J Supercomput (2015) 71:2516–2538
DOI 10.1007/s11227-015-1402-3

CAMEL: collective-aware message logging

Esteban Meneses · Laxmikant V. Kalé

Published online: 13 March 2015

Abstract The continuous progress in the performance of supercomputers has made possible the understanding of many fundamental problems in science. Simulation, the third scientific pillar, constantly demands more powerful machines to use algorithms that would otherwise be unviable. That will inevitably lead to the deployment of an exascale machine during the next decade. However, fault tolerance is a major challenge that has to be overcome to make such a machine usable. With an unprecedented number of parts, machines at extreme scale will have a small mean-time-between-failures. The popular checkpoint/restart mechanism used in today's machines may not be effective at that scale. One promising way to revamp checkpoint/restart is to use message-logging techniques. By storing messages during execution and replaying them in case of a failure, message logging is able to shorten recovery time and save a substantial amount of energy. The downside of message logging is that memory footprint may grow to unsustainable levels. This paper presents a technique that decreases the memory pressure in message-logging protocols by only storing the necessary messages in collective-communication operations. We introduce CAMEL, a protocol that has a low memory overhead for multicast and reduction operations. Our results show that CAMEL can reduce memory footprint in a molecular dynamics benchmark for more than 95 % on 16,384 cores.

Keywords Fault tolerance · Resilience · Message logging · Collective-communication operations

E. Meneses (✉)
Center for Simulation and Modeling, University of Pittsburgh, Pittsburgh, PA, USA
e-mail: esteban.meneses@acm.org; esteban.meneses@gmail.com

L. V. Kalé
Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL, USA
e-mail: kale@illinois.edu

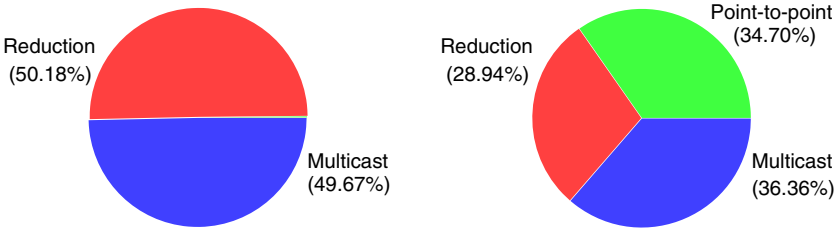
1 Introduction

The persistent advance in supercomputing has made possible the exploration of many fundamental problems in science. Methods that were once considered intractable are now practical, thanks to the availability of well-established petascale systems. The next step in the evolution of supercomputers, fueled by many performance-hungry problems in computational science, will lead to the deployment of exascale machines in the next decade. The computational power provided by extreme-scale systems is considered a fundamental tool in extending the frontier of our knowledge of nature and the universe. However, a threat glooms the future of supercomputing. The mere number of components assembled in an exascale computer will dramatically decrease the mean-time-between-failures (MTBF) of the system. Projections at exascale estimate the MTBF will be measured in minutes [6,19,27]. Even today, large-scale systems face frequent failures [5] and it is estimated more than 20 % of the utilization of the machine is lost due to failures [9].

Fault tolerance will be an ineludible consideration for extreme-scale computing. The traditional way to tolerate failures in high-performance computing (HPC) systems is to use rollback-recovery techniques [10]. Checkpoint/restart is the most popular strategy and consist in periodically saving the state of the system (checkpoint) to rollback to the latest checkpoint in case of a failure. Although checkpoint/restart has several available implementations [14,25,29], it is clear that this scheme alone will not provide an effective resilient solution at exascale and beyond [12,23]. A promising technique to revamp checkpoint/restart is to add message logging. By storing the messages sent during an execution, it is possible to avoid a *global* rollback and instead only rollback the crashed node. That way, the recovery time can be shortened [7] and a substantial amount of energy can be saved [24].

Message-logging protocols require, in principle, to store every single message the application sends. In case of a failure, the messages sent to the crashed node are replayed. Storing messages create an additional memory overhead, something that may become critical in view of the shrinking memory size per node of future architectures [27]. Avoiding excessive memory footprint in message-logging techniques is imperative to leverage all its potential benefits. Past research has focused on general strategies that group nodes and avoid logging messages internal to the groups [21,26]. Those strategies create a tradeoff between memory footprint and recovery cost. However, it is possible to design new message-logging protocols that reduce memory pressure without increasing the recovery cost.

Collective communication operations, or simply *collectives*, are a fundamental building block of parallel applications. Not only are these operations helpful in providing a more expressive construct for a program, but they are useful in improving the scalability and performance of the code as well [28]. Collectives are commonplace in most scientific computing codes. In some cases, they may carry most of the communication traffic. As an example, Fig. 1 shows the breakdown of the total communication volume into three operations: multicast, reduction and point-to-point. Figure 1a presents the split for LeanMD (a molecular dynamics benchmark), where almost all the communication volume is transferred through either multicast or reduction. Figure 1b shows the division for OpenAtom (a quantum chemistry application),



(a) LeanMD **(b)** OpenAtom

Fig. 1 Distribution of communication volume among types of communication operations in two particle-interaction applications

where multicast, reduction, and point-to-point receive approximately, a third of the communication volume each.

This paper aims to provide a general framework to decrease the memory footprint of message logging protocols for collective-communication operations. We introduce CAMEL, a mechanism that only stores the minimum amount of messages necessary to replay a collective during recovery. It does not increase the recovery cost and minimally interferes with a running message-logging protocol.

This paper makes the following contributions:

- An algorithmic description of simple causal message logging (SCMEL), a traditional fault tolerance protocol on which CAMEL is based (Sect. 2).
- A formalization of collective-aware message logging (CAMEL), a method that minimizes the memory overhead of message-logging protocols (Sect. 3).
- The design principles of CAMEL and its implementation in a scalable parallel computing library (Sect. 4).
- An experimental evaluation of CAMEL with two representative applications and on two supercomputing platforms using up to 16,384 cores (Sect. 5).

2 Background

2.1 Related work

Hursey and Graham [15] presented a proposal to build fault-tolerant collective communication operations on MPI. Their mechanism supports algorithm-based fault tolerance (ABFT) based on a resilient implementation of MPI collectives. The programmer of the fault-tolerant algorithm can incorporate the optimized implementation of resilient collectives. A recent proposal in the MPI Forum addressed the need for resilient MPI calls and, among other operations, defined `MPI_Comm_validate_all` that helps a rank to recognize all failures in a communicator. Therefore, the application is aware of the failure in different ranks. Hursey and Graham reviewed three different designs for tree-based collectives. In the first approach, a rerouting technique would check for a failed process before interacting with it and route around crashed processes in a recursive fashion. The second method consists in a lookup-avoiding design that

would remove the check for failures and calculate the relationships parent/child at the end of `MPI_Comm_validate_all`. Finally, the third method uses rebalancing to remove the check for failures and balance the tree for the collective call at the end of `MPI_Comm_validate_all`. The third design provides the best performance and a negligible overhead compared to the fault-unaware implementation. Our approach markedly differs from theirs. The protocol presented in this paper aims to provide an automatic solution for fault tolerance in collectives, without the intervention of the programmer.

Bronevetsky et al. [4] presented a protocol for application-level coordinated checkpoint that targets applications without global synchronization points. In their extension for collective communication operations, the checkpoint infrastructure is based on an algorithm that creates a coordinated checkpoint similar to Chandy-Lamport algorithm [8]. In their protocol, an *initiator* starts the checkpoint process and coordinates the rest of the procedure. This protocol assumes the checkpoint calls are not made in global synchronization points, hence the presence of special measures to log messages and non-deterministic events during checkpoint. Recovery works properly by replaying the necessary messages and reproducing all non-deterministic events. Their algorithm finds two important consistent cuts. The first cut is composed by the collection of local checkpoints of the processes. This cut forms a recovery line to which all processes may roll back in case of a failure. The second cut is composed by the points at which processes stop recording messages and non-deterministic events. The algorithm makes strong claims about the consistency of both cuts. In particular, there must not be any data flow from collectives crossing the second cut. Our approach is based on synchronized checkpoints. Therefore, we removed all the complexity at checkpoint time. In addition, we extend a full message-logging protocol that features the benefits of faster recovery and low energy consumption.

This paper presents a message-logging protocol that minimizes the memory overhead for collective communication operations. A different mechanism, named *team-based message logging* [21,26], creates groups of nodes in the system (called *teams*) and only logs messages crossing team boundaries. That way, if teams are created to match the communication graph of the application, a substantial amount of the total communication volume can be contained internal to the teams and does not need to be stored. That scheme may apply to any implementation of collective communication operations. However, a failure in one node requires the whole team to roll back, increasing the recovery cost. This paper presents a mechanism that decreases memory pressure in collectives and does not increase the recovery cost.

2.2 System model

We define the system on which a parallel application runs as a set $\mathcal{N}$ of processes. Each process stores a portion of the application's data in its own private memory. Message passing is the only mechanism to share information in the system. The channels that connect processes are assumed to respect FIFO ordering. Therefore, messages between same source and same destination can not be reordered.

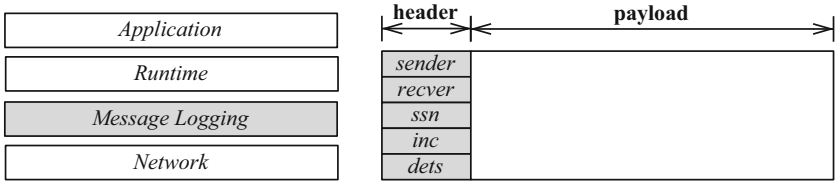
Processes fail according to the *fail-stop* model. After a process fails, it ceases all activity and becomes unreachable. An *incarnation* number is associated with each instance of a process. A failure is represented by a set $\mathcal{F}$ of failed processes. Different fault tolerance protocols offer higher or lower resilience levels according to the maximum size of set $\mathcal{F}$ they can handle. In any case, failed processes are replaced by a new incarnation, keeping constant the size of set $\mathcal{N}$. All processes in $\mathcal{N}$ have been instrumented with a checkpoint function that dumps the state of each process to stable storage.

2.3 Message logging

Rollback-recovery [10] is the most popular mechanism to build a fault-tolerant system in HPC. Checkpoint/restart is the simplest implementation of rollback-recovery. It consists in having all processes in $\mathcal{N}$ periodically storing their checkpoint. The set of all checkpoints form a *restart line* from which the system can recover in case of a failure. As execution progresses and processes take checkpoints, the number of restart lines increases. Not every restart line will bring the system to a consistent state. For instance, if there are in-flight messages crossing a particular restart line, it is not possible to restart from that restart line relying exclusively on a set of checkpoints. Some extra measures are needed to avoid a *cascading rollback* in such situations. If checkpoints are uncoordinated among the processes, in-flight messages could be logged and replayed during recovery. Alternatively, checkpoints can be coordinated among the processes, guarantying a consistent restart line. This last option is an appealing one for most HPC applications that already have global synchronization points. Triggering checkpoints at those global synchronization operations creates a synchronized (and coordinated) valid checkpoint for restart with no in-flight messages. In the rest of this paper, we assume checkpoints are coordinated.

Checkpoint/restart requires a *global* rollback to recover from a failure. That leads to an excessive recovery cost in terms of performance and energy consumption [24]. There are mechanisms that only require a *local* rollback, i.e. rolling back only those processes in $\mathcal{F}$. Message logging is one of those mechanisms. It requires, in principle, to log all the application messages to replay them during recovery. Messages are usually stored in the memory of the sender process [16]. By replaying the messages sent to processes in $\mathcal{F}$, a healthy process avoids to rollback and helps failed processes to recover. To ensure a consistent state is reached after recovery, the recovering processes must deliver all replayed messages in the same order as before the crash. Figure 2a presents the typical location of the message-logging layer in a software stack. Starting from the top, application messages are handled by a runtime system and later tagged by the message-logging protocol. The final message is eventually transmitted through the network. At the receiver side, the message-logging layer captures all network messages and processes them before delivering them to the upper layers.

In addition to storing messages, a message-logging protocol must ensure that recovery is consistent. An inconsistent state may be reached if a process waits for a message that never comes. Such process is usually referred to as an *orphan*. For example, a system with processes X and Y may generate an orphan if Y sends message m to



(a) Layers in the software stack **(b)** Structure of an application message

Fig. 2 Message-logging infrastructure. The software stack has to be extended with an additional layer, and the message header has additional fields

X, Y fails and rolls back, and during recovery Y does not send m to X. In that case, X would be orphan. This is in fact possible, because Y may have sources of non-determinism. Therefore, non-deterministic decisions must be safely stored and provided to the recovering process to ensure a consistent recovery. Message reception is, in general, non-deterministic. In this paper, we will assume message reception is the only source of non-determinism. A *determinant* is a piece of information that records the non-deterministic information from a message reception. For message-logging protocols, the determinant $\#m$ of a message m is usually composed of the tuple $\#m = \langle sender, recver, ssn, rsn \rangle$, where $sender$ and $recver$ are the IDs of the processes exchanging message m. The *sender sequence number* (ssn) corresponds to a unique number that identifies message m. The *receive sequence number* (rsn) represents the order reception of message m. Figure 2b presents the structure of an application message, showing the fields in the header that will be used to build a determinant. More concretely, every application message will carry information about source and destination ($sender$ and $recver$), the message identifier at the source (ssn), the incarnation number of the source (inc), and potentially some determinants ($dets$).

The way in which a protocol handles determinants gives rise to several flavors of message logging [2]. All protocols must guarantee a consistent state is reached after a failure. One traditional family of message-logging protocols is known as the *causal* variant [2]. A causal message-logging protocol ensures determinants are safely stored if they causally affect other events in the system. For instance, upon reception of message m on X, the determinant d, that the reception of m generates, does not need to be stored at that point in time. Instead, it can be piggybacked on outbound message from X and stored at the receivers of those messages. The number of copies of determinant d that have to be stored in the system depends on the reliability of the protocol (i.e. the maximum size of set $\mathcal{F}$ that can be tolerated). One particular group of causal message-logging protocols is known as the Family-Based Protocols [1], or Simple Causal Message Logging [20]. These protocols tolerate single-process failures, i.e. $|\mathcal{F}| = 1$. We will focus the theoretical presentation of the material on this type of protocols.

2.3.1 Simple causal message logging

The most basic version of causal message logging [11] tolerates a single process failure at a time [1] and requires all checkpoints to be globally coordinated [20]. We

Table 1 Data structures used in simple causal message logging (Algorithm 1)

Name	Type	Description
ssn	$\mathbb{N}$	Sender sequence number
rsn	$\mathbb{N}$	Receive sequence number
$rsnMap$	$\mathcal{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Maps a process and an ssn to an rsn
$detLog$	$2^{\mathcal{D}}$	Storage of incoming determinants
$detBuf$	$2^{\mathcal{D}}$	Buffer of outgoing determinants
$msgLog$	$2^{\mathcal{M}}$	Storage of outgoing messages
inc	$\mathbb{N}$	Incarnation number
$incMap$	$\mathcal{N} \rightarrow \mathbb{N}$	Maps a process to its incarnation number

name this protocol *simple* causal message logging. Table 1 shows the fundamental data structures required at each process to carry out all the operations of the protocol. Let us name $\mathcal{M}$ the set of messages sent during the execution of the application and $\mathcal{D}$ the set of determinants generated by those messages.

Each process keeps two scalars to identify messages: ssn uniquely identifies the outgoing messages, while rsn assigns a reception order to the incoming messages. Upon reception of a message m, a process associates the sender of m and its ssn with the corresponding rsn and stores that mapping into $rsnMap$. If the received message comes with piggybacked determinants, they will be stored in $detLog$. The determinant created as the result of a message reception is temporarily stored in $detBuf$ and it will be piggybacked in the next outgoing messages, until an acknowledgment has been received. At that point, the determinant can be removed from $detBuf$. Therefore, *remote* determinants (created at other processes) are stored in $detLog$, while *local* determinants (created by the process itself) are temporarily stored in $detBuf$. Each sent message will be stored in $msgLog$ until the following checkpoint stage, when message logs are emptied. Finally, the current incarnation of a process is kept in inc and it will be incremented every time a process fails. In addition, a process keeps the current incarnation of other processes in $incMap$.

The details of the simple causal message logging protocol, or SCML, are shown in Algorithm 1. The SENDMSG procedure is executed at process X. It catches a message from the runtime layer to the network layer (see Fig. 2a), and fills out the header entries of the message depicted in Fig. 2b. The counter ssn has to be increment with each message submission to make sure each message has a unique identifier. Before the message gets effectively sent through NETSENDMSG, the message is stored into $msgLog$. The RECEIVEMSG procedure presents the more complex case of message reception. The first step is to check for a message that should not be delivered at that point in time. The function OUTOFORDER checks for multiple conditions under which the message should be held in an out-of-order queue or should be discarded. More specifically, an *old* message comes from a process Y has an incarnation number lower than the current incarnation number of Y. A *duplicate* message is a message that has already been received. Both old and duplicate messages have to be suppressed, and they appear in different scenarios during failure and recovery of processes. A message

can also be received out-of-order if it has an rsn already assigned, but the receiving process is behind that number. This may occur during recovery when messages may overrun earlier messages and reach the destination out of order. For those messages, a special queue usually stores and delivers them at the appropriate point in time. If a message is successfully received, it receives a unique rsn, a determinant for its reception is generated and the determinants it carries are stored. An acknowledgment is sent for those determinants. The RECEIVEACKS procedure removes the acknowledged determinants from $detBuf$ and stops its replication through the system.

Algorithm 1 SCMEL: Simple causal message logging protocol

```
1: procedure SendMsg(msg, recver)
2:     msg.sender ← X                                   ▷ Adds sender
3:     msg.recver ← recver                              ▷ Adds receiver
4:     msg.ssn ← Increment(ssn)                         ▷ Updates ssn
5:     msg.inc ← inc
6:     msg.dets ← detBuf                                ▷ Piggybacks determinants
7:     msgLog ← msgLog ∪ {msg}                          ▷ Stores message
8:     NetSendMsg(msg, recver)
9: end procedure
10: procedure ReceiveMsg(msg)
11:     if OutOfOrder(msg) then return                  ▷ Checks for out-of-order
12:     end if                                          ▷ messages
13:     rsnMap(msg.sender, msg.ssn) ← Increment(rsn)    ▷ Updates rsn
14:     detBuf ← detBuf ∪ {⟨msg.sender, X, msg.ssn, rsn⟩}   ▷ Creates determinant
15:     detLog ← detLog ∪ msg.dets                      ▷ Adds remote determinants
16:     NetSendDetAck(msg.dets, msg.sender)
17:     ProcessMsg(msg)
18: end procedure
19: procedure ReceiveDetAck(dets)
20:     detBuf ← detBuf \ dets                          ▷ Removes determinants
21: end procedure
22: procedure Checkpoint( )
23:     detLog ← ∅                                      ▷ Empties remote determinants
24:     detBuf ← ∅                                      ▷ Empties local determinants
25:     msgLog ← ∅                                      ▷ Empties message log
26:     StoreCheckpoint()                               ▷ Creates a restart line
27: end procedure
28: procedure Failure(Y)
29:     dets ← ∅
30:     Increment(incMap(Y))                            ▷ Updates Y's incarnation
31:     for all det ∈ detLog do                         ▷ Collects all determinants
32:         if det.recver = Y then                      ▷ bound to Y
33:             dets ← dets ∪ {det}                     ▷ in detLog
34:         end if
35:     end for
36:     NetSendDets(dets, Y)
37:     Barrier()                                       ▷ Global synchronization
38:     for all msg ∈ msgLog do                         ▷ Collects all messages
39:         if msg.recver = Y then                      ▷ bound to Y
40:             NetSendMsg(msg, Y)                      ▷ in msgLog
41:         end if
42:     end for
43: end procedure
44: procedure ReceiveDets(dets)
45:     for all det ∈ dets do                           ▷ Receives determinants
46:         rsnMap(det.sender, det.ssn) ← det.rsn       ▷ and populates rsnMap
47:     end for
48:     if AllDets() then
49:         Barrier()                                   ▷ Global synchronization
50:     end if
51: end procedure
```

A process failure is handled by various procedures. The CHECKPOINT procedure assumes the checkpoint call is globally coordinated and empties all data structures holding determinants and messages. The checkpoint is assumed to be stored in a safe place, such as the file system or the memory of a different node. When the system detects process Y has failed, it will find a spare node to reinstate process Y using its latest checkpoint. The system will also call procedure FAILURE on all other processes. The call of FAILURE will increase the incarnation number for Y and proceed to send all determinants related to Y and messages bound to Y in two phases. Once all determinants have been received by Y, the messages bound to Y will be sent. Procedure RECEIVEDETS presents the counterpart in the recovery process. Process Y will be receiving determinants from other processes and it will store them into $rsnMap$. Once it has a response from all other processes, it will call the barrier to start the reply of logged messages. The global barrier may be replaced by a more efficient mechanism that separates the determinant collection from the message replay.

2.3.2 Theoretical formulation

To formalize the property a message-logging protocol should enforce, we provide the traditional definition of DEP and LOG sets [2].

Definition 1 (*Depend set*) The set of all processes that reflect the delivery of message m, denoted by DEP(m), contains the destination of message m and any other process that has delivered a message sent causally after the delivery of m.

$$\text{DEP}(m) = \left\{ X \in \mathcal{N} \;\middle|\; \begin{array}{l} (X = m.recver) \wedge deliver_X(m) \vee \\ \exists\, m' : deliver_{m.recver}(m) \rightarrow deliver_X(m') \end{array} \right\}$$

Definition 2 (*Logging set*) The logging set of a message m, denoted by LOG(m), contains all processes that have stored a copy of the determinant of m.

$$\text{LOG}(m) = \left\{ X \in \mathcal{N} \;\middle|\; \begin{array}{l} (X = m.recver) \wedge deliver_X(m) \vee \\ \exists\, m' : \big(deliver_{m.recver}(m) \rightarrow deliver_X(m')\big) \wedge \#m \in m'.dets \end{array} \right\}$$

To ensure a consistent recovery, a message-logging protocol must avoid the creation of orphan processes. We provide the formal definition of an orphan process [2] and specify the non-orphan property.

Definition 3 (*Orphan process*) A process X becomes orphan after the failure of processes in set $\mathcal{F}$ if the following condition holds:

$$(X \in \mathcal{N} \backslash \mathcal{F}) \wedge (\exists\, m : X \in \text{DEP}(m) \wedge \text{LOG}(m) \subset \mathcal{F})$$

Property 1 (No-orphan execution) A consistent recovery produces a no-orphan execution:

$$\forall m \in \mathcal{R} : \text{DEP}(m) \neq \emptyset \implies \text{LOG}(m) \not\subset \mathcal{F}$$

where $\mathcal{R}$ is the set of replayed messages.

Theorem 1 (Correctness of simple causal message logging) *The* SCMEL *protocol detailed in Algorithm 1 complains with Property 1 by creating no orphan process during recovery.*

Proof Let us proceed by contradiction. Negating Property 1 is equivalent to:

$$\exists m \in \mathcal{R} : \text{DEP}(m) \neq \emptyset \wedge \text{LOG}(m) \subset \mathcal{F}$$

which means there is at least a message m in the set of replayed messages and there is at least a process X that depends on that message, but its corresponding determinant is not available. That is, $X \in \text{DEP}(m)$, but $\text{LOG}(m) \subset \mathcal{F}$. Since $|\mathcal{F}| = 1$, let us call Y the only failing process, or $\mathcal{F} = \{Y\}$. By Definifition 3, we have $X \neq Y$. Using Definition 1, we have to consider two cases with respect to X:

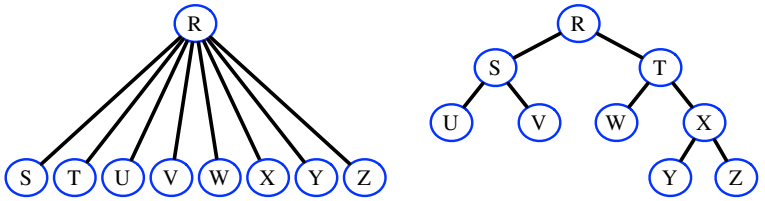
- *Case 1*, $(X = m.recver) \wedge deliver_X(m)$. In this case, X is the original receiver of the message and, by Definition 2, $X \in \text{LOG}(m)$. Therefore, it is not possible to have $\text{LOG}(m) \subset \mathcal{F}$ because X survives the failure of Y. $\Rightarrow\Leftarrow$
- *Case 2*, $\exists\ m' : deliver_{m.recver}(m) \rightarrow deliver_X(m')$. Let us assume $Z = m.recver$. Therefore, $Z \in \text{LOG}(m)$ by Definition 2. If $Z \neq Y$, then $Z \notin \mathcal{F}$ and $\text{LOG}(m) \not\subset \mathcal{F}$. On the other hand, if $Z = Y$ we have to demonstrate that there is at least one process, other than Y, storing a replica of $\#m$. Let us assume there is a message chain of length n connecting the reception of m at Y and the reception of m' at X. We will proceed by induction on n. For the base case, $n = 1$, either Y piggybacked $\#m$ on m' to X, in which case $X \in \text{LOG}(m)$, or Y received the ACK from process W that $\#m$ was safely stored. In either case, there is at least one process that survives the crash and $\text{LOG}(m) \not\subset \mathcal{F}$. For the inductive step, we assume $\text{LOG}(m) \not\subset \mathcal{F}$ is true for $n - 1$. If the message chain is size n, then we are more than certain $\#m$ is safely stored, because the inductive hypothesis tells us that by the time the message chain reached the previous hop, it was size $n - 1$ and $\text{LOG}(m) \not\subset \mathcal{F}$. One more hop in the message chain to reach X does not change that statement. $\Rightarrow\Leftarrow$

□

3 Collective-aware message logging

3.1 Intuitive presentation

The increase of the memory footprint is the major concern of message-logging protocols. Fortunately, collective-communication operations present opportunities to significantly reduce the memory pressure of such protocols. We will focus the discussion on two prevalent operations in large-scale parallel computing: multicast and reduction. Intuitively, a multicast message should not have more than one copy stored in the message logs. Even when a multicast message reaches potentially many processes by virtue of being replicated many times, only one replica in the system is necessary to store to guarantee consistent recovery. A reduction operation involves several messages from different processes. But, once the reduction has been finished, not all the



(a) Serial Implementation **(b)** Tree-based Implementation

Fig. 3 Two strategies to implement collective-communication operations

contributing messages have to be stored. In particular, if messages are agglomerated along the way to the root of the reduction, partial contributions are no longer needed if the last contribution messages are safely stored.

The easiest way to understand the multicast and reduction operations is by considering its *serial* implementation. For instance, a multicast from a source to many destinations is nothing else than a sequence of messages from the source to each destination. Similarly, a reduction from many sources to a common destination is nothing else than a sequence of messages from each source to the same destination. Figure 3a illustrates these ideas with a system having processes R, S, ..., Z and being R the source of a multicast or the destination of a reduction. This implementation of multicast and reduction operations allows the use of any traditional message-logging protocol. That is why popular message-logging implementations in HPC have used this strategy [3,7,13]. One advantage of the serial implementation is that it permits a multicast operation save only one copy of the message at the source. The same can not be said about a reduction operation.

The downside of a serial implementation is its scalability. Very quickly, the source in the multicast or the destination in the reduction becomes a bottleneck and large scaling is not feasible. Figure 3b presents a tree-based implementation for collective operations. This alternative increases scalability and enables parallelism, but presents a challenge for message-logging protocols. A collective operation can run on a spanning tree using regular message-logging protocols, but they can not avoid saving unnecessary copies of certain messages. For example, a multicast on the spanning tree of Fig. 3b would require the same message to be stored four times (at processes R, S, T, and X). The memory-reducing advantage of the serial implementation is lost. A goal of CAMEL, the message-logging protocol introduced in this section, is to use a spanning tree and store a multicast message only *once*. Similarly, a reduction on the spanning tree of Fig. 3b using regular message logging would require the storage of eight messages. However, once the reduction is finished, only two messages are necessary to store (the ones reaching the root of the spanning tree). CAMEL aims to provide a mechanism that only stores the contributing messages reaching the root in a reduction. In a nutshell, CAMEL extends the traditional message-logging protocols and stores the *minimum* number of messages necessary to provide a correct recovery.

In designing CAMEL, it is crucial to understand the different failure scenarios and how collective-communication operations can be reproduced. Let us start with a multicast and a failure that occurs before the multicast message reaches all the destinations.

Imagine a case in Fig. 3b where a multicast message from R has reached all processes but Y and Z. Now, if process W fails, it will not be able to receive the multicast message from T, because multicast messages are not stored at intermediate nodes as in regular message-logging protocols. The original sender of the multicast message, R, will be the only one retaining a copy of the message and it will provide it directly to W. If it is process X the one failing, then the question is how processes Y and Z will receive the multicast message. Once again, during recovery, process R will send the message to X, and X will forward the message to Y and Z. Finally, if R fails during a multicast operation, then such operation must finish. Therefore, processes Y and Z will not drop the multicast message, even if it comes from a sender with an old incarnation number. Multicast operations are thus *atomic*, once started they must be completed.

Similarly, a reduction operation require special considerations. In the same system of Fig. 3b, a reduction with root R is being carried out when a failure happens. Let us assume the contributing messages have arrived at S and T, but not R. If process T fails, then it will not be able to send the final contribution to R unless its children store copies of the contributing messages. Therefore, we will require all processes to temporarily store reduction messages until an acknowledgment confirms that a particular reduction has been finished. The reduction messages from S and T to R must be kept all the time as they are necessary to recover R. If instead of failing T, it is X that fails, the same mechanism will ensure the children of X send the messages again. Upon reception, X will send the contribution to T. This message is a duplicate and can easily be discarded by the standard mechanism of message logging. Finally, to recover any process, the root of every group will provide a *reduction number* that will determine which reduction messages are old and should be discarded.

3.2 Algorithmic description

The formalization of the ideas presented above is called collective-aware message logging (CAMEL). The design of CAMEL is influenced by the functionality of each layer in Fig. 2a. More specifically, the runtime layer interacts with the message-logging layer through a set of functions. The first group of functions is related to the structure of the spanning tree. We assume a group G of processes is formed at the runtime layer through some calls from the application. The runtime layer, however, exposes the spanning tree of any group G to the message-logging layer. In particular, the message-logging layer at process X can query the runtime layer about the parent of X, the children of X, and the root of the spanning tree of group G. This functionality will be represented by a collection of data structures. The second set of functions in the interface between the runtime and the message-logging layer are concerned with the execution of a reduction. When several contributing messages reach a particular process, they will be merged by functions provided by the runtime layer. We make a subtle distinction between *accumulate* messages, and *agglomerate* messages. The former refers to merely collecting the messages, whereas the latter represents the construction of a consolidated message from the individual contributions. Thus, a process that stands at an intermediate node in the spanning tree of a group during a

Table 2 Additional data structures used in CAMEL (Algorithm 2)

Name	Type	Description
$rootMap$	$2^{\mathcal{N}} \rightarrow \mathcal{N}$	Maps a group to its corresponding spanning tree root
$parentMap$	$2^{\mathcal{N}} \times \mathcal{N} \rightarrow \mathcal{N}$	Maps a group and a process to its parent process in that group
$childrenMap$	$2^{\mathcal{N}} \times \mathcal{N} \rightarrow 2^{\mathcal{N}}$	Maps a group and a process to its children in that group
$redSsnMap$	$2^{\mathcal{N}} \rightarrow \mathbb{N}$	Maps a group to a reduction ssn
$redRsnMap$	$2^{\mathcal{N}} \times \mathcal{N} \times \mathbb{N} \rightarrow \mathbb{N}$	Maps a group, a process, and an ssn to a reduction rsn

reduction will accumulate the messages from its children (and itself), and once all messages have been received, it will agglomerate them and forward the consolidated message to its parent.

An extension to the message header in Fig. 2b is necessary to identify the particular group a collective message belongs to. Therefore, a message header will contain a $group$ field that uniquely identifies a set of processes. Similarly, a determinant will include a new field for the group. The reception of a reduction message will be associated with determinant $\langle sender, recver, group, ssn, rsn \rangle$. Note that regular messages do not require a group identifier.

Table 2 presents a list of the data structures in each process that represent part of the interface between the runtime and the message-logging layer. A group G represents a group of processes in $\mathcal{N}$, therefore $G \in 2^{\mathcal{N}}$. Each group G will have an associated spanning tree. The data structures *rootMap*, *parentMap*, and *childrenMap* associate a group with a root process, the parent, and children in the spanning tree, respectively. Each process will hold two additional data structures to keep track of reduction operations. The $redSsnMap$ maintains a sender sequence number for each reduction operation in each group. Therefore, a particular process can determine how many contributions it has made to a particular group. Finally, $redRsnMap$ stands for reduction receive sequence number map and it is used only at the root of a reduction. It associates a reception sequence number with a sender (one of its chidren in the spanning tree for the particular group), a group and a reduction number.

CAMEL extends Algorithm 1 by handling messages of collective-communication operations in a way that minimizes the amount of memory in the message logs. The protocol is presented in Algorithm 2. Auxiliary functions for CAMEL are listed in Algorithm 3. We assume a multicast or a reduction message has a flag that allows the runtime system decide whether to call the regular message methods or the collective-aware ones. For instance, in case of a message emission, the runtime system will check that flag to call either SENDMSG or SENDMULTICASTMSG. The notation $\overline{msg}$ indicates that the identifier of the message is being transmitted, and not the message itself.

Algorithm 2 CAMEL: Collective-aware message logging

```
1: procedure SendMulticastMsg(msg, group)
2:     msg.sender ← X                                          ▷ Adds sender
3:     msg.group ← group                                       ▷ Adds receiver
4:     msg.ssn ← Increment(ssn)                                ▷ Updates ssn
5:     msg.inc ← inc
6:     msg.dets ← detBuf                                       ▷ Adds determinants
7:     msgLog ← msgLog ∪ {msg}                                 ▷ Stores message
8:     SendMsgToChildren(msg)
9: end procedure
10: procedure ReceiveMulticastMsg(msg)
11:     if OutOfOrder(msg) then return                         ▷ Cheks for out-of-order
12:     end if                                                 ▷ messages
13:     rsnMap(msg.sender, msg.ssn) ← Increment(rsn)           ▷ Updates rsnMap
14:     detBuf ← detBuf ∪ {⟨msg.sender, X, msg.ssn, rsn⟩}      ▷ Creates determinant
15:     detLog ← detLog ∪ msg.dets                             ▷ Adds remote determinants
16:     NetSendDetAck(msg.dets, msg.sender)
17:     ProcessMsg(msg)
18:     msg.dets ← ∅                                           ▷ Removes piggybacked determinants
19:     SendMsgToChildren(msg)
20: end procedure
21: procedure SendReductionMsg(msg, group)
22:     msg.sender ← X                                         ▷ Adds sender
23:     msg.group ← group                                      ▷ Adds group
24:     msg.ssn ← Increment(redSsnMap(group))                  ▷ Updates redSsnMap
25:     msg.inc ← inc
26:     VerifyReductionMsg(msg)
27: end procedure
28: procedure ReceiveReductionMsg(msg)
29:     if OutOfOrder(msg) then return                         ▷ Checks for out-of-order
30:     end if                                                 ▷ messages
31:     if X = rootMap(msg.group) then
32:         redRsnMap(msg.sender, msg.group, msg.ssn) ← Increment(rsn)   ▷ Updates redRsnMap
33:         detBuf ← detBuf ∪ {⟨msg.sender, X, group, msg.ssn, rsn⟩}     ▷ Adds determinant
34:     end if
35:     detLog ← detLog ∪ msg.dets                             ▷ Adds remote determinants
36:     NetSendDetAck(msg.dets, msg.sender)
37:     msg.dets ← ∅                                           ▷ Removes piggybacked determinants
38:     VerifyReductionMsg(msg)
39: end procedure
```

Functions SENDMULTICASTMSG and RECEIVEMULTICASTMSG in Algorithm 2 deal with emission and reception of multicast messages. The former is a straightforward extension of the regular message send. The latter shows the creation of a regular determinant, and the forwarding of the message down the spanning tree. However, message copy is avoided at the intermediate nodes of the spanning tree. Once a multicast message arrives at some process, function OUTOFORDER checks whether the message is valid or not. This verification process includes checking whether the message is a duplicate or is old. In the case of multicast messages, because the root relies on the spanning tree to deliver the message, old messages are not discarded.

Function SENDREDUCTIONMSG tags the message with the group ID and the corresponding $rssn$. Then, it calls VERIFYREDUCTIONMSG, which is a generic function that checks whether the reduction message is ready to be propagated up in the spanning tree. Algorithm 3 presents the implementation of VERIFYREDUCTIONMSG. If the reduction message is not ready, it *accumulates* the new contribution and leaves the forwarding for a future call. Otherwise, it *agglomerates* all contributing messages and submits the reduction message. It will temporarily store the reduction message. Once

the reduction has been completed, copies of the reduction messages will be eliminated, except for the direct children of the reduction root. Function RECEIVEREDUCTIONMSG receives a contributing message and generates a determinant only at the root of the spanning tree. It also calls VERIFYREDUCTIONMSG to complete the execution of the reduction.

Algorithm 3 Auxiliary CAMEL functions

```
1: procedure SENDMSGTOCHILDREN(msg)
2:     for all recver ∈ childrenMap(msg.group) do
3:         msg.recver ← recver                                  ▷ Updates receiver
4:         NETSENDMSG(msg, recver)
5:     end for
6: end procedure
7: procedure VERIFYREDUCTIONMSG(msg)
8:     if REDUCTIONREADY(msg) then
9:         msg ← AGGLOMERATE(msg)                               ▷ Consolidates message
10:        msg.sender = X                                       ▷ Updates sender
11:        if X = rootMap(msg.group) then
12:            PROCESSMSG(msg)
13:            SENDMSGACKTOCHILDREN(msg)
14:        else
15:            msg.recver ← parentMap(msg.group)                ▷ Updates receiver
16:            msg.inc ← inc
17:            msg.dets ← detBuf                                ▷ Piggybacks determinants
18:            msgLog ← msgLog ∪ {msg}                          ▷ Stores message
19:            NETSENDMSG(msg, parent)
20:        end if
21:    else
22:        ACCUMULATE(msg)
23:    end if
24: end procedure
25: procedure SENDMSGACKTOCHILDREN(msg)
26:    for all recver ∈ childrenMap(group) do
27:        NETSENDMSGACK(msg, recver)
28:    end for
29: end procedure
30: procedure RECEIVEMSGACK(msg)
31:    group ← msg.group                                        ▷ Updates group
32:    parent ← parentMap(group)
33:    if parent ≠ msg.sender then
34:        msgLog ← msgLog \ msg                                ▷ Removes message
35:    end if
36:    for all recver ∈ childrenMap(group) do
37:        NETSENDMSGACK(msg, recver)
38:    end for
39: end procedure
```

3.3 Formal proof of correctness

We prove the correctness of CAMEL algorithm in two steps. First, we show all required messages are available in the message log of surviving processes. Second, we demonstrate there are no orphan processes after a failure.

Lemma 1 (Availability of replayed messages) *All required messages are available during recovery with* CAMEL *protocol.*

Proof The only messages CAMEL handles differently than traditional message-logging protocols are multicast and reduction messages. Therefore, we will show both those

types of messages are available during recovery. In the first case, a multicast message m received by a failed process X is always available at the root Y of the spanning tree for m. Therefore, a recovering X will have Y replaying the message, regardless of whether Y directly sent a message or not to X before the failure. As for reduction messages, these are temporarily stored until the reduction is finished. If X fails and a reduction is ongoing, the children of X will replay the messages. In any other case, X will not need the contributing messages because it will advance its current reduction sequence number for each group to the latest. If X happens to be the root of a reduction, its children will store the contributing messages. □

Theorem 2 (Correctness of Collective-Aware Message Logging Protocol) *The collective-aware message logging (*CAMEL*) protocol detailed in Algorithm 2 complains with Property 1 by creating no orphan process during recovery.*

Proof We will proceed by contradiction. Let us assume there is an orphan process, in other words:

$$\exists m \in \mathcal{R} : \text{DEP}(m) \neq \emptyset \wedge \text{LOG}(m) \subset \mathcal{F}$$

Message m belongs to one of the three classes of messages: regular, multicast, or reduction. We analyze each case separately.

- *Case 1* (regular message). Theorem 1 guarantees there are not orphan processes for regular messages.
- *Case 2* (multicast message). This case is analogous to the regular case, because other than the storage of the message, multicast messages are handled as regular messages at the receiver.
- *Case 3* (reduction message). To have $X \in \text{DEP}(m)$, the whole reduction has to be finished. Therefore, $X \in \text{DEP}(\hat{m})$, where $\hat{m}$ is one message reaching the root of the reduction tree. Intermediate messages are irrelevant for consistency of recovery. Again, Theorem 1 ensures $\hat{m}$ gets its associated determinant. $\Rightarrow\Leftarrow$

□

4 Implementation

The CHARM++ parallel programming runtime system [18] was chosen for CAMEL's implementation. There are several features of the CHARM++ execution model that make it a general framework for message-passing programs. The CHARM++ philosophy encourages modularity and locality by having *objects* that carry out the execution of the program. Each object, called a *chare* in CHARM++'s terminology is an independent execution unit that exports remote methods. Each method invocation is transformed into an active message that triggers a particular method upon reception. Thus, the collection of objects in a CHARM++ program performs a computation by exchanging messages in an asynchronous fashion, providing what is called message-driven execution. This mechanism is more general than the one enforced by the message-passing interface (MPI). The CHARM++ runtime system conceives the underlying machine as

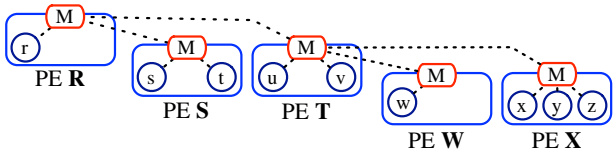


Fig. 4 Architecture of CAMEL implementation in CHARM++

a collection of *processing entities* (PEs). The unit of failure in the CHARM++ model is one PE. Therefore, we will consider in the rest of the paper the loss of one PE as the representative example of a failure in HPC systems. The set of objects in a computation is divided among the set of PEs by the runtime system. Because multiple objects may reside on the same PE, it is said that CHARM++ provides *overdecomposition*. The runtime system also handles migration of objects between different PEs to achieve load balance, fault tolerance, and power management.

Figure 4 presents a general view of the architecture in CHARM++ that implements the CAMEL protocol. The processing entities take the place of the processes described in Sect. 2. Therefore, the failure of one PE implies multiple objects are lost simultaneously. Figure 4 also shows two spanning trees: one for the set of PEs $\{R, S, T, W, X\}$ rooted at R, one for the group of objects $\{r, s, t, \ldots, z\}$ rooted at r. This figure replaces the view of the spanning tree between the group of processes in Fig. 3b. In CHARM++, both PEs and objects have spanning trees, although they are interdependent. A manager object (tagged with a letter M in the diagram) represents a special kind of object that manages collective operations among the set of objects and defines the spanning tree. All the information relative to spanning trees for collectives is maintained in this set of objects. In addition, the manager is in charge of sending and receiving all collective communication messages. Because of its strategic role, the manager object may interact with the runtime system layer (see Fig. 2a) to perform all necessary operations in Algorithm 2, in particular functions ACCUMULATE and AGGLOMERATE.

Seemingly, the two major challenges of the implementation of CAMEL in CHARM++ are the non-FIFO channels and the asynchrony of execution. If channels do not conserve FIFO ordering, then Algorithms 1 and 2 will still work because reception order is stored and reproduced during recovery. However, the programmer has to be aware of this property and design the application accordingly. By the same token, asynchronous operations do not prevent the algorithms to be correct, but create a restriction on the mind of the programmer as to what type of assumptions can be made.

5 Experimental evaluation

5.1 Setup

The implementation of CAMEL protocol presented in Sect. 4 was deployed on two different supercomputers, Intrepid and Stampede. *Intrepid* is housed by the Argonne Leadership Computing Facility (ALCF) in Argonne National Laboratory (ANL). It is an IBM Blue Gene/P machines with 40,960 nodes. Each node consists of one quad-

core 850 MHz PowerPC 450 processor and 2 GB DDR2 of main memory. Intrepid has a total of 163,840 cores, 80 terabytes of RAM, and a peak performance of 557 TeraFLOPs. *Stampede* is hosted at Texas Advanced Computing Center (TACC). It has 6,400 nodes, with each node containing 2 Intel Xeon processors (16 cores total) and 32GB of memory. Stampede's interconnect uses Mellanox FDR Infiniband technology in a two-level fat-tree topology. With a total of 96,000 cores, it can deliver more than 10 PetaFLOPs.

A couple of CHARM++ applications were selected to experimentally evaluate the implementation of CAMEL. These applications were introduced in Fig. 1. *LeanMD* is a mini-application that emulates the communication pattern in NAMD. The major goal of LeanMD is to compute the interaction forces between particles in a three-dimensional space based on the Lennard-Jones potential. It does not include long-range force computation. The object decomposition divides a three-dimensional space into hyperrectangles. Each of these boxes, called cells, contains a subset of the particles. A specific object, called a compute, is connected to the two cells and receives the particles from both cells to perform the particle interaction computation. In each iteration of LeanMD, each cell sends its particles to all the computes attached to it and receives the updates from those computes. *OpenAtom* is a parallel application for molecular dynamics simulations based on fundamental quantum mechanics principles. Car-Parrinello ab-initio molecular dynamics (CPAIMD) is a well-known approach that has proven to be efficient and useful in this type of simulations. The parallelization of this approach beyond a few thousand processors is challenging, due to the complex dependencies among various subcomputations. This may lead to complex communication optimization and load balancing problems. OpenAtom implements CPAIMD in CHARM++.

5.2 Results

The main goal of CAMEL is to reduce the memory footprint of the message log in message-logging protocols. We measured the size and type of all messages logged during executions of both LeanMD and OpenAtom and present the breakdown of the message log. Figure 5 presents the message log memory consumption of LeanMD on Stampede using the traditional SCMEL protocol (listed in Algorithm 1) and CAMEL (listed in Algorithm 2). The former is represented by letter *S* in the figure, while the latter by letter *C*. Figure 5a shows a small problem size, whereas Fig. 5b shows a moderate problem size. In both cases, CAMEL manages to dramatically decrease the memory pressure. CAMEL reduces the size of the memory log by at least 95 %. Both figures also show the strong scale of CAMEL from 1K cores up to 16K cores. This dramatic reduction comes from the fact that LeanMD transfers most of the data through collective operations (as shown in Fig. 1a). In fact, less than 1 % of the data is sent via point-to-point operations. LeanMD exchange particle information between *cells* and *computes* using multicast and reductions. We should note that CAMEL is more effective reducing the memory footprint of multicast. That is reflected in Fig. 5a,b. In general, CAMEL decreases multicast memory pressure from 50 to 1 %, whereas reduction memory pressure goes down from 50 to 3 %. This is a natural effect of having

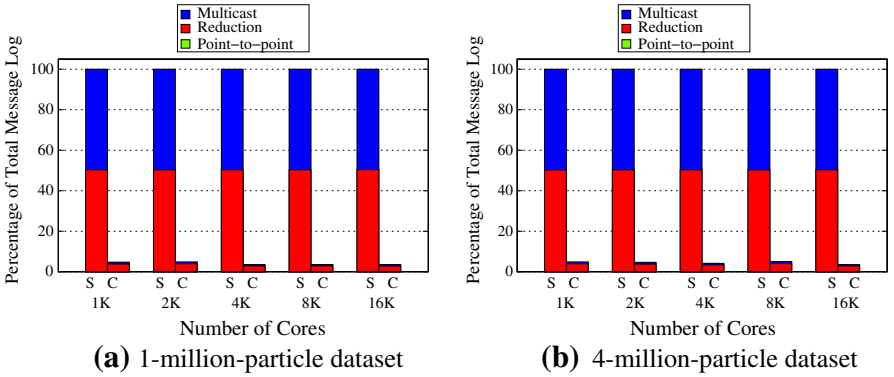


Fig. 5 Relative message log size of SCMEL (S) and CAMEL (C) in LeanMD

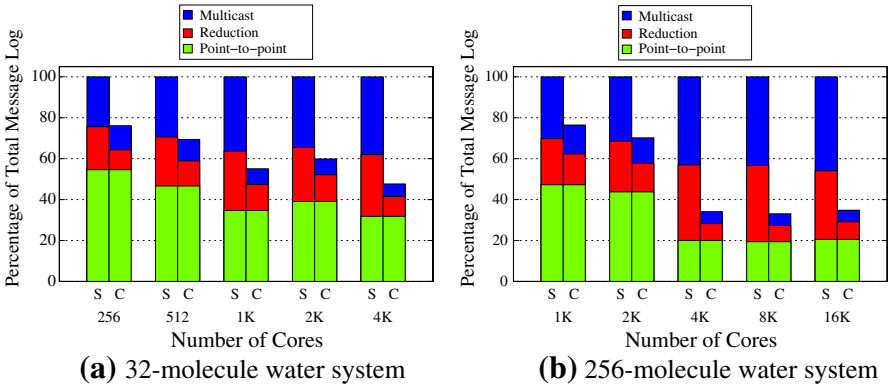


Fig. 6 Relative message log size of SCMEL (S) and CAMEL (C) in OpenAtom

to store only one copy of the multicast message, but as many reduction messages as the root of the spanning tree receives (see Algorithm 2).

Figure 6 shows the OpenAtom results on Intrepid. Two problem sizes, small and moderate, were used and appear in Fig. 6a, b, respectively. Again, CAMEL shows a significant reduction of the message log. In Fig. 6 the total memory required by the message log gets reduced from 24 to 5 % as we move from 256 to 4K cores. The fraction of messages that belong to point-to-point operations remains unchanged because CAMEL only addresses collective messages. That fraction is significant in OpenAtom, taking from 55 to 32 %. Again, CAMEL has a higher impact on multicast messages than in reduction messages. For instance, at 4K in Fig. 6, the fraction of the message log related to reduction messages goes from 30 to 10 %, whereas its multicast counterpart goes from 38 to 6 %. Figure 6b shows similar results for a bigger problem size in OpenAtom. The decrease on the message log for this case goes from 24 to 65 % as we strong scale the program from 1K to 16K cores. Figure 6a, b share the same trend to decrease the message log used by CAMEL as the program strong scales. The reason for such pattern is that OpenAtom uses more heavily collective communication operations as more cores are available. That can be verified by the decreasing fraction

Table 3 Message and determinant statistics for LeanMD

	Number of cores				
	1024	2048	4096	8192	16,384
Messages ($\times 10^6$)	24.32	23.7	32.33	32.05	33.09
SCMEL					
Message log (GB)	269.49	261.19	363.44	359.59	370.82
Determinants ($\times 10^6$)	54.72	54.04	62.68	62.4	63.44
Piggybacked ($\times 10^6$)	348.23	265.33	275.84	221.68	185.39
CAMEL					
Message log (GB)	12.77	12.77	12.79	12.83	12.94
Determinants ($\times 10^6$)	28.94	28.61	32.99	32.87	33.44
Determinants (%)	52.88	52.94	52.63	52.68	52.71
Piggybacked ($\times 10^6$)	185.48	143.52	136.62	129.19	111.51
Piggybacked (%)	53.26	54.09	49.53	58.28	60.15

of point-to-point messages in the spectrum. Many HPC applications apply the same logic of relying more heavily on collective operations as the scale gets larger. Finally, a difference in the magnitude of the impact of CAMEL on LeandMD and OpenAtom relates to the *depth* of the spanning tree. LeanMD uses spanning trees that reach more cores and hence CAMEL manages to get a more dramatic decrease in the size of the message log.

An additional benefit of CAMEL is the elimination of certain determinants for reduction operations and the avoidance of piggybacking certain determinants for multicast operations. Algorithm 2 shows in function RECEIVEREDUCTIONMSG how a reduction message generates determinants only at the root of the spanning tree. Therefore, compared with SCMEL, CAMEL does not create determinants for reception of intermediate contribution messages to a reduction. In the case of multicast, Algorithm 2 shows in function RECEIVEMULTICASTMSG how piggybacked determinants are removed from the original multicast messages before being forwarded to the children by function SENDMSGTOCHILDREN. Table 3 presents relevant communication statistics for LeanMD running 100 iterations on the one-million particle dataset. The test was run on Stampede for the range between 1024 and 16,384 cores. The first row in Table 3 reports the total number of messages sent in the system during the whole execution. Although the table presents a strong scaling test, the number of total messages increases with the system size due to a different and deeper structure in the spanning tree. The same holds true for the size of the message log. Table 3 compares SCMEL and CAMEL for the size of the message log (the proportion appears in Fig. 5a), the total number of determinants created and the total number of determinants piggybacked. Again, the numbers correspond to the sum of all determinants in all the processes for the whole execution. Besides the dramatic reduction in message log size, CAMEL is also able to approximately reduce in half the number of determinants created and the number of determinants piggybacked. Eliminating determinants (and their distribution) can alleviate the main source of performance penalization of message-logging protocols [17].

It does, however, depend on the type of interconnection. The performance difference on Stampede was not significant.

6 Analysis

Traditional message-logging protocols can cope with messages from collective-communication operations [3,7,13], but fail to leverage the structure of those operations to decrease the memory size of the message log. CAMEL capitalizes on the implementation of collective operations on a spanning tree and only saves the minimum amount of data needed to reproduce the result of such operation after a failure. CAMEL is meant to *extend* any traditional message-logging protocol by adding extra few bits of information and handling collective messages slightly differently. Although this paper uses causal message logging as an example, CAMEL philosophy is applicable to other protocols.

CAMEL does not require the generation of determinants for reduction messages, except at the root of the spanning tree. Determinism is not needed when agglomerating messages at the intermediate nodes because replicating the same deterministic decisions will not have any impact on the correctness of the execution. More specifically, if process X is an intermediate node in a spanning tree and agglomerates reduction messages in a non-deterministic way, the exact same decision does not need to be made during recovery because the agglomerated message emitted by X will be discarded. If functions ACCUMULATE and AGGLOMERATE are deterministic, then determinant generation can be removed from the root node in Algorithm 2.

The failure unit assumed in this paper is a process (or a PE in the CHARM++ implementation). A more realistic assumption, given the failure pattern of modern supercomputers [22], is to assume a node as the unit of failure. It is straightforward to extend CAMEL's ideas into a multicore node-based runtime system. At the node level, CAMEL would work exactly the same way as in the PE-based case. For instance, the SCMEL protocol in Algorithm 1 has been extended to multicore node systems [22] and CAMEL could be adjusted to such environment.

The design of CAMEL includes a tight collaboration with the runtime system. Figure 2a shows this interaction in which certain operations in CAMEL call the runtime layer for accumulation and agglomeration of messages. One additional function at the runtime layer might be to provide message buffering in case of concurrent collective operations.

One of the main features of CAMEL is its appeal for scalability. The results in Sect. 5 demonstrated the benefits of CAMEL in both strong and weak scaling. As more processes are used to run collective-intensive programs, the spanning trees get deeper and there is a bigger impact on saving messages near the root that would otherwise have been stored throughout the whole tree.

There are several ways in which CAMEL can be optimized. First, during recovery multicast messages are always propagated, regardless whether the multicast has actually reached all nodes. Duplicate multicast messages can be suppressed by having acknowledgments from children. Therefore, parents in the spanning tree would not propagate a multicast message if the operation has been confirmed by its chil-

dren. Second, other structures for collectives can be used. In some cases, processes would delegate collective messages even if they are not part of the group involved in the collective. Third, it is possible to merge CAMEL with other memory-reducing techniques for message logging. For instance, team-based message logging [21,26] groups processes into teams and avoids storing communication within teams. Team-based message logging can be integrated with CAMEL.

7 Conclusions and future work

The effective usage of future supercomputers will depend on the ability of the system to overcome the high rate of failures predicted for such large systems. Rollback-recovery strategies have been widely adopted in the HPC community. One of those strategies is message logging, which stores communication to avoid a global rollback. That feature makes message-logging protocols able to reduce execution time and energy consumption on a faulty machine. However, a drawback of message logging is the increased memory pressure due to the message log.

This paper introduces CAMEL, a collective-aware protocol to reduce the size of the message log in memory. CAMEL extends traditional protocols by adding a few bits of extra information and only stores messages that are absolutely necessary to reproduce multicast and reduction operations during recovery. Results on two different platforms and using two different applications demonstrate CAMEL's ability in substantially reduce the memory pressure of message logging.

Other types of collective communication operations can be incorporated into CAMEL's model. We will explore those operations in the future along with other applications that make a significant use of collective operations. In addition, we will design a holistic approach that combines CAMEL with other memory-reducing techniques for message logging.

Acknowledgments This research was supported in part by the US Department of Energy under Grant DOE DE-SC0001845 and by a machine allocation on the Teragrid under award ASC050039N. This work also used machine resources from PARTS project and Directors discretionary allocation on Intrepid at ANL for which authors thank the ALCF and ANL staff.

References

1. Alvisi L, Hoppe B, Marzullo K (1993) Nonblocking and orphan-free message logging protocols. In: FTCS, pp 145–154
2. Alvisi L, Marzullo K (1995) Message logging: pessimistic, optimistic, and causal. International conference on distributed computing systems, pp 229–236
3. Bouteiller A, Bosilca G, Dongarra J (2010) Redesigning the message logging model for high performance. Concurr Comput Pract Exp 22(16):2196–2211
4. Bronevetsky G, Marques D, Pingali K, Stodghill P (2003) Collective operations in application-level fault-tolerant MPI. In: Proceedings of the 17th annual international conference on supercomputing, ICS '03ACM, New York, NY, USA, pp 234–243
5. Cappello F (2009) Fault tolerance in petascale/ exascale systems: current knowledge, challenges and research opportunities. IJHPCA 23(3):212–226
6. Cappello F, Geist A, Gropp B, Kale L, Kramer B, Snir M (2009) Toward exascale resilience. Int J High Perform Comput Appl 23(4):374–388. doi:10.1177/1094342009347767

7. Chakravorty S, Kale LV (2007) A fault tolerance protocol with fast fault recovery. In: Proceedings of the 21st IEEE international parallel and distributed processing symposium. IEEE Press
8. Chandy KM, Lamport L (1985) Distributed snapshots : determining global states of distributed systems. ACM transactions on computer systems
9. Elnozahy EN, Bianchini R, El-Ghazawi T, Fox A, Godfrey F, Hoisie A, McKinley K, Melhem R, Plank JS, Ranganathan P, Simons J (2008) System resilience at extreme scale. Defense Advanced Research Project Agency (DARPA), Tech. Rep
10. Elnozahy EN, Alvisi L, Wang YM, Johnson DB (2002) A survey of rollback-recovery protocols in message-passing systems. ACM Comput Surv 34(3):375–408
11. Elnozahy EN, Zwaenepoel W (1992) Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. IEEE Trans Comput 41(5):526–531. doi:10.1109/12.142678
12. Ferreira K, Stearley J, Laros III JH, Oldfield R, Pedretti K, Brightwell R, Riesen R, Bridges PG, Arnold D (2011) Evaluating the viability of process replication reliability for exascale systems. In: Supercomputing, ACM, New York, pp 44:1–44:12
13. Guermouche A, Ropars T, Brunet E, Snir M, Cappello F(2011) Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. In: IPDPS, pp 989–1000
14. Hargrove PH, Duell JC (2006) Berkeley lab checkpoint/restart (BLCR) for linux clusters. In: SciDAC
15. Hursey J, Graham RL (2011) Preserving collective performance across process failure for a fault tolerant MPI. In: Proceedings of the 2011 IEEE international symposium on parallel and distributed processing workshops and PhD forum., IPDPSW '11IEEE Computer Society, Washington, DC, USA, pp 1208–1215
16. Johnson DB, Zwaenepoel W (1987) Sender-based message logging. In: In digest of papers: 17 annual international symposium on fault-tolerant computing, IEEE Computer Society, pp 14–19
17. Jonathan Lifflander EM, Menon H, Miller P, Krishnamoorthy S, Kale L (2014) Scalable replay with partial-order dependencies for message-logging fault tolerance. In: Proceedings of IEEE Cluster 2014. Madrid, Spain
18. Kalé L, Krishnan S (1993) Charm++ : a portable concurrent object oriented system based on C++. In: Proceedings of the conference on object oriented programming systems, languages and applications
19. Kogge P, Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hiller J, Karp S, Keckler S, Klein D, Lucas R, Richards M, Scarpelli A, Scott S, Snavely A, Sterling T, Williams RS, Yelick K (2008) Exascale computing study: technology challenges in achieving exascale systems
20. Meneses E, Bronevetsky G, Kale LV (2011) Evaluation of simple causal message logging for large-scale fault tolerant HPC systems. In: 16th IEEE workshop on dependable parallel, distributed and network-centric systems in 25th IEEE international parallel and distributed processing symposium (IPDPS 2011)
21. Meneses E, Mendes CL, Kale LV (2010) Team-based message logging: preliminary results. In: 3rd workshop on resiliency in high performance computing (Resilience) in clusters, clouds, and grids (CCGRID 2010)
22. Meneses E, Ni X, Kale LV (2011) Design and analysis of a message logging protocol for fault tolerant multicore systems. Tech. Rep. 11–30, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign
23. Meneses E, Ni X, Zheng G, Mendes CL, Kale LV (2014) Using migratable objects to enhance fault tolerance schemes in supercomputers. In: IEEE transactions on parallel and distributed systems
24. Meneses E, Sarood O, Kale LV (2014) Energy profile of rollback-recovery strategies in high performance computing. Parallel Computing 40(9), 536–547 (2014). doi:10.1016/j.parco.2014.03.005. http://www.sciencedirect.com/science/article/pii/S0167819114000350
25. Moody A, Bronevetsky G, Mohror K, de Supinski BR (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC, pp 1–11
26. Ropars T, Guermouche A, Uçar B, Meneses E, Kalé LV, Cappello F (2011) On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. Euro-Par 1:567–578
27. Snir M, Gropp W, Kogge P (2011) Exascale research: preparing for the post moore era. https://www.ideals.illinois.edu/bitstream/handle/2142/25468/Exascale%20Research.pdf
28. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of collective communication operations in mpich. Int J High Perform Comput Appl 19(1), 49–66 (Spring 2005). doi:10.1177/1094342005051521
29. Zheng G, Shi L, Kalé LV (2004) FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: 2004 IEEE Cluster, San Diego, CA, pp 93–103