Reducing the Overhead of Message Logging in Fault-Tolerant HPC Applications

Esteban Meneses

National Advanced Computing Collaboratory, National High Technology Center
and School of Computing, Costa Rica Institute of Technology
Costa Rica
esteban.meneses@acm.org

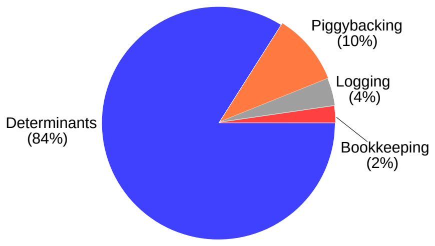
Abstract. With the exascale era within reach, the high performance computing community is preparing to embrace the challenges associated with extreme-scale systems. Resilience raises as one of the major hurdles in making those systems usable for the advance of science and industry. Message logging is a well-known strategy to provide fault tolerance, one that is promising due to its ability to avoid global restart. However, message-logging protocols may suffer considerable overhead if implemented for the general case. This paper introduces a new message-logging protocol that leverages the benefits of a flexible parallel programming paradigm. We evaluate the protocol using a particular type of applications and demonstrate it can keep a low performance penalization when scaling up to 128,000 cores.

Keywords: resilience, fault tolerance, message logging.

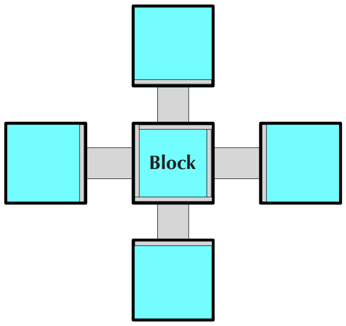
1 Introduction

The imminent arrival of extreme-scale supercomputers sometime next decade will provide scientists and engineers with the right tool to accelerate fundamental discoveries in the grand challenges of several disciplines. From cosmological simulations of the origin of the universe to drug design for personalized medicine, our understanding of nature heavily depends on the ability to efficiently exploit the massive computational power of those machines. However, with an exascale machine already in sight, reliability stands out as one of the major obstacles in reaching a productive supercomputing system [2, 10]. Considering the gigantic number of components that must be assembled into extreme-scale systems, failure rate will undoubtedly increase. Addressing the resilience challenge is crucial to the advance of science and industry.

The High Performance Computing (HPC) community has traditionally relied on rollback-recovery strategies to provide fault tolerance for large-scale simulations. One technique that has recently gained some attention is message logging, which seems promising given its lower energy consumption in a faulty environment [9]. Message logging, however, incurs some performance overhead due to



(a) Contribution of each factor to message-logging performance penalty.



(b) Communication structure in LULESH.

Fig. 1. Performance cost of determinants in LULESH. The main source of the performance penalty of message logging with LULESH comes from determinants. The communication structure of LULESH, however, suggests there is a way to avoid determinants altogether.

the additional mechanisms that ensure a correct recovery. One of those mechanisms is called *determinants*, which are in charge of storing all information about non-deterministic events.

Determinants play a major role in the performance penalization of message logging. Figure 1(a) presents the breakdown of that overhead in an experiment. Using $1,024$ cores on Stampede supercomputer at Texas Advanced Computing Center (TACC), and the LULESH benchmark, we examine the four sources of performance loss for a traditional message-logging protocol. *Bookkeeping*, a mechanism to avoid duplicate messages, contribute with little over 2%. *Logging*, or storing outgoing messages in main memory, represents about 4%. *Piggybacking*, or the overhead of adding additional information, contributes with 10%. Finally, *Determinants*, or the cost of creating, storing, sending, and acknowledging determinants, is the dominant contributor of performance overhead with 84%. On the other hand, Figure 1(b) shows why that overhead can be dramatically reduced. The program LULESH comes from a big family of codes collectively referred to as *stencils*. These programs work on a multidimensional grid by applying an operation to each element. The usual data partition algorithm splits the grid into *blocks*. In Figure 1(b) we observe a two-dimensional grid and a block with all its neighbors. As the computation progresses the blocks exchange the border elements with each neighbor in every iteration before proceeding to apply the transformation of its own elements of the grid. High-level script languages provide a mechanism to express this type of computation explicitly, but more importantly, in a very simple way. Therefore, using that information, new message-logging protocols may take advantage of simple and deterministic communication patterns to avoid creating unnecessary determinants.

The focal points of this paper are:

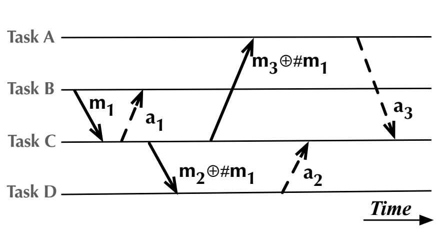
– A description of a particular kind of programs and high level programming language constructs that exposes the determinism in communication (§ 3).
– The design of *fast message-logging*, a protocol that removes all determinants in this type of programs (§ 4).
– A comparative evaluation between simple causal message-logging and fast message-logging is offered (§ 5).

2 Background

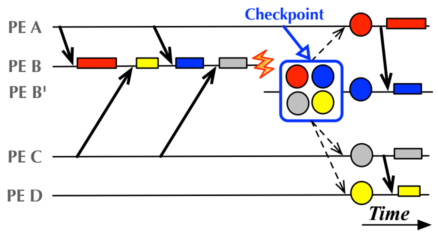
Rollback-recovery is the most popular strategy to build resilient applications in HPC. In the usual form, called checkpoint/restart, a program starts execution and periodically stores its global state. If a failure strikes the system (failures are assumed to be permanent), the application rolls back to the previous checkpoint and resumes from that point. If messages are also stored, then the failure of one component only requires the rollback of the tasks running on that component. Messages can be replayed to the recovering element until it reaches a consistent state with the rest of the system. Therefore, in most of HPC implementations, message logging requires checkpoints to be taken at global synchronization points and messages to be stored at the sender side.

There are several variants of message logging [3], which differ in the way they handle *determinants*. Since message logging provides local rollback (i.e. only the failed tasks must roll back to the previous checkpoint), certain non-deterministic decisions must be stored to provide a consistent recovery. Determinants represent those decisions and protocols become piece-wise deterministic. Figure 2(a) shows an example of a protocol called simple causal message-logging [7]. Assume an application is composed by several tasks, from A to D. Message reception is, in general, non deterministic. Therefore, every message reception will generate a determinant. For instance, message m_1 from B to C generates determinant $\#m_1$. That determinant stores the order in which message m_1 was received at C and it will be necessary for recovery as long as other events in the system causally depend on that. Therefore, determinant $\#m_1$ will be piggybacked on outgoing messages from C until an acknowledgment a_i confirms the determinant has been safely stored somewhere else. The simple causal message logging protocol tolerates a single task failure but works in the general case. More advanced message-logging protocols bank on certain properties of HPC applications to reduce the number of determinants generated. For instance, a program may show *send determinism* [4] if the sequence of send events is always the same for every valid execution. Such program may use a protocol that avoids creating some determinants and thus reduces the overhead of the protocol.

In the parallel-objects programming model, an application is decomposed into objects, also called *chares*. Objects export a list of *entry* methods that other objects may call remotely through messages. Therefore, execution is always message driven. Each message contains the object recipient and the name of the method to be invoked, much like Active Messages [11]. The system is



(a) Simple causal message-logging.



(b) Parallel recovery.

Fig. 2. Message logging leverages the overdecomposition property of the parallel-objects programming model.

represented by a set of *processing entities* (PE) interconnected through a fast network that does not guarantee FIFO ordering. The number of objects usually exceeds the number of PEs in the system and the environment is said to be *overdecomposed*. The ratio between objects and PEs is called *virtualization ratio*. Since the location of an object is irrelevant to the programmer, a smart runtime system may decide to migrate objects between PEs. Some goals of migration include load balancing, proactive fault tolerance, and reduction of power consumption. The parallel-objects model has been implemented into Charm++ [5], a C++ language extension. It is precisely overdecomposition what gives parallel objects an edge on message logging. As shown in Figure 2(b), if PE B holds 4 objects (represented by the colored circles), the recovery of PE B (replaced by B') can be done concurrently. This is called *parallel recovery* and entails migrating objects in the recovering PE to other PEs during recovery.

In this paper we explore a scripting language that makes the control flow explicit. This is a language extension to Charm++ that provides certain structure and tame the relative flexibility in which messages can be processed at their destination in the parallel-objects model. These extra constructs in the language are referred as SDAG or *structured dagger*. Using programs written in SDAG, we aim to reduce the number of determinants generated in the program and reduce the total execution-time overhead of message-logging.

3 Removing Determinants in Parallel Programs

We assume the underlying machine is formed by a set Σ of PEs. We assume the network is reliable but does not ensure FIFO ordering in the delivery of messages between any pair of PEs. The application is decomposed into a set Γ of objects. We assume there is no shared memory in the program and the only mechanism to exchange information is via message passing. In particular, the objects of the application are *reactive* and execute a method upon the reception of a message. This asynchronous mechanism provides a message-driven execution of the program. All message sends are asynchronous, including the contributions to reductions.

In the parallel-objects model, each message has a *type*, determined by the particular method that gets triggered when it is received. Thus, it is possible to distinguish between two messages based on their type. Furthermore, messages of the same type can also be differentiated by the particular combination of sender object and receiver object. Finally, each message may carry a *tag* (called *reference number* in SDAG) that is used to decide whether to process a message at reception. The tag can be used to further separate two otherwise identical messages. The combination of type, sender, receiver, and tag is an essential component of a message-logging infrastructure.

The parallel-objects model is able to express a wide variety of parallel programs. In particular, it is very well suited for a class of programs named as *stencil codes*. These programs are iterative kernels that update elements on a grid in a very structured way. Partial differential equations (PDEs) are a good source for this type of codes. Usually, to numerically solve a PDE, a discretized version of the space is required. The space can thus be represented by a grid or mesh. Other types of stencil codes are used in image processing, computational fluid dynamics, and other kinds of scientific simulations. The nature of stencil codes consists of applying an operation to all the points in the grid until some convergence criteria is met or after certain number of iterations have been completed. In a parallel implementation of this type of codes, the whole grid is split into *blocks*. Each processing element holds a subset of the blocks. The computation proceeds by iteratively exchanging some values between neighboring blocks, applying an operation over the elements in the blocks and following to the next step. Most of the time, the values exchanged between blocks correspond to the neighboring elements of each block. The communication is very structured and does not change during the execution. The name *stencil* actually comes from the fixed pattern in which message exchange takes place. A program formulated in the parallel-objects model is shown in Figure 3(a).

Two types of objects are shown: `Main` and `Block`. The former orchestrates the execution of the total number of iterations in the program. It initially broadcasts the message `start` to all blocks (line 5). The latter represents each block and handles exchanging messages between all the blocks. Each iteration starts by sending the ghost elements to the neighbors (line 15). Then, each block must keep track of how many ghost-element messages it has received. Once the counter has reached the total number of neighbors, the block proceeds to apply the computation to its elements. At the end of every iteration (line 23), all the set of blocks contribute to a reduction that returns the control to `Main`. The cycle starts again until the total number of iterations is completed. Note that the reduction after each iteration is inevitable in this model. Since the channels are not FIFO in delivering the messages, then it is possible for a couple of messages between neighboring blocks to go out-of-order. Therefore, a synchronization must be used to avoid such scenario. The reduction effectively separates messages from consecutive iterations.

The very nature of the parallel-objects model offers a lot of flexibility in the way the parallel program executes. It does, however, introduce concerns

```
1  class Main{
2   Collection<Block> blocks;
3   method main(){
4    iter = 0;
5    blocks.start();
6   }
7   method reduction(){
8    iter++;
9    if(iter <= TOTAL)
10     blocks.start();
11  }
12 }
13 class Block{
14  method start(){
15   sendGhostElems();
16  }
17  method ghostElems(msg){
18   copyElems(msg);
19   counter++;
20   if(counter == neighbors){
21    counter = 0;
22    compute();
23    contribute(Main::reduction);
24   }
25  }
26 }
```

(a) Two-dimensional stencil code.

```
1  class Main{
2   Collection<Block> blocks;
3   method main(){
4    blocks.start();
5   }
6  }
7  class Block{
8   method start(){
9    for(iter=1; iter<=TOTAL; iter++){
10    sendGhostElems();
11    overlap[neighbors]{
12     when ghostElems[iter](msg)
13      copyElems(msg);
14    }
15    compute();
16   }
17  }
18 }
```

(b) High-level script for 3(a).

Fig. 3. Program structure of a two-dimensional stencil code. Each block exchanges the ghost elements with its neighbors in each step. There is an asynchronous barrier after each iteration in 3(a). Its high-level script counterpart in 3(b) removes the synchronization reduction after each iteration, and handles message reordering by tagging each message with an iteration number.

regarding the order in which messages are going to be received. The `counter` in Figure 3(a) reflects the palliative effort in the program to allow concurrency but avoid erroneous receptions. The same effect can be achieved through a high-level script language that expresses the control dependencies between certain types of events. Additionally, this script language must retain the advantages of the reactive characteristic of objects in the program. To illustrate the way a high-level description would work with the same example as in Figure 3(a), we show the new version of the code in Figure 3(b).

The high-level script in Figure 3(b) features various grammatical constructs that alleviate the issues a programmer faces when writing code in the parallel-objects model. The main difference between the two versions of the code resides in the fact that the control mechanism is moved away from the `Main` object. The only responsibility of the `Main` object is to spawn the start of the program (line 4). The number of iterations is controlled by the `Block` objects (line 9). Each iteration will consist in sending the ghost elements to the neighboring blocks, followed by the reception of messages and the computation code. The first grammatical construction is the `overlap` region that allows the execution of its enclosing statements in any order. If the statements are all the same, it may be specified as a parameter. For instance, the `overlap` construct in Figure 3(b) (line 11) has `neighbors` as a parameter, meaning it allows the reception of the `ghostElems` messages in any order. This clearly expresses the structure of the code, that requires the ghost data from all the neighbors before performing the computation. The second grammatical construct is the `when` statement that specifies a message type and a tag (line 12). In the example the message type corresponds to the transmission of ghost elements and the tag matches the iteration number. Using tagging removes the need of a synchronization call after every iteration, because it separates messages from different iterations with otherwise equal type, sender and receiver.

The code in Figure 3(b) makes explicit the messages that must be received before the computation can take place. It defines the control dependencies in the program. Given that message receptions are specified in the code, this high-level description promotes a *receiver centric* view of the program. However, it still retains the flexibility of the model by allowing different reception orderings via the `overlap` construct. Permitting different sequences of message receptions may be a source of non-determinism. However, if the behavior of the program is the same, regardless of the reception order of messages within an `overlap` region, then we say the statements in the region *commute* and any ordering in the reception of those messages is always correct. Note that different sequences of message receptions may lead to a different order of message emissions. However, a different order in the sending of messages is natural in the system model. The receiver has to ensure that messages are actually processed in the right order.

Expressing more clearly the control and data dependencies in the program not only permits eliminating artificial synchronization calls, but effectively removing determinants. The reason of existence of determinants is to guarantee a consistent recovery. Therefore, determinants ensure messages are received in the same order during recovery as they were before the failure. Additionally, the combination of $\langle \mathit{sender}, \mathit{receiver}, \mathit{ssn} \rangle$ is used in discarding duplicate messages. In the stencil code for the two-dimensional stencil of Figure 1(b), it is possible to avoid the generation of determinants if a high-level script is used. Figure 3(b) shows that a `Block` would receive the four messages from its neighbors in every iteration before computing and sending out the messages for the next iteration. If a block α would be recovering, then the high-level structure of the code would order all the messages being resent from the other objects. No determinants are required to guarantee a successful recovery. The other objects should, nevertheless, discard duplicate messages.

Using the stencil code as an example, we provide a list of conditions for the total elimination of determinants in a program expressed in a high-level language:

1. **Unique messages.** It should be possible to uniquely identify each message. Besides the combination of $\langle \mathit{sender}, \mathit{receiver} \rangle$, each messages should be tagged with a msgID that will tell apart otherwise identical messages. For instance, the msgID may be a composition of message type and iteration number. That particular combination would make messages unique in stencil codes.

2. **Commutative overlap regions.** The statements in `overlap` regions must commute.
3. **Explicit causal ordering.** The program must make explicit the causal order between two messages.

In addition, if determinants disappear from the message-logging layer, it must be guaranteed that all internal structures in the runtime system are still correct. This includes all the data structures for load balancing, checkpointing, collective communication operations, etc.

4 Fast Message-Logging Protocol

Using the insight from the previous section and the background work of Section 2, we present a new breed of message-logging protocols. We name it *fast message-logging* because it focuses on accelerating the three sources of overhead in a resilience solution: forward path, checkpointing, and recovery. This is brief justification about why it is fast message-logging:

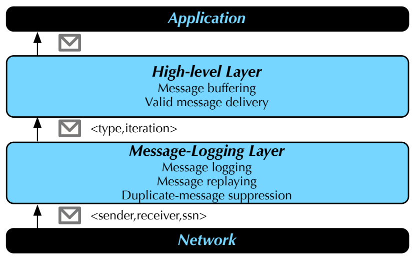
Fast forward-path By removing determinants from an execution, it reduces the slowdown of message-logging. According to Figure 1(a) determinants are the main source of overhead, so eliminating them should bring the performance overhead to a minimum.

Fast checkpoint It uses local checkpoint to avoid checkpoint-time congestion in the file system. This reduces the robustness of the system, but just by a tiny margin, given that most failure in HPC systems only involve a single node [8].

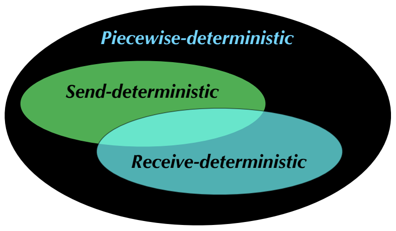
Fast recovery It uses parallel recovery to accelerate the recovery of the failed components.

The fast message-logging protocol only generates message identifiers and does not create determinants. At send time, the tuple $\langle sender, receiver, msgID \rangle$ is used to label each message. This information will be used to eventually ignore messages if they are repeated. It relies on the high-level scripting infrastructure to buffer early messages and to deliver them in a correct order for the program. This way, there is a clear separation of concerns with respect to the functions that must be executed during recovery. Figure 4(a) show the way this protocol achieves a separation of concerns between the layers in the software stack. The bottom layer stands for the message-logging protocol that is in charge of reacting to a failure. That layer is in charge of performing two tasks. The first task is to replay the messages to the failed objects. The second task is to suppress duplicates during recovery based on the tuple $\langle sender, receiver, msgID \rangle$. The top layer is the high-level language infrastructure that guarantees the consistency in the state of the failed objects by delivering the messages in the correct order.

We call the type of programs that comply with all the conditions to remove all the determinants *receive deterministic*. Even when receives and sends may not be deterministic, the state of the objects will nevertheless be the same. Figure 4(b) shows a Venn diagram with the relationships between piece-wise deterministic, send-deterministic and receive-deterministic programs.



(a) Separation of concerns with the fast message-logging protocol.



(b) Relationships between piecewise, send and receive determinism.

Fig. 4. Fast message-logging protocol. The functionalities of the protocol are split with the high-level scripting infrastructure. The type of programs that can run with this protocol overlap send-determinism and require piece-wise determinism.

4.1 Algorithmic Description

The fast message-logging algorithm assigns a message identifier $msgID$ to each message. There are various sources for this identifier. If the application is send-deterministic, the protocol may assign a sender sequence number (ssn) to each message based on the combination $\langle sender, receiver \rangle$. If the application is a receive-deterministic stencil, the $msgID$ can be the concatenation of iteration and message type. Otherwise, the message type and the tag associated with that message will form the identifier for the message.

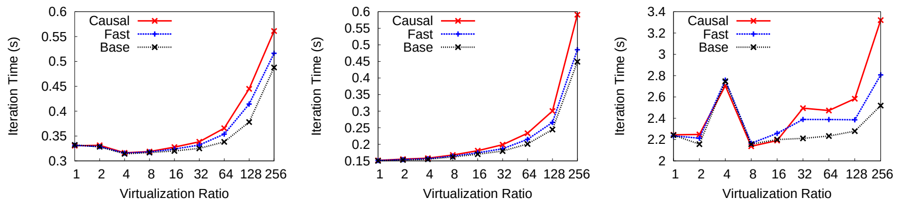
There are a handful of major data structures that keep the protocol correct. The structure `idTable` returns the $msgID$ for each message and combination of $\langle sender, msg, receiver \rangle$. For each of the cases explained above, that structure will return a unique identification for each message. The structure `dupTable` determines whether a message is a duplicate or not. Again, depending on the properties of the application, this structure can be optimized to improve performance. The structure `msgLog` stores all messages sent between the PEs. Finally, a couple of lists keep track of objects that have been distributed to other PEs for parallel recovery. If an object α resides in a PE A that crashes, α might be sent to other PE B for recovery. In that case, we refer to α as an *emigrant* from the perspective of A and as an *immigrant* from the perspective of B. The sets `listImmigrants` and `listEmigrants` store the list of objects in each category, respectively.

Algorithm 1 presents the main procedures of the fast message-logging protocol. Procedures Send and Receive describe the process for sending and receiving a message, correspondingly. At the sender side, messages must carry the combination $\langle sender, receiver, id \rangle$. All messages are stored in the message log. The reception of a message includes verifying that the message is not a duplicate. Once this is ensured, the message is passed to the layers above to be correctly processed. This may include buffering the message, forwarding the message to other PE or delivering the message to the application.

Algorithm 1 FASTMESSAGELOGGING

```
procedure SEND($\alpha, msg, \beta$)                                ▷ Object $\alpha$ sends $msg$ to object $\beta$
    $id \leftarrow$ idTable.getID($\alpha, msg, \beta$)
    $msg.id \leftarrow id$
    $msg.sender \leftarrow \alpha$
    $msg.receiver \leftarrow \beta$
    $msg.incarnation \leftarrow$ IncarnationNumber
    if $\alpha$.PE $\neq$ $\beta$.PE then
        msgLog.add($msg$)                                            ▷ Storing remote message
    end if
    NetworkSend($msg$)
end procedure
procedure RECEIVE($\alpha, msg, \beta$)                             ▷ Object $\beta$ receives $msg$ from object $\alpha$
    $num \leftarrow msg.incarnation$
    if OldIncarnation($num$) then
        DiscardOld($msg$)                                            ▷ Ignoring old message
    end if
    $flag \leftarrow$ dupTable.getFlag($\alpha, \beta, msg.id$)
    if $flag$ then
        DiscardDuplicate($msg$)                                      ▷ Ignoring repeated message
        return
    end if
    Process($msg$)                                                   ▷ Forward to high-level layer
end procedure
procedure CHECKPOINT                                                 ▷ Called at PE $A$
    Send all objects $\alpha$ in listImmigrants
    Wait for all objects $\alpha$ in listEmigrants
    $ckptMsg \leftarrow \{\}$
    msgLog.clean()
    for all objects $\alpha$ do
        $ckptMsg.add(\alpha.state)$
        NetworkSend($ckptMsg$)
    end for
end procedure
procedure RESTART($A$)                                               ▷ Received at every PE except for $A$
    for all objects $\alpha$ in $A$ do
        Send all messages bound to $\alpha$ in msgLog
    end for
end procedure
procedure RECOVERY                                                   ▷ Called at PE $A$
    for all objects $\alpha$ in $A$ do
        Distribute $\alpha$ to a PE in $\{A_1, A_2, ..., A_P\}$
        Add $\alpha$ to listEmigrants
    end for
end procedure
```

Fast message-logging relies on globally coordinated synchronized checkpoint. The programmer uses global synchronization points in the application to trigger the checkpoint calls. Procedure CHECKPOINT presents the steps included in the checkpoint process. First, all immigrant objects are send back to their original PEs. These objects may have been migrated to a PE for parallel recovery purposes. Next, a PE waits for all its emigrant objects to arrive. The message log is cleaned up and a new checkpoint message is built with the state of all the objects. Once a failure has been detected, the runtime system notifies the rest of the PEs about the crash of one PE. Let us assume PE A fails. The failure notification activates RESTART in other PEs, which replay the messages in msgLog bound to objects in PE A. Concurrently, PE A executes RECOVERY to distribute all its objects to as many PEs are involved in parallel recovery. The set $\{A_1, A_2, ..., A_P\}$ stands for the set of other P PEs helping the recovery of A.



(a) Wave2D. (b) Jacobi3D. (c) LULESH.

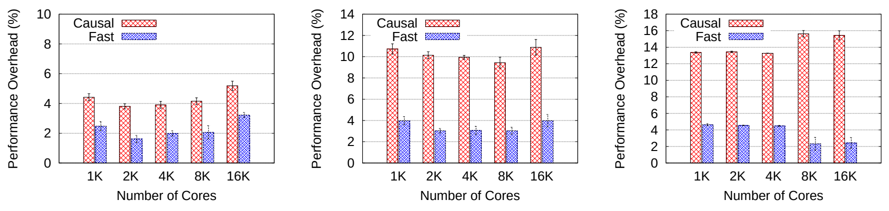
Fig. 5. Effect of virtualization ratio on performance. Stencil codes admit a high virtualization ratio without sacrificing a significant portion of its performance.

5 Experimental Results

In order to understand the performance penalization imposed by the fast message-logging protocol, we examined three stencil codes: Wave2D, Jacobi3D and LULESH. These programs were implemented in SDAG to decrease execution time (since barriers between iterations are not needed anymore) and allow the possibility of removing determinants.

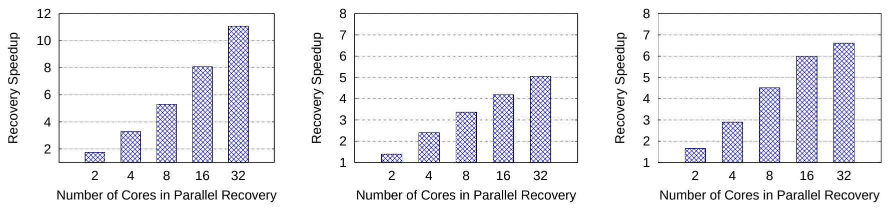
We first examine the effect of virtualization ratio on each program. Recall from Section 2 that virtualization ratio stands for the average number of objects per PE. For this experiment, each PE is a core. Figure 5 presents the results on $1,024$ cores of Intrepid supercomputer at Argonne National Laboratory (ANL). Each data point reflects the average of 5 repetitions of the same experiment. Wave2D in Figure 5(a) seems to manifest that Wave2D benefits from virtualization. In fact, the performance of the program stays within a 5% difference up to 32 objects per core. A similar behavior occurs in Jacobi3D in Figure 5(b), except in this case the program is always slightly slowed down by virtualization. Again, the average iteration time stays within a 10% margin up to 32 objects per core. Finally, Figure 5(c) accepts a higher virtualization ratio without a significant performance degradation. Even 128 objects per core stays within a 3% margin. Having a high virtualization ratio is beneficial for applications in view of the eminent thermal variation of future systems. With more objects per core, it is possible to obtain a better load balance in case some of the PEs have to be slowed down to reduce their temperature.

Using a virtualization ratio of 32 for Wave2D and Jacobi3D and 128 for LULESH, we ran 5 times each program and compared the results between fast message-logging and simple causal message-logging. The results are presented in Figure 6 and offer the relative performance overhead when compared with the base Charm++. The figure shows a weak-scale experiment on Intrepid from 1K to 16K cores. For Wave2D, Figure 6(a) shows that fast message-logging is able to halve the performance across all the spectrum. Figure 6(b) tells about a more beneficial scenario for the fast protocol with Jacobi3D. The most dramatic difference appear with LULESH. Figure 6(c) shows the big difference determinants can make in message-logging. As the program scales the difference between



(a) Wave2D. (b) Jacobi3D. (c) LULESH.

Fig. 6. Performance overhead of fast message-logging. It approximately reduces performance overhead to a half when compared with causal message-logging.



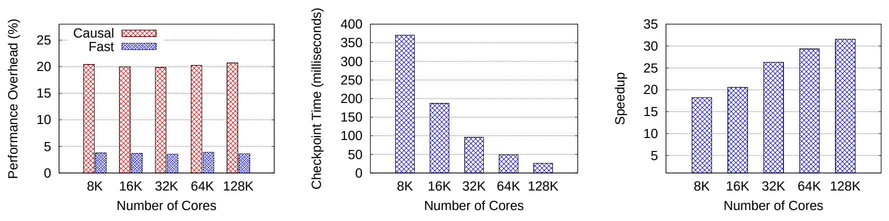
(a) Wave2D. (b) Jacobi3D. (c) LULESH.

Fig. 7. Parallel recovery in fast message-logging. The more cores are used to recover, the faster the recovery.

the fast and causal variants increases to the point where the overhead of fast message-logging is around 15% of the overhead of causal.

An important feature of the fast message-logging protocol is it ability to work in conjunction with parallel recovery. Section 2 introduced parallel recovery. The more PEs help in recovery, the faster it is expected the failed PE will catch up with the rest of the system. There are, however, a couple of considerations to keep in mind. First, the number of PEs helping in recovery cannot exceed the virtualization ratio. Second, it is possible to have diminishing returns with the addition of more PEs. This latter effect results stems from the fact that more PEs involve sending more remote messages and potentially hitting network bottlenecks. Figure 7 shows for the 3 applications the speedup in recovery when the number of PEs helping ranges from 2 to 32. Wave2D offers the best scenario for parallel recovery. This is due to its relatively simple communication characteristics. Therefore, spreading the objects to more PEs results in a greater speedup. Both Jacobi3D and LULESH present moderate speedup levels during recovery.

Figure 8 offers an overall demonstration of the advantages of using fast message-logging. This scenario corresponds to a large-scale run where the number of cores varies from 8K to 128K on Intrepid. Additionally, it represents a strong-scale test, that stresses even more the message-logging protocol. Figure 8(a) shows the overhead in the forward path can always be kept low. Whereas causal message-logging has an overhead that goes up to approximately 20%, fast message-logging has an overhead lower than 4%. Checkpoint time is measured



(a) Performance Overhead. (b) Checkpoint Time. (c) Speedup.

Fig. 8. Large strong-scale experiment with LULESH. Fast message-logging keeps a low performance overhead, fast checkpoint and fast recovery, compared to checkpoint/restart.

Program	Remote Messages	Generated Determinants	Piggybacked Determinants	Determinants per Message
Wave2D	72.90	144.40	317.59	4.36
Jacobi3D	141.10	214.80	870.78	6.17
LULESH	3181.85	4295.55	32411.52	10.19

Table 1. Average statistics on messages and determinants per iteration.

in milliseconds. Figure 8(b) offers the checkpoint time as the program is scaled. Based on the runtime-level checkpoint and in-memory checkpoint, the dump of the state of an application is fast. Finally, the speedup over checkpoint/restart is reported in Figure 8(c). For this case, the program runs for two checkpoint periods (for an arbitrary checkpoint interval), with a failure injected in the second one. The speedup reported in the figure corresponds to the total execution time of fast message-logging versus checkpoint/restart.

Table 1 presents statistics about the number of messages and determinants in different programs collected on Intrepid with 1,024 cores and with the same configuration as in Figure 6. The causal message-logging protocol was used to collect the data. The table shows the number of instances per iteration, averaged across all cores. LULESH has the highest numbers, because it has a more complex communication graph (neighbors in 3 dimensions) and a higher virtualization ratio (128 compared to 32). The higher number of messages is proportional to the number of determinants generated. In Table 1 there are more determinants than messages, explained by the fact that local messages generate determinants too. The number of piggybacked determinants also grows as the number of messages. The ratio of determinants piggybacked over determinants generated provide an idea on the number of copies each determinant has in the system. Although causal message-logging only requires one copy of each determinant to be safely stored, usually several more copies are logged in the system. Finally, the last row in Table 1 reports the average number of determinants piggybacked per message. This quantity directly dictates the performance overhead of causal message-logging. Not surprisingly, more determinants piggybacked per message implies a higher performance overhead (supported by the results in Figure 6).

6 Related Work

The seminal work on send determinism [4] provided a clear insight on the futility of creating determinants for all message receptions. Protocols based on that property rely on FIFO channels. We removed that assumption to use the asynchronous execution model of Charm++. However, to tame the excessive flexibility in the order of message reception, we relied on high level programming language constructs. The send-determinism protocol requires to replay the causally-dependent messages in a synchronized fashion, which hampers the ability of message-logging to achieve a faster recovery. By using a reception control mechanism, fast message-logging is able to receive all messages at once and completely sort them in a valid program order. The large scale simulator *BigSim* [12] used parallel discrete event simulation (PDES) to make predictions on the performance of scientific codes for supercomputers. Since BigSim is based on POSE (a framework for PDES), it would allow speculative simulation of different threads of events. In that regard, programs that allow only one possible execution sequence would find a very efficient simulation. In BigSim, *linear order parallel programs* are a special kind of codes for which messages are processed in exactly one order. A scalable replay system was designed based on an algebraic framework to store partial-order dependencies in message-passing programs [6]. That framework improves on most replay algorithms, which make minimal assumptions about the programming model. If some deterministic decisions can be extracted from the programming model, the framework incorporate them into the number of determinants that must be stored. The management of determinant of the fast message-logging protocol presented in this paper is a particular case in that algebraic framework. A new design of the message-logging layer for Open MPI [1] revealed a vast amount of determinants that are not necessary to guarantee a safe recovery. By interposing a message-logging substrate between the application and the network layer, this new design is able to create matchings of events posted by the application and arrival events coming from the network. Matchings may be deterministic if the expected sender and the actual sender of the message are the same. Use of wildcards will create non-deterministic matches. By logging only non-deterministic matches, this design is able to dramatically reduce the number of determinants.

7 Conclusion and Future Work

This paper examines the major source of performance overhead in traditional message-logging protocols. By leveraging the infrastructure of a flexible parallel programming paradigm, a new strategy was proposed. The *fast* message-logging protocol addresses the 3 main sources of overhead by removing unnecessary determinants, checkpointing in memory, and recovering in parallel. The experimental results with typical stencil codes demonstrate the impact of this protocol. The tests scaled up to 128,000 cores. In the future, we plan to devise new protocols for particular types of applications where communication patterns are somehow regular and general-case expensive assumptions can be removed. We also plan to use message-tagging to implement the protocol in MPI.

Acknowledgments. This work was partially supported by a machine allocation on the XSEDE under award ASC050039N, and by a machine allocation on Argonne Leadership Computing Facility awarded by the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.
2. F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.
3. E. N. 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, 2002.
4. A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *IPDPS*, pages 989–1000, 2011.
5. L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.
6. J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. Kale. Scalable Replay with Partial-Order Dependencies for Message-Logging Fault Tolerance. In *Proceedings of IEEE Cluster 2014*, Madrid, Spain, September 2014.
7. E. Meneses, G. Bronevetsky, and L. V. Kale. Evaluation of simple causal message logging for large-scale fault tolerant HPC systems. In *16th IEEE DPDNS in 25th IEEE IPDPS*, May 2011.
8. E. Meneses, X. Ni, and L. V. Kale. A Message-Logging Protocol for Multicore Systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.
9. E. Meneses, O. Sarood, and L. V. Kale. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Computing*, 40(9):536 – 547, 2014.
10. M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *IJHPCA*, 28(2):129–173, 2014.
11. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.
12. G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.