# Practical Scalable Consensus for Pseudo-Synchronous Distributed Systems

Thomas Herault
ICL, University of Tennessee

Aurelien Bouteiller
ICL, University of Tennessee

George Bosilca
ICL, University of Tennessee

Marc Gamell
Rutgers University

Keita Teranishi
Sandia National Laboratories

Manish Parashar
Rutgers University

Jack Dongarra
ICL, University of Tennessee
Oak Ridge National Lab.
Manchester University

## 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. This paper presents an algorithm that implements an Early Returning Agreement (ERA) in pseudo-synchronous systems, which 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, which is an extremely desirable property for any algorithm which targets 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.

## CCS Concepts

•**Computing methodologies** → *Distributed algorithms;* •**Computer systems organization** → **Reliability; Fault-tolerant network topologies;** •**Software and its engineering** → Software 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 [21], 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, 30]. 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 a combination of all values proposed by deciding processes (a major difference with a 1-set agreement), 2) failures consist of permanent crashes in a pseudo-synchronous system (no data corruption, loss of message, or malicious behaviors 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*: that is 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 pro-

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*SC '15, November 15 - 20, 2015, Austin, TX, USA*

© 2015 ACM. ISBN 978-1-4503-3723-6/15/11...$15.00

DOI: http://dx.doi.org/10.1145/2807591.2807665

cess 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 return, 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 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, to denote that uncertainty, `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[1] 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 progress 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`).

[1]NB: all living processes must still return the same value as the outcome of the agreement; only in the odd case when all processes that returned $v$ failed, may the surviving processes return $v' \neq v$ [8]

# 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 message 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 ($\diamond P$ in the terminology of [6, 28]) 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 $\mathcal{C}$ be the set of all possible agreement values. All processes share an associative and commutative deterministic binary idempotent operation $F$, such that $\mathcal{C} \bigcup \perp$ with $F$ is a monoid of identity $\perp$.

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, $v_p^a$, denoting the value of variable $v$ for process $p$ during the agreement $a$.

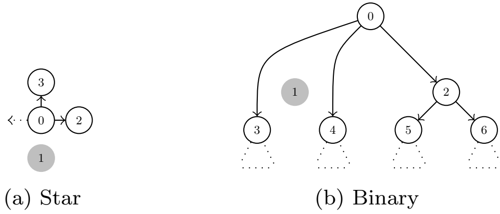
Algorithm ERA Part 1 presents the variables used for the algorithm. We write the algorithm without making assumptions on the number of parallel agreements, as long as each agreement is uniquely identified (by $a$ in our notation). In MPI this unique identifier can be easily computed, and since an agreement 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 $S$

**ERA Part 1:** Variables

**Variable**: $S_p$: array of process state (`dead` or `alive`), initialized with `alive` by default
**Variable**: $RequestedResult_p^a$: set of processes that asked $p$ to provide the result of agreement $a$, initialized to $\emptyset$ by default
**Variable**: $Result_p^a$: result of agreement $a$, as remembered by process $p$, initialized to $\perp$ by default
**Variable**: $Current_p^a$: current value of agreement $a$ for process $p$, initialized to $\perp$ by default
**Variable**: $Status_p^a$: current status of process $p$ in agreement $a$; one of `notcontributed`, `gathering`, `broadcasting`. Initialized to `notcontributed` by default
**Variable**: $Contributed_p^a$: list of processes that gave their contribution to process $p$ for agreement $a$. Initialized to $\emptyset$ by default

are dead, and 2) $Children(S, p)$, the children of $p$, assuming the same. The tree changes with the occurrence of failures. However, links in the tree are reconsidered if and only if one of the nodes has died. Figures 1a to 1b are examples of how the tree is mended when the process named 1 is dead, for different forms of the original tree.




(a) Star (b) Binary

Figure 1: Mended star and binary tree with node 1 dead

For the binary tree, the $Parent$ function is formally defined as follows. Consider the set $Anc(S_p, p)$ of ancestors of process $p$ defined as:

$$Anc(S_p, p) = \{q \text{ s.t. } q = \lfloor p/2^i \rfloor, i \geq 1 \wedge S_p[q] \neq \texttt{dead}\}$$

We also define the set of elders of process $p$ as:

$$Eld(S_p, p) = \{q < p \text{ s.t. } S_p[q] \neq \texttt{dead}\}$$

The Parent of process $p$, assuming that the dead processes (and only them) are marked `dead` in $S_p$ is, $Parent(S_p, p) =$

$$\begin{cases} \max Anc(S_p, p) & \text{if } Anc(S_p, p) \neq \emptyset \\ \min Eld(S_p, p) & \text{if } Eld(S_p, p) \neq \emptyset \wedge Anc(S_p, p) = \emptyset \\ \perp & \text{if } Eld(S_p, p) = \emptyset \wedge Anc(S_p, p) = \emptyset \end{cases}$$

Note that we call a process for which $Parent(S_p, p) = \perp$ *root*. The Children function can be generically written as:

$$Children(S_p, p) = \{q \text{ s.t. } Parent(S_p, q) = p\}$$

Procedure Agreement presents the actions that a process executes to participate in an agreement. Initializing the agreement consists of setting itself in the `gathering` mode, and combining the contributed value with the current value of the agreement. The process then waits for the decision to be reached before returning. While waiting, a process may serve requests for past agreements (agreements from which it already returned).

**Procedure** Agreement$(v, a)$: agreement routine.

**Input**: $v$: process's contributed value
**Input**: $a$: agreement identifier
**Output**: Agreement decided value
$Status_p^a \leftarrow$ `gathering`
$Contributed_p^a \leftarrow Contributed_p^a \bigcup \{p\}$
$Current_p^a \leftarrow F(Current_p^a, v)$
**Wait Until** $Result_p^a \neq \perp$ **Then**
  **return** $Result_p^a$

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 $Result_p^a$, and sends the decision to its children and all other processes that requested it ($RequestedResult_p^a$).

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

**Input**: $v$: decision value
**Input**: $a$: agreement identifier
$Result_p^a \longleftarrow v$
**for** $n \in Children(S_p, p) \bigcup RequestedResult_p^a$ **do**
  `Send(`$DOWN(c, Result_p^a)$`)` to $n$
$RequestedResult_p^a \longleftarrow \emptyset$

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 for the decision to be made, 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 $Eld(S_p, p)$ has decided on a pending agreement. In this case, the process also decides and broadcasts the result to its children and additional requesters (if any).

***A $RESULTREQUEST$ 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 2:** Rules when a message is received

**Rule** Recv($UP(a,v)$) *from* $q \longrightarrow$
- **if** $Status_p^a =$ *notcontributed* $\vee$ $Status_p^a =$ *gathering* **then**
  - $Current_p^a \longleftarrow F(Current_p^a, v)$
  - $Contributed_p^a \longleftarrow Contributed_p^a \bigcup \{q\}$
- **else if** $Result_p^q \neq \perp$ **then**
  - Send($DOWN(a, Result_p^a)$) to $q$

**Rule** Recv($DOWN(a,v)$) *from* $q \longrightarrow$
- **if** $Result_p^a = \perp \wedge q = Parent(S_p, p)$ **then**
  - $Decision(v, a)$

**Rule** Recv($RESULTREQUEST(a)$) *from* $q \longrightarrow$
- **forall the** $r < q$ **do**
  - $S_p[r] \longleftarrow$ dead
- **if** $Result_p^a \neq \perp$ **then**
  - Send($DOWN(a, Result_p^a)$) to $q$
- **else if** $Status_p^a =$ *broadcasting* **then**
  - Send($UP(a, Current_p^a)$) to $q$
- **else**
  - $RequestedResult_p^a \longleftarrow RequestedResult_p^a \bigcup \{q\}$

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 root for this agreement it makes the decision; if it is not the root, this triggers the normal propagation of contributions to the parent process. In both cases, the process enters into the `broadcasting` state.

**ERA Part 3:** Spontaneous Rule

**Rule** $Status_p^a =$ *gathering* $\wedge$ $Children(S_p, p) \bigcup \{p\} \subseteq Contributed_p^a \longrightarrow$
- **if** $Parent(S_p, p) = \perp$ **then**
  - $Decision(Current_p^a, a)$
- **else**
  - Send($UP(a, Current_p^a)$) to $Parent(S_p, p)$
- $Status_p^a \leftarrow$ broadcasting

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 $S_p$ 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 $p$ that notices the failure, process $p$ will react: if it becomes the root of the new tree, it will start the decision process by first reentering the `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 ($UP$ message), to ensure that the contribution is not lost.

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

However, if $p$ 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, $p$ 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

**Rule** *Process q is discovered dead* $\longrightarrow$
- $S'_p \longleftarrow S_p$
- $S_p[q] \longleftarrow$ dead
- **forall the** $a$ **s.t.** $Status_p^a =$ *broadcasting* $\wedge$ $q = Parent(S_p, p)$ **do**
  - **if** $Parent(S_p, p) = \perp$ **then**
    - $Contributed_p^a \longleftarrow \{p\}$
    - $Status_p^a \longleftarrow$ gathering
    - **for** $n \in Children(S_p, p)$ **do**
      - Send($RESULTREQUEST(a)$) to $n$
  - **else**
    - Send($UP(a, Current_p^a)$) to $Parent(S_p, p)$
- **forall the** $a$ **s.t.** $q \in Children(S'_p, p) \wedge$ $Parent(S'_p, p) = \perp \wedge Status_p^a =$ *gathering* **do**
  - **for** $n \in Children(S'_p, q)$ **do**
    - Send($RESULTREQUEST(a)$) to $n$

## 2.1 Correctness

We define a correct agreement with the following traditional properties ([6, 28]), adapted to the MPI context. We say that process $p$ has contributed to value $v'$ if for any value $v$ proposed by $p$, $F(v', v) = v'$.

**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.

THEOREM 1 (IRREVOCABILITY). *Once a living process decides a value, it remains decided on that value.*

PROOF. Processes decide in the procedure Agreement. When $Result_p^a$ 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). $\square$

To prove the other properties, we introduce the following

lemmas. We refer the reader to [19] for the formal proofs, and present only a sketch of the proof in this article.

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

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

Sketch of Proof. (1) and (3) are proven by the strong completeness of the failure detector (triggering the actions of ERA Part 4 only when a process is dead) and by proving by recursion on the execution that a process sends *RESULT-REQUEST* messages only if it is the root process in that configuration, thus preventing the corresponding rule in ERA Part 2 from marking a live process as `dead`.

(2) is a consequence of the eventual strong accuracy of the failure detector. The algorithm does not assign processes to `alive` after the initialization, thus property (4) holds. □

Lemma 2. *Eventually $\exists p$ s.t. $p$ is root and $p$ is alive, and it remains root forever unless it dies.*

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 $\diamond P$, all processes $q < p$ are eventually marked `dead` in $S_p$. Thus, eventually, $Anc(S_p, p) = \emptyset$, and $Eld(S_p, p) = \emptyset \Longrightarrow Parent(S_P, p) = \perp$.

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

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

Sketch of Proof. Consider a process $p$ that does not become root. By observation of the algorithm, if no children of $p$ dies, the condition $DC$ is true only when all children have contributed to $Current_p^a$ (Rule $Recv(UP)$).

The children set can change due to failures, but failures are eventually detected both on the parent and children of the failed process (lemma 1), and the condition $DC$ remains false until all the children of the dead process have contributed. Thus, when $DC$ is true on a node, all alive processes under that node in the tree have contributed. All nodes are under the root of the tree, so all nodes have contributed to the value decided by the root.

Consider now that $p$ becomes root during the execution. $p$ can adopt children that were not in its subtree. $DC$ might have been true temporarily for $p$ before it became root, but when it becomes root, any $UP$ message it sent was sent to a process that is now dead. By resetting $Contributed_p^a$ to $\{p\}$, in ERA Part 4, $DC$ becomes false for $p$ until all its children contribute (potentially multiple times; in that case, $Current_p^a$ is unchanged by repeated contributions because $F$ is idempotent). Thus, $p$ can only decide in ERA Part 3 when all the nodes have contributed. □

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 $v_1 = v_2$.*

Sketch of Proof. We prove first that when a root process decides, if it survives, all other surviving processes decide the same value. This is done by contradiction, showing that if another process decides a different value for the same agreement, that value must have come from another process than this root, and thus the decision was reached while the condition $DC$ is false for the root process, which is impossible. Lemmas 2 and 3 imply that the root process remains root, and that it can decide only after all processes in the tree have contributed to its decision.

Then, we prove that non-root processes always decide the same value as the root, by recursion on the topology of the tree, because all nodes remain connected to the root. □

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

Proof. By lemma 2, there is eventually a root in the system. By lemma 3, 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 3). 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 3). □

# 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 $2\log_\delta n$ parallel steps if no failures happen and at most $O(2\log_\delta n + f\delta)$ parallel steps if $f$ failures happen.*

Proof. Consider first the case when no failure happen during agreement $a$: ERA Part 4 is never triggered, no process send *RESULTREQUEST* message related to agreement $a$, and thus, $RequestedResult_p^a \bigcup Children(S_p, p) = Children(S_p, p)$ for all process $p$. Let $d$ be the depth of the tree.

ERA Part 3 is triggered in parallel for all leafs of the tree, that send a single message $UP$ to their parent. Non-leaf nodes receives exactly $|Children(S_p, p)|$ $UP$ 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 $|Children(S_p, p)|$ $DOWN$ messages to its children (since

$RequestedResult_p^a \bigcup Children(S_p, p) = Children(S_p, p)$). 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 $\log_\delta n$, and the algorithm terminates in $2\log_\delta n$ parallel steps.

Consider now that for some $F \geq 0$, if $F$ failures happen during agreement $a$, the algorithm *completes* in $O(F\delta + \log_\delta n)$ parallel steps. Consider an execution $\mathcal{E}$ during which $F+1$ failures happen while computing agreement $a$. There is a prefix of $\mathcal{E}$ in which only $F$ failures happened. Consider the configuration $C$ at the end of that prefix. $C$ is reached in at most $O(F\delta + 2\log_\delta n)$ parallel steps (by assumption).

Consider first the case when the $F+1$ process that fails in $\mathcal{E}$ 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 $Children_p(S_p, p)$ received the decision (theorem 2). Let $c$ be $|Children(S_p, p)|$ 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 $O(\log_\delta n + (F+1)\delta)$ parallel steps after $C$.

Consider now the case when the $F+1$ process that fails in $\mathcal{E}$ 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 $C'$, $|Children_q(S_q, q)| \leq (F+1)\delta$. 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 $(F+1)\delta + \log_\delta n$ parallel steps after $C$.

Since $O(\log_\delta n + F\delta) + O(\log_\delta n + (F+1)\delta) = O(\log_\delta n + (F+1)\delta)$, $\mathcal{E}$ completes in at most $O(\log_\delta n + (F+1)\delta)$ 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 (*i.e.*, $RequestedResult_p^a$, $Current_p^a$, $Status_p^a$, $Contributed_p^a$) 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 ULFM 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 on 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 $Result_p^a$ 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.

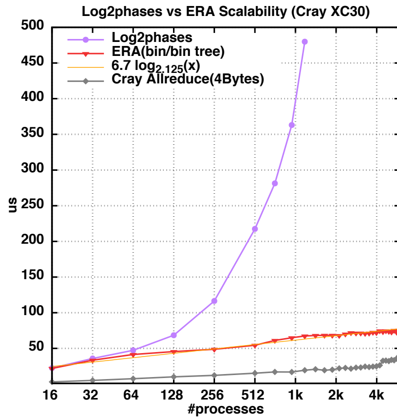
*Topology Deterioration Mitigation.* As processes die, the tree on which ERA works deteriorates. Because trees are mended in a way that respects hierarchy (a process can only become the child of a node in its initial ancestry, or a child of the current root), the topology can deteriorate quickly to a 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 Rebalacing*.

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.

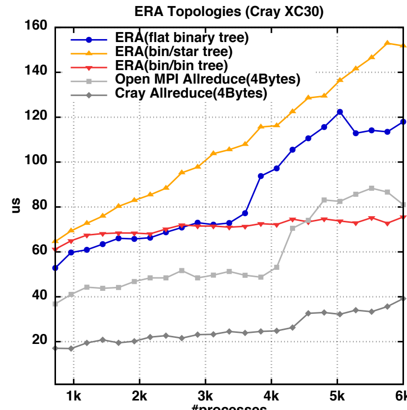
*Topologies.* The ERA algorithm is designed to work with any tree topology. In the performance evaluation, we will consider primarily the binary tree, although we implemented two other extreme topologies for testing purposes: the star (all processes connected directly to the root), and the string of 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

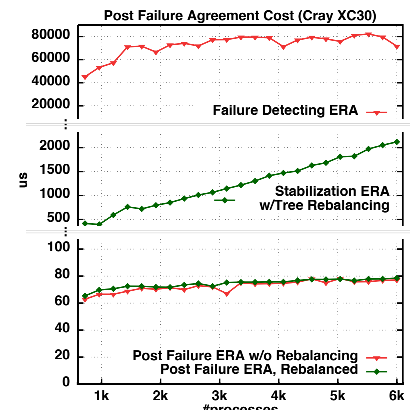
The ERA Algorithm is implemented in the ULFM fork of Open MPI 1.6(r26237) [4]. To follow its guarded rules rep-




(a) ERA versus Log2phases Agreement scalability in the failure-free case.




(b) ERA performance depending on the tree topology.




(c) Post Failure Agreement Cost.

| Failed Ranks | 0 (root) | 4 (child of 0) | 16 (node master) | 17 (child of 16) | 16–31 (full node) |
|---|---|---|---|---|---|
| Detecting Agreement | 12,659 | 93,816 | 80,023 | 112,414 | 82,171 |
| Stabilize Agreement | 104.9 | 102 | 98.9 | 104.2 | 117.1 |
| Post-failure Agreement | 69.7 | 75.7 | 77.1 | 76.7 | 85.2 |

(d) Cost ($\mu s$) depending on the role of the failed process in a bin/bin ERA w/o rebalancing, 6000 procs.

Figure 2: Synthetic benchmark performance of the agreement.

resentation, it is implemented just above the Byte Transfer Layer of Open MPI (below the MPI semantic layer): this enables the reception of $RESULTREQUEST$ 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 succesfully while tolerating 146,213 failures.

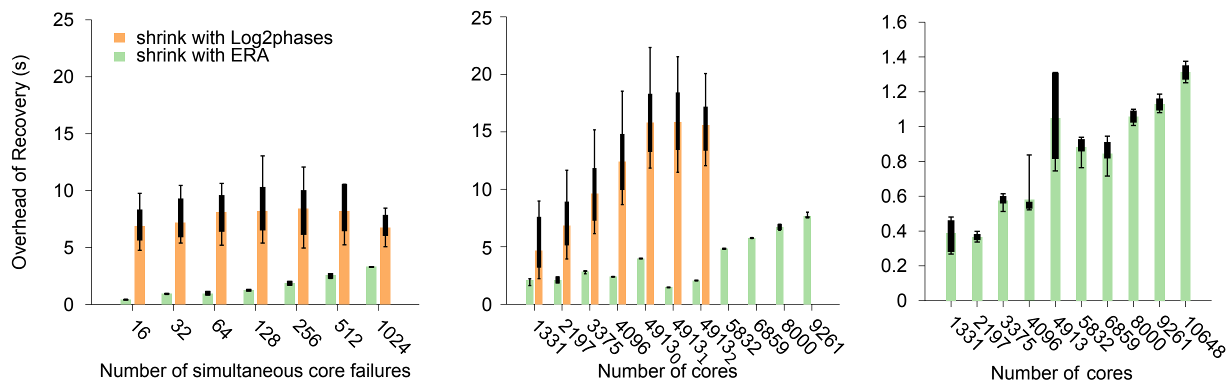
## 5.1 Agreement Performance

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 `sm` transport layer for inter-core communication.

The benchmark calls `MPIX_COMM_AGREE` in a loop, with failures injected at controllable iterations and processes. We consider four types of agreements: *failure-free* agreements precede the injection of a failure. The first agreement during which a failure manifests is the *failure detecting* agreement; it returns `MPI_ERR_PROC_FAILED` per ULFM specification. One additional *stabilizing* agreement, or more for complex failure scenarios, is then necessary to acknowledge the failure(s), optimize the agreement tree, and return `MPI_SUCCESS`. Subsequent *post-failure* agreements do not experience supplementary failures. For each participant, we collect the mean duration, and the standard deviation over 32k agreements; the reported mean time is the maximum between the mean times collected at all processes.

*Scalability.* In Figure 2a, 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 [23], 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 $era(x) = 6.7 \log_{2.125}(x)$. In contrast, the Log2phases algorithm exhibits a linear scaling with the number of nodes, despite the expected theoretical bound proposed in [23]. 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 2b 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 in 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 agnos-




(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 3: Recovery overhead of the shrink operation, which uses the agreement algorithm. In Figure 3b, the subindex in the 4913-cores tests indicates a different distribution of failures within the 512-cores group.

tic; 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 `memcopy` 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 2c 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 is 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.

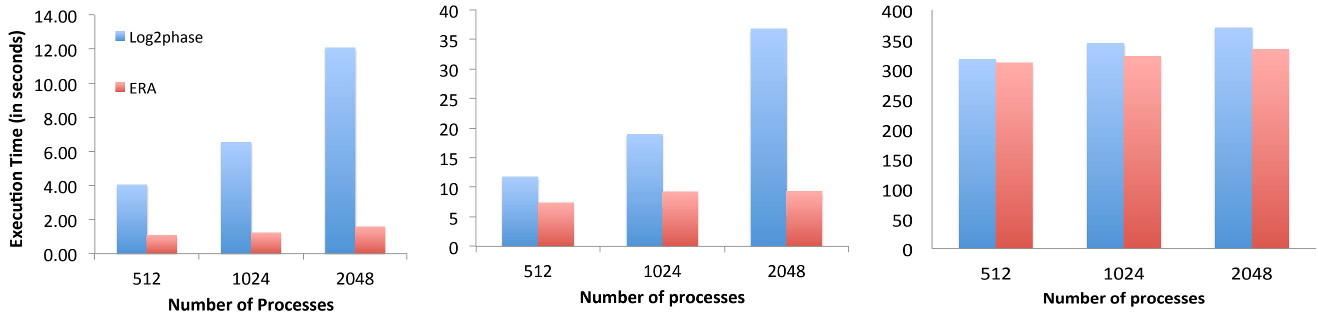
We considered other scenarios of failure in Table 2d. Starting from a setup with 6,000 processes, we used the same benchmark as above, but instead of always injecting failures on the same rank, we considered different cases of process failures: a) when the rank 0 fails (initial root of the agreement tree); b) when a direct child of the root of the agreement tree process fails; c) when a node-representative process fails; d) when some process that is not a node-representative fails; and e) when all the processes of an entire node fail but not the root of the agreement tree. As can be observed and was explained before, the detecting agreement is subject to a high latency due to limitations in the failure detection implementation; then the stabilize agreement pays the overhead of establishing additional connections to bypass the failed processes, and the post-failure agreements return to a small latency that is function of the new reduction tree. As the tree is not re-balanced in this experiment, one can observe a slight reduction of performance when the failure is injected lower in the tree. Hence, a practical approach would be to trigger the tree-rebalancing only when an agreement must be executed on a communicator after multiple failures. Moreover, in a context where the communicators are rebuilt after a failure, the cost of the tree-rebalancing can be spared.

## 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,




(a) Process and communicator recovery. (b) Global agreement during 20 time steps. (c) Total execution time (1 process failure).

Figure 4: Performance of the LFLR-enabled MiniFE, computing 20 time steps (20 linear system solutions).

(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 `ERR_PROC_FAILED` error code if the runtime detects that a process failure prevents the successful completion of 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 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 reassign rank numbers, Fenix uses the split operation to set them to their previous value. Note that this procedure allows $N-1$ 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 3 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 3a 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 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 3b 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 a number 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 3c 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 3a, 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 [20] 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 sup-

port 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) [32], 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 consistent 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 a 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 [32]. 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 [32] for the details) using the ERA and Log2phases (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 4a 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 4b, 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 4c 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 4a where 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, 31, 22]) 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.*

[25, 24] 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 [33], define distributed atomic operations on shared objects [26], or reduce the number of phases in the failure-free cases [27]. 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 [25]. This would require first reducing the contribution to the subset of proposers (*e.g.*, through an allreduce), and only then deciding 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 normal case), and better adaptability (as the reduction tree can be updated to maintain optimal performance).

In [23], 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 agreement in necessary. 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 faculty 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 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 highlighted 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 that 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, Michael Heroux and Josep Gamell for interesting discussions related to this work. This work is partially supported by the NSF (award #1339820), and 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

[1] S. Amarasinghe and et al. Exascale Programming Challenges. In *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Jul 2011.

[2] T. Aysal, M. Yildiz, A. Sarwate, and A. Scaglione. Broadcast gossip algorithms for consensus. *Signal Processing, IEEE Transactions on*, 57(7):2748–2761, July 2009.

[3] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing and Applications*, 27(3):244–254, 2013.

[4] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of

User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[5] D. Buntinas. Scalable distributed consensus to support MPI fault tolerance. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012*, pages 1240–1249, Shanghai, China, May 2012.

[6] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[8] B. Charron-Bost and A. Schiper. Uniform consensus is harder than consensus. *J. Algorithms*, 51(1):15–37, Apr. 2004.

[9] S. Chaudhuri, M. Erlihy, N. A. Lynch, and M. R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, Sept. 2000.

[10] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Computers*, 51(1):13–32, 2002.

[11] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974.

[12] A. Dimakis, A. Sarwate, and M. Wainwright. Geographic gossip: efficient aggregation for sensor networks. In *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, pages 69–76, 2006.

[13] D. Dolev and C. Lenzen. Early-deciding consensus is expensive. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 270–279, New York, NY, USA, 2013. ACM.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[15] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[17] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[18] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar. Exploring Failure Recovery for Stencil-based Applications at Extreme Scales. In *The 24th International ACM Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, June 2015.

[19] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. J. Dongarra. Practical scalable consensus for pseudo-synchronous distributed systems: Formal proof. Technical Report ICL-UT-15-01, University of Tennessee, Innovative Computing Laboratory, http://www.icl.utk.edu/~herault/TR/icl-ut-15-01.pdf, April 2015.

[20] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[21] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 100(6):518–528, 1984.

[22] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *distributed computing*, pages 209–223, 1999.

[23] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham. A Log-scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 255–263, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[25] L. Lamport. PAXOS made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4 – Whole Number 121):51–58, Dec. 2001.

[26] L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[27] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[28] M. Larrea, A. Fernández, and S. Arévalo. Optimal Implementation of the Weakest Failure Detector for Solving Consensus. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 334–, New York, NY, USA, 2000. ACM.

[29] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *SIGOPS OSR*, volume 19, pages 40–52. ACM, 1985.

[30] P. Raipin Parvedy, M. Raynal, and C. Travers. Strongly terminating early-stopping k-set agreement in synchronous systems with general omission failures. *Theory of Computing Systems*, 47(1):259–287, 2010.

[31] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, pages 149–157, 1997.

[32] K. Teranishi and M. A. Heroux. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM.

[33] P. Zielinski. Paxos At War. In *Proceedings of the 2001 Winter Simulation Conference*, 2004.