel steps if no failures happen and at most \( O(2 \log_2 n + \delta) \) parallel steps if \( f \) failures happen.

**Proof.** Consider first the case when no failure happens during agreement \( a \): ERA Part 4 is never triggered, no process send `RESULTREQUEST` message related to agreement \( a \), and thus, \( \text{RequestedResult}^a_p \cup \text{Children}(S_p) = \text{Children}(S_p, p) \) for all process \( p \). Let \( d \) be the depth of the tree.

ERA Part 3 is triggered in parallel for all leaves of the tree, that send a single message `UP` to their parent. Non-leaf nodes receives exactly \( \text{Children}(S_p) \) `UP` messages, which makes the guard in ERA Part 3 true to them, and they then send one `UP` to their parents if they are not root. If all steps happen in parallel, the root process calls `Decision` after the information contained in the `UP` message of the deepest leaf of the tree reaches it, thus after \( d \) parallel steps.

In the `Decision` procedure, the root process sends exactly \( \text{Children}(S_p) \) `DOWN` messages to its children (since `RequestedResult^a_p \cup \text{Children}(S_p, p) = \text{Children}(S_p, p)`) for all process \( p \).

Every node \( p \) that receive such `DOWN` message does the same.

The algorithm terminates when the last leaf has received the `DOWN` message. If all steps happen in parallel, the last leaves to receive such message are the deepest in the tree, after \( d \) parallel steps.

If the maximum degree of the tree is \( \delta \), then the depth of the tree is \( \log_2 n \), and the algorithm terminates in \( 2 \log_2 n \) parallel steps.

Consider now that for some \( F \geq 0 \), if \( F \) failures happen during agreement \( a \), the algorithm completes in \( O(F \delta + \log_2 n) \) parallel steps.

Consider an execution \( E \) during which \( F + 1 \) failures happen while computing agreement \( a \). There is a prefix of \( E \) in which only \( F \) failures happened. Consider the configuration \( C \) at the end of that prefix. \( C \) is reached in at most \( O(F \delta + 1) \) parallel steps (by assumption).

Consider first the case when the \( F + 1 \) process that fails in \( E \) is the root process in \( C \). There is a configuration \( C' \) after \( C \) in which \( p \) is the new root for \( a \). If the decision reached another process, \( q \), \( p \) either received the decision, or one of \( \text{Children}_p \) received the decision (theorem 2). Let \( c \) be \( \{\text{Children}(S_p)\} \) in \( C' \) and the remaining configurations. As there was \( F + 1 \) failures, \( c \leq (F + 1)\delta \). Process \( p \) will send \( c \) `RESULTREQUEST` messages, and wait for the \( c \) `UP` or `DOWN` messages before deciding (ERA Part 4 and ERA Part 3). Then, it will broadcast the decision (ERA Part 3), which will take up to \( d \) parallel steps, completing the algorithm \( O(\log_2 n + (F + 1)\delta) \) parallel steps after \( C \).

Consider now the case when the \( F + 1 \) process that fails in \( E \) is not the root process in \( C \). Let \( p \) be the failing process. There is a configuration \( C' \) after \( C \) in which the parent \( q \) of \( p \) in \( C \) considers the children of \( p \) in \( C' \) as its children. In \( C' \), \( \{\text{Children}_q\} \leq (F + 1)\delta \). Thus, \( q \) must receive up to \( (F + 1)\delta \) `UP` messages, and send up to \( (F + 1)\delta \) `DOWN` messages. Thus, the algorithm will complete at most \( (F + 1)\delta + \log_2 n \) parallel steps after \( C \).

Since \( O(\log_2 n + F\delta) + O(\log_2 n + (F + 1)\delta) = O(\log_2 n + (F + 1)\delta) \), \( E \) completes in at most \( O(\log_2 n + (F + 1)\delta) \) parallel steps.

By recursion on the number of failures, the theorem holds.

## 4. ERA Optimizations

We incorporated a set of optimizations in our implementation of the algorithm. We kept these out of the formal presentation of the algorithm for clarity’s sake.

**Garbage Collection** ERA keeps a set of variables for every agreement. The storage of these variables is implemented through a hash table and variables default to their initialization value if they are not present in the hash table. Once a process returns from a given agreement, and since no other decision can be reached for the agreement, most of the variables that maintain the state of the agreement (e.g., `RequestedResult`, `Current`, `Status`, `Contributed`) have no further use and can be reclaimed. The result of the agreement itself, however, can be requested by another participant long after the result was locally returned and must be kept.

If the program would loop over agreements, this could exhaust the available memory. Because the ULM proposal of MPI allows for the definition of an immediate agreement (a nonblocking agreement call), causal dependency between agreements happening on the same communicator cannot
be enforced: immediate agreement \( i + 1 \) may complete and be waited upon by the calling program before immediate agreement \( i \). The implementation features a garbage collection mechanism: amongst the values on which every process agree during any agreement, we pass a range of agreement identifiers that were returned by the participating process, and the reduction function \( F \) computes the smallest intersection of the contributed ranges. When an agreement completes, it thus provides a global range of previous agreements that were returned by all alive processes of that communicator, and for which the \( \text{Result}^i \) variables can be safely disposed of, since they will never be requested again. When the communicator is freed, if agreements were used, a blocking flushing agreement is added to collect all remaining values on that communicator.

**Topology Deterioration Mitigation** As processes die, the tree on which ERA works deteriorates. Because trees are mended in a way that respects hierarchy (a process can only become the child of a node in its initial ancestry, or a child of the current root), the topology can deteriorate quickly to a star, risking the loss of the per-process-logarithmic message count property. This is unavoidable during the progression of a given agreement, because preserving the hierarchy of the initial tree is necessary to guarantee that only a subset of the alive processes need to be contacted for previous result requests when a process is promoted to root. However, between agreements, our ERA implementation mitigates this deterioration effect by allowing Tree Rebalancing.

During an agreement, processes also agree upon a set of dead processes. This enables the semantic required by ULFM on return codes (any non globally acknowledged dead process forces the agreement to consistently return a failure). We make use of this shared knowledge to maintain a consistent list of alive processes that can then be used to build balanced trees for future agreements, and therefore operating as if no failure had deteriorated the communicator.

**Topologies** The ERA algorithm is designed to work with any tree topology. In the performance evaluation, we will consider primarily the binary tree, although we implemented two other extreme topologies for testing purposes: the star (all processes connected directly to the root), and the string of processes. In addition to the topology, the implementation features an architecture-aware option, allowing for multi-level hierarchical topologies, where each level can have its own tree and root, and where the roots of one level are the participants of the next. Such hierarchical topologies allow for optimized mappings between the hardware topologies and the algorithmic needs, reducing the number of message exchanges on the most costly inter-process links. We denote the shape of the tree used as either a flat binary tree, when the locality-improvement is not used, or an \( X/Y \) tree, when groups are organized internally using a Y-tree, and between them using an X-tree (e.g., a bin/star tree organizes group roots along a binary tree, and each other process of a group is connected to its root directly).

5. PERFORMANCE EVALUATION

The ERA Algorithm is implemented in the ULFM fork of Open MPI 1.6(r26237) [4]. To follow its guarded rules representation, it is implemented just above the Byte Transfer Layer of Open MPI (below the MPI semantic layer):

```c
1. MPI_Comm_set_errhandler(MPI_COMM_WORLD, 
   MPI_ERRORS_RETURN);

2. MPI_Barrier(MPI_COMM_WORLD);

3. for(i = 0; i < WARMUP; i++) {
   rc = MPIX_Comm_agree(MPI_COMM_WORLD, &flag);
}

4. if( faults[rank] ) {
   raise(SIGKILL);
   do { pause(); } while(1);
}

5. start = MPI_Wtime();

6. rc = MPIX_Comm_agree(MPI_COMM_WORLD, &flag);

7. failure = MPI_Wtime() - start;

8. while(rc != MPI_SUCCESS) {
   MPIX_comm_failure_ack(MPI_COMM_WORLD);
   start = MPI_Wtime();
   rc = MPIX_Comm_agree(MPI_COMM_WORLD, &flag);
   stabilization += MPI_Wtime()-start;
 }

9. MPIX_Comm_agree(MPI_COMM_WORLD, &flag);

10. for(i = 0; i < NBTEST; i++) {
    start = MPI_Wtime();
    rc = MPIX_Comm_agree(MPI_COMM_WORLD, &flag);
    agreement += MPI_Wtime()-start;
 }
```

Figure 2: Code skeleton of the synthetic benchmark for the Agreement evaluation.

this enables the reception of \( \text{RESULTREQUEST} \) messages even when outside an MPIX_COMM_AGREE call, as imposed by the early returning property of the algorithm. Additionally, based on our prior studies highlighting the fact that local computations exhibiting linear behaviors dominate the cost, even in medium scale environments, we have taken extra steps to ensure that, when possible, all local operations follow a logarithmic time-to-solution.

This implementation was validated using a stress test that performs an infinite loop of agreements, where any failed process is replaced with a new process. Failures are injected by killing random MPI processes with different frequencies. A 24h run on 128 processors (16 nodes, 8 cores each, TCP over Gigabit Ethernet) completed 969,739 agreements successfully while tolerating 146,213 failures.

5.1 Agreement Performance

We deploy a synthetic benchmark on the NICS Darter supercomputer, a Cray XC30 (cascade) machine, to analyze the agreement latency with and without failures at scale. We employ the ugni transport layer to exploit the Cray Aries interconnect, and the as transport layer for inter-core communication.

5.1.1 Benchmark

To evaluate the performance of the ERA algorithm, we implemented a synthetic benchmark illustrated in Figure 2. After the initialization and warmup phase (lines 1–9), failures are injected in that benchmark at lines 10–13. Runs without failures are obtained by setting the entire faults
array to \texttt{false}, while a single node is set to \texttt{true} in the runs with failures reported below. Every surviving rank then reports the time to complete the failure agreement (lines 15–17). Not that for all measurements, when the rank that reports a number is not specified, this number represents the largest value among all ranks. Then, surviving processes enter the stabilization phase (lines 19–25): they acknowledge failures, and agree, until all processes agree that all failures have been acknowledged (see Section 1.1). We then use a single agreement (line 27) to synchronize the timing measurement between all processes, and measure the time to run \texttt{NBTTEST} agreements (lines 29–34), and report the average time as the agreement time.

**Scalability** In Figure 3a, we present the scalability trend of ERA when no failures are disturbing the system. We consider two different agreement implementations, 1) the known state-of-the-art 2-phase-commit Agreement algorithm presented in [22], called Log2phases, and 2) our best performing version of ERA. We also add, for reference, the performance of an Allreduce operation that in a failure-free context would have had the same outcome as the agreement.

With the bin/bin topology on the darter machine using one process per core, thus 16 processes per node, the average branching degree of non-leaf nodes is 2.125. The ERA and the Allreduce operations both exhibit a logarithmic trend when the number of nodes increase, as can be observed by the close fit (asymptotic standard error of 0.6%) of the logarithmic function \( \text{era}(x) = \log_{1.25}(x) \). In contrast, the Log2phases algorithm exhibits a linear scaling with the number of nodes, despite the expected theoretical bound proposed in [22]. As a result, we stopped testing the performance of the Log2phases algorithms at larger scale or under the non failure-free scenarios.

**Communication Topologies** In Figure 3b we compare the performance of different architecture-aware versions of the ERA algorithm. In the flat binary tree, all ranks are organized in a binary tree, regardless of the hardware locality of ranks collocated on cores of the same node. In the hierarchical methods, one rank represents the node and participates to the inter-node binary tree; on each node, collocated ranks are all children of the representing rank in the bin/star method, or are organized along a node-local binary tree in the bin/bin method. The flat binary topology ERA and the Open MPI Allreduce are both hardware locality agnostic; their performance profiles are extremely similar. In contrast, the Cray Allreduce exhibits a better scalability thanks to accounting for the locality of ranks. Counterintuitively, the bin/star hierarchical topology performs worse than the flat binary tree: the representing rank for a node has 16 local children and the resulting 16 sequential memcpy operations (on the shared-memory device) take longer than the latency to cross the supplementary long-range links. In the bin/bin topology, most of these memory copies are parallelized and henceforth the overall algorithm scales logarithmically. When compared with the fully optimized, non fault tolerant Allreduce, the latency is doubled, which is a logical consequence of the need for the ERA operation to sequentialize a reduce and a broadcast that do not overlap (to ensure the consistent decision criterion in the failure case), while the Allreduce operation is spared that requirement and can overlap multiple interweaved reductions.

**Impact of Failures** In Figure 3c we analyze the cost of failure-detecting, stabilizing and post-failure agreements as defined in Section 5.1. The cost of the failure-detecting agreement is strongly correlated to the network layer timeout and the propagation latency of failure information in the failure detector infrastructure (in this case, out-of-band propagation over a TCP overlay in the runtime layer of Open MPI). The stabilizing ERA exhibits a linear overhead resulting from the cost of rebuilding the ERA topology tree, an operation that ensures optimal post-failure performance, but optional for correctness. Indeed, the performance of a rebalanced post-failure ERA is indistinguishable from a failure-free agreement. When only one failure is injected, the cost of rebalancing the tree is not justified since the performance of the post failure non-rebalanced agreement is similar to the rebalanced agreement. Meanwhile, the cost of the stabilizing agreement without tree rebuilding is similar to a post-failure agreement, suggesting that the tree rebuilding should be conditional, and triggered only when the topology has degenerated after a large number of failures.

**5.2 Application Usage**

**5.2.1 S3D and FENIX**

S3D is a highly parallel method-of-lines solver for partial differential equations and is used to perform first-principles-based direct numerical simulations of turbulent combustion. It employs high order explicit finite difference numerical schemes for spatial and temporal derivatives, and realistic physics for thermodynamics, molecular transport, and chemical kinetics. S3D has been ported to all major platforms, demonstrates good scalability up to nearly 200K cores and has been highlighted by [1] as one of five promising applications on the path to Exascale.

Fenix is a framework aimed at enabling online (i.e., without disrupting the job) and transparent recovery from process, node, blade, and cabinet failures for parallel applications in an efficient and scalable manner. Fenix encapsulates mechanisms to transparently capture failures through ULFM return codes, re-spawn new processes on spare nodes when possible, fix failed communicators using ULFM capabilities, restore application state, and return the execution control back to the application. Fenix can leverage existing checkpointing solutions to enable automatic data recovery, but this evaluation uses application-driven, diskless, implicitly-coordinated checkpointing. Process recovery in Fenix involves four key stages: (i) detecting the failure, (ii) recovering the environment, (iii) recovering the data, and (iv) restarting the execution. In this section, we briefly describe the implementation of (i) and (ii), which relies on ULFM’s capability. The description of all the stages is available in our previous work [17].

**Failure detection** is delegated to ULFM-enabled MPI, which guarantees that MPI communications should return an \texttt{ERR_PROC_FAILED} error code if the runtime detects the process failure of a peer interrupted the operation. The error codes are detected in Fenix using MPI’s profiling interface. As a result, no changes in the MPI runtime itself are required, which will allow portability of Fenix when interfaces such as ULFM become part of the MPI standard.

**Environment recovery** begins with invalidating all communicators, and then propagating the failure notification to all ranks. In the current Fenix prototype, this is done
(a) ERA versus Log2phases Agreement scalability in the failure-free case.
(b) ERA performance depending on the tree topology.
(c) Post Failure Agreement Cost.

Figure 3: Synthetic benchmark performance of the agreement.

(a) Simultaneous failures on an increasing number of cores, over 2197 total cores.
(b) 256-cores failure (i.e., 16 nodes) on an increasing number of total cores.
(c) 16-cores failure (i.e., 1 node), on an increasing number of total cores.

Figure 4: Recovery overhead of the shrink operation, which uses the agreement algorithm. In Figure 4b, the subindex in the 4913-cores tests indicates a different distribution of failures within the 512-cores group.
using MPIX_COMM_REVOKE for all communicators in the system – the user must register their own communicators using Fenix calls. After that, a call to MPIX_COMM_SHRINK on the world communicator will remove all failed processes, while the other communicators are freed. If this step succeeds, new processes are spawned and merged with the old world communicator using the dynamic features of MPI-2. As this may reallocate rank numbers, Fenix uses the split operation to set them to their previous value. Note that this procedure allows $N \rightarrow 1$ simultaneous process failures, $N$ being the number of processes running. Alternatively, it is possible to fill the processes from a previously allocated process pool if the underlying computing system does not support MPI_COMM_SPAWN. Once Fenix’s communicators are recovered, a long jump is used to return execution to Fenix_Init(), except in the case of newly spawned processes – or processes in the process pool – which are already inside Fenix_Init(). From there, all processes, both survivors and newly spawned, are merged into the same execution path to prepare the recovery of the data.

In the rest of the section, we use S3D augmented with Fenix to test the effect of the new agreement algorithm on the recovery process when compared to the baseline agreement algorithm (revision b24c2e4 of the ULFM prototype). Figure 4 shows the results of these experiments in terms of total absolute cost of each call to the shrink operation. The MPIX_COMM_SHRINK operation, which uses the agreement algorithm, has been identified in [17] as the most time consuming operation of the recovery process. On Figure 4a we see how the operation scales with an increasing number of failures, from one node (16 cores) up to 64 nodes (1024 cores). We observe the drastic impact of the new ERA agreement algorithm compared with the previous Log2phases algorithm, and the absolute time is clearly smaller with the new agreement algorithm, in all cases. By using the new agreement, however, the smaller the failure, the faster it is to recover. This is a highly desirable property, as described in [18], and cannot be observed when using the former agreement algorithm, in which case the recovery time takes the same amount of time regardless of the failure size. The results shown in Figure 4b represent executions injecting 256-cores failures using an increasing total number of cores. The new agreement is not only almost an order of magnitude faster, but scales to numbers of processes not reachable before. It is also worth noting that the shape of the failure (i.e., the position of the nodes that fail, not only the number of nodes that fail) affects the recovery time with the new agreement algorithm, while this did not happen with the former. Finally, Figure 4c shows the scalability of the Fenix framework when injecting a 16-cores failure, which corresponds to a single node on Titan. As we can observe, the time to recover the communicator, while exhibiting a linear behavior, remains below 1.4 seconds when using more than 10,000 total cores. Clearly, we see a significant reduction in all cases.

As was shown in Section V.F of [17], the recovery cost due to communicator shrink accounts for 14% of the total execution time when simulating a 47-s MTBF (out of a total overhead due to faults and fault tolerance of 31%), a 7% with a 94-s MTBF (out of 15%) and a 4% with a 189-s MTBF (out of 10%) using S3D augmented with Fenix. Each of these experiments were done by injecting node failures (16 cores) in a total of 2197 cores. If we look at Figure 4a, we can observe that injecting 16-core failures in a 2197-core execution triggered a 6.85-second shrink with the former agreement algorithm and a 0.43-second shrink with the new agreement. Given that we see a 16-fold cost reduction of the shrink operation, it is safe to assume that the total overhead due to failures and fault tolerance has been reduced from 31% to 17.9%, from 15% to 8.4%, and from 10% to 6.2% for the 47-s, 94-s and 189-s MTBFs, respectively.

5.2.2 MiniFE and LFLR Framework

MiniFE is part of the Mantevo mini-applications suite [19] that represents computation and communication patterns of large scale parallel finite element analysis applications that serve a wide spectrum of HPC research. The source code is written in C++ with extensive use of templates to support multiple data types and node-level parallel programming models for exploration of various computing platforms, in addition to the flat-MPI programming model.

MiniFE has been integrated with a resilient application framework called LFLR (Local Failure Local Recovery) [31], which leverages ULFM to allow on-line application recovery from process loss without the traditional checkpoint/restart (C/R). LFLR extends the capability of ULFM through a layer of C++ classes to support (i) abstractions of data recovery (through a commit and restore method) for enabling application-specific recovery schemes, (ii) multiple options for persistent storage, and (iii) an active spare process pool to mitigate the complications from continuing the execution with fewer processes. In particular, LFLR exploits active spare processes to keep the entire application state in sync, by running the same program with no data distribution on the spare processes. Similar to Fenix (Section 5.2.1), the process recovery involves MPIX_COMM_SHRINK and several communicator creation and splitting calls to reestablish a sane execution environment. Contrary to Fenix, the state of the processes is periodically checked using MPIX_COMM_AGREE at the beginning of commit. This synchronization works as notification of failures across the surviving processes and also triggers the recovery operations. After the process recovery, the data objects are reconstituted through a restore call using the checkpoint data made at the previous commit.

The original MiniFE code only performs a single linear system solution with relatively quick mesh generation and matrix assembly steps. Despite its usefulness, the code may not reflect the whole-life of realistic application executions, running hours to simulate nonlinear responses or time dependent behaviors of physical systems. For these reasons, we have modified the code to perform a time-dependent PDE solution, where each time step involves a solution of a sparse linear system with the Conjugate Gradient (CG) method [31]. This modification allows us to study the situation where process failures happen in the middle of time stepping, and the recovery triggers a rollback to repeat the current time step after the LFLR recovery of processes and data. In addition to commit calls for checkpointing the application data at every time step (before/after CG solver call), MPIX_COMM_AGREE is called at every CG iteration inside the linear system solver. This serves as an extra convergence condition so that, when a process failure occurs, the solver can safely terminate in the same number of CG iterations across all surviving processes.

The performance of LFLR-enabled MiniFE is measured on the TlCC2 PC cluster at Sandia National Laboratories (see [31] for the details) using the ERA and Log2phases
Figure 5: Performance of the LFLR-enabled MiniFE, computing 20 time steps (20 linear system solutions).

(a) Process and communicator recovery.  (b) Global agreement during 20 time steps.  (c) Total execution time (single process failure).

(rev. b24c2e4) agreement from the ULFM prototype. In this study, the MiniFE code makes 20 time steps (20 linear system solutions) on 512, 1,024, and 2,048 processes with 32 stand-by spare processes, and problem sizes set to $(256 \times 256 \times 512)$, $(256 \times 512 \times 512)$ and $(512 \times 512 \times 512)$, respectively. The failure is randomly injected in a single process once during the execution.

Figure 5a presents the performance of communicator recovery, executed after a process failure has been detected. The ERA agreement achieves significantly better scaling than the Log2phases algorithm, imposing a low cost on the recovery process. Even when a single failure is considered, the benefit of the new agreement algorithm remains visible in Figure 5b, which indicates the total overhead, across all steps of the application, of the global agreement inside LFLR’s commit calls and the linear system solver in our MiniFE code. The total execution time presented in Figure 5c indicates that ERA improves the total solution time by approximately by 10% for a 2,048 processes case. However, the most interesting outcome of these results is that, even if at the scale where the results are presented in Figure 5a the absolute improvement is significant, but small, the scalability of the two approaches (Log2phases and ERA) are drastically different, suggesting that only one of these approaches could sustain the scale where the original application will be executed.

6. RELATED WORK

[16] determined long ago that without assumption on the delay of transmission of messages, the consensus problem was simply impossible to solve in distributed systems even with a single failure. This result called for a large set of studies (e.g., [6, 30, 21]) to refine the computability result of the consensus problem: the primary goal was to define the minimal set of assumptions that allows for solving the consensus despite failures. Few of these approaches have a practical application, or provided an actual implementation: the proposed algorithms are phase-based, and an $n^2$ communication (all-to-all exchange) is used during each phase. This phase based model does not directly match parallel programming paradigms based on asynchronous messages like MPI.

In volatile environments like ubiquitous computing and grids, probabilistic algorithms (known as gossip algorithms) have been proposed to compute, with a high probability, a consensus between loosely coupled sets of processes [2, 12]. However, the context of MPI, where the volatility is expected to be lower, the synchronization between processes is tighter, and the communication bounds are more reliable, calls for a more direct approach.

Paxos

[24, 23] is a popular and efficient agreement algorithm for distributed systems. It is based on agents with different virtual roles (proposers, acceptors, and learners) and decides on one of the values proposed by the proposers. Paxos uses voting techniques, to reach a quorum of acceptors that will decide upon a value. It is based on phases, during which processes of a given role will communicate with a large enough number of processes of another role to guarantee that the information is not lost. The first algorithm has been extended to a large family of algorithms, e.g., to tolerate byzantine failures [32], define distributed atomic operations on shared objects [25], or reduce the number of phases in the failure-free cases [26]. Paxos targets high-availability systems, like distributed databases, storing the state of the different processes in reliable storage to tolerate intermittent failures.

The first reason to consider a different algorithm is that in the MPI context, all processes contribute to the final decided value – that is a combination of the proposed values by a group of processes (namely, processes that were alive when entering the consensus routine). In Paxos, the decided value is one of the values proposed by the proposers [24]. This would require to first reducing the contribution to the subset of proposers (e.g., through an allreduce), and only then decide on this value using Paxos. Our algorithm implements the decision and the reduction in the same phase.

The second reason to consider a different algorithm is that the set of assumptions valid in a typical MPI environment is different from the assumptions made in Paxos: the failure model is fail-stop in our work, and we do not have to tolerate intermittent failures that Paxos considers; message loss and duplication are resolved at the lower transport layer, and we do not need to tolerate such cases; last, but not least, the concept of the process group and collective calls provided by MPI simplifies some steps of the algorithms, as the allocation of a unique number uniquely identifies the agreement, removing the need for one of the phases in Paxos.

Multiple Phase Commit

In [5], the author proposed a scalable implementation of an agreement algorithm, based on reduction trees to implement three phases similar to the three phases commit protocol: first, a ballot number is chosen; then a value is proposed; last it is committed. Each of these phases involves a logarithmic reliable propagation of information with feedback. Our algorithm provides the same functionality with better performance (a single logarithmic phase is used in the nor-
nal case), and better adaptability (as the reduction tree can be updated to maintain optimal performance).

In [22], the authors propose a two-phase commit approach, where processes gather the information to be agreed upon over a single, globally known root, and then broadcast the result. Similarly to [5], this algorithm was designed for a different specification, where only a blocking version of the algorithm is required. As a result, only up to two agreements can co-exist in the same communicator at any given time, simplifying the bookkeeping but limiting the usability of the algorithm. Theoretically, this approach is similar to our ERA in the failure-free case. However, unlike the ERA, this two-phase algorithm lacks, by design, the capability to be extended to a non-blocking agreement case. Moreover, a practical evaluation failed to demonstrate the expected logarithmic behavior of this two-phase algorithm, and instead demonstrated that in the failure-free case, the ERA implementation significantly outperforms this algorithm (see Section 5 for a discussion on the reasons). In the case of failures, the two-phase commit algorithm tries to re-elect a root to reconcile the situation. Stressing experiments show, however, that even a small set of random failures will eventually make the implemented election fail and the system enters a safety abort procedure for the agreement.

7. CONCLUSION

Facing the changes in the hardware architecture, together with the expected increase in the number of resources in future exascale platforms, it becomes reasonable to look for alternative, complementary, or meliorative solutions to the traditional checkpoint/restart approaches. Introducing the capability to handle process failure in any parallel programming paradigm, or providing any software layer with the facility to gracefully deal with failures in a distributed system will empower the deployment of new classes of application resilience methods that promise to greatly reduce the cost of recovery in distributed environments. Among the set of routines necessary to this goal, the agreement is bound to have a crucial role as it will not only define the performance of the programming approach, but will strictly delineate the usability and practicability of the proposed solutions. Based on the core communication concepts used by parallel programming paradigms, this paper introduces an Early Returning Agreement (ERA), a property that optimistically allows processes to quickly resume their activity, while still guaranteeing Termination, Irrevocability, Participation and Agreement properties despite failures. We proved this algorithm using the guarded rules formalism, and presented a practical implementation and its optimizations.

Through synthetic benchmarks and real applications, including large scale runs using a modified version of Fenix (S3D) and LFLR (MiniFE), we investigated the ERA costs, and highlighted the correspondence between the theoretical and practical logarithmic behavior of the proposed algorithm and the implementation. We have shown that using this algorithm, it is possible to design efficient application or domain specific fault mitigation building blocks that preserve the original application performance while augmenting the original applications with the capability to handle any type of future execution platforms at any scale.

Previous uses of the ULFM constructs spotlighted the overhead of the agreement operation as one of the major obstacles preventing a larger adoption of the concepts. The new ERA algorithm addresses this problem entirely, allowing the implementors to identify other limiting or poorly scalable elements of the fault management building blocks. From the performance presented in this paper, it becomes clear the next largest overhead is the failure detection. We plan to address this challenge in the near future.

8. ACKNOWLEDGMENTS

The authors would like to thank Robert Clay and Michael Heroux for interesting discussions related to this work. This work is partially supported by the CREST project of the Japan Science and Technology Agency (JST). This work is also partially supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

9. REFERENCES


