

Event Logging: Portable and Efficient Checkpointing in Heterogeneous Environments with Non-FIFO Communication Platforms

Zhao Peng

*Department of Computer Science
University College Dublin, Belfield
Dublin 4, Ireland
Peng.Zhao@ucd.ie*

Alexey Lastovetsky

*Department of Computer Science
University College Dublin, Belfield
Dublin 4, Ireland
Alexey.Lastovetsky@ucd.ie*

Abstract

The Chandy-Lamport checkpointing algorithm is widely used in fault tolerant implementations of MPI. However, it assumes the FIFO property of message passing, which is not guaranteed by the MPI standard at the application level. Therefore, this algorithm cannot serve as a basis for an implementation-independent fault tolerant MPI. In this paper, we present a variant of the Chandy-Lamport algorithm that does not rely on the FIFO property. This algorithm can be implemented on top of MPI and, hence, used for development of a supplement software component enabling the fault tolerance of any MPI implementation compliant with the MPI standard. We prove the correctness of the algorithm and analyze its performance. Experimental results demonstrating the efficiency of the algorithm are also presented.

1. Introduction

Fault tolerance is one of the primary issues of parallel programming for common heterogeneous networks of computers because in such environments the probability of resource failures is much higher than in dedicated parallel computer systems of the same scale. Checkpoint/recovery is an important technique used to enable the fault tolerance of parallel programs. By resuming the execution from where a recovery line has been created, the program does not need to restart from the beginning but can rollback to the latest recovery line. MPI [6, 9] is probably the most popular parallel programming tool for heterogeneous networks. Therefore, no wonder that research and development efforts aimed at enabling the fault tolerance of MPI programs are constantly growing. The Chandy-Lamport

checkpointing algorithm [5] is used in most of fault tolerant implementations of MPI [1, 22, 24]. However, it relies on the FIFO property of message passing, which is not guaranteed by the MPI standard at the application level. Therefore each of these fault tolerant MPIs has to resort to the low-level layer's help to implement the Chandy-Lamport algorithm. As a result, the software component responsible for checkpointing will be inherently implementation-specific, hardly built into the particular implementation of MPI and not portable to other MPI implementations.

An alternative approach to enabling the fault tolerance of MPI applications is the development of an autonomous and portable software component responsible for checkpoint/recovery that could be used together with any (non-fault-tolerant) MPI implementation compliant with the MPI standard [9]. Such a software component can only be based on a checkpointing algorithm that does not rely on any property of MPI which is not guaranteed by its standard.

In this paper, we present Event Logging, a variant of the Chandy-Lamport algorithm that does not rely on the FIFO property of message passing. This algorithm can be implemented on top of MPI and, hence, used for development of a supplement software component enabling the fault tolerance of any MPI implementation compliant with the MPI standard. We prove the correctness of the algorithm and analyze its performance. Experimental results demonstrate that this portable algorithm is still quite efficient. We also briefly present libELC, a portable prototype library for enabling the fault tolerance of MPI applications based on the Event Logging algorithm.

2. Background

2.1. Problem Space

Creation of a checkpoint of an MPI program includes the creation of a checkpoint for each individual process. Uniprocess checkpointing captures the process's state and outputs it to an external stable storage, from where this state can be reloaded when an error occurs. Comprised of a set of the individual process's checkpoints, the global checkpoint can be used to restore a previous execution state of the MPI program. In this context, the checkpoint of individual process is named a *local checkpoint*, and the *global checkpoint* is also called a *recovery line*.

However, only the local checkpoints are not sufficient to form a recovery line for the running MPI program. As MPI processes use message passing to communicate, messages are part of the state of the running program. As any communication has latency, there may be some messages on fly at the time when an individual process's state is saved. Therefore, the checkpoint algorithm must be able to capture the program's communication state.

Generally, in the context of checkpointing there are three kinds of messages in MPI: *intra messages*, *in-transit messages* and *orphan messages*. Lamport [14] proposed a relation called *Happens Before* to indicate the partial order of events in a distributed system. This is an irreflexive, antisymmetric, transitive relation that can be applied to define these messages.

1. If events a and b happen on the same process and a happens before b , then $a \rightarrow b$.
2. If events a and b happen on different processes, and a is a sending event of message M , b is a receiving event of M , then $a \rightarrow b$.
3. If neither $(a \rightarrow b)$ nor $(b \rightarrow a)$, then a and b are concurrent events.

Given the definition of the *Happens Before*

relation, these three kinds of messages can be expressed as follows:

1. *Intra message*: $CKPT_i \rightarrow S(M) \rightarrow CKPT_{i+1}$ and $CKPT_i \rightarrow R(M) \rightarrow CKPT_{i+1}$.
2. *In-transit message*: $S(M) \rightarrow CKPT_i$ and $CKPT_{i+1} \rightarrow R(M)$
3. *Orphan message*: $CKPT_i \rightarrow S(M)$ and $R(M) \rightarrow CKPT_i$.

Here, $CKPT_i$ denotes the process's i -th local checkpoint, and $S(M)$, $R(M)$ denote the events of sending and receiving of message M respectively.

Among these messages, intra message is harmless, because the passing of an intra message does not cross the recovery line, which means that the message will not exist in the communication channel. However, in-transit and orphan messages are dangerous. Consider the in-transit message example depicted in Figure 1. Assume that a failure occurs after the system finishes the recovery line C1-C2, and the execution then is restarted. Without saving the communication state, process P1 thinks that it has sent the message to process P2, so it will never replay the sending. On the other side, the local checkpoint of P2 is taken before it receives the message. It leads to the problem that P2 will be blocked on waiting for the in-transit message after recovery but the message may have been lost or discarded by the network during the program's failure. This problem caused by in-transit message is called *unrecoverable*.

In Figure 2, a message is sent after P1's local checkpoint and gets received before P2's. Upon recovery, P1 will re-send this orphan message, which actually has been received and saved in P2's checkpoint. Although the execution can be recovered with the existence of the orphan message, the damage of this message is not only a waste of the buffer space. More seriously, it produces a duplicated message that may break the communication semantics. This problem is called *inconsistency*.

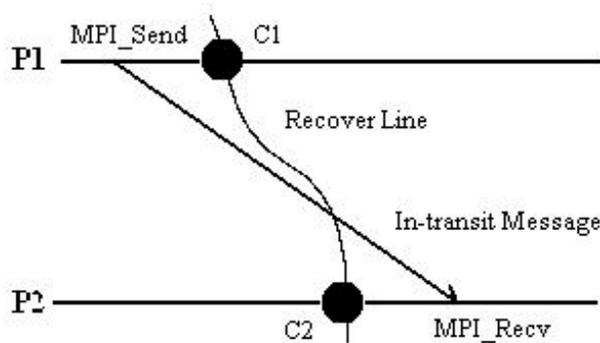


Figure 1. In-transit message.

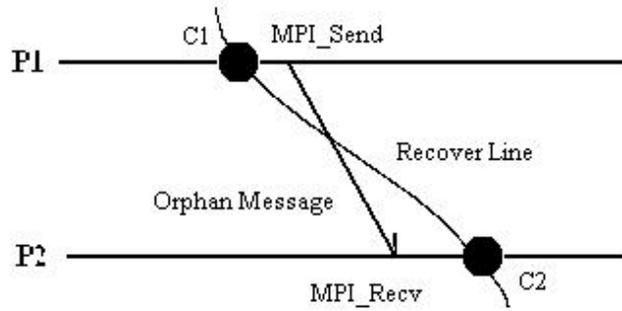


Figure 2. Orphan message.

So in order to create a valid recovery line, the following two conditions should be satisfied:

1. **Recoverability:** either there is no in-transit message or, if present, such a message can be regenerated after recovery.
2. **Consistency:** either there is no orphan message or, if present, such a message can be eliminated after recovery.

2.2. Related Work

2.2.1. Overview

A checkpoint algorithm can be classified as coordinated or uncoordinated. A coordinated algorithm forms the recovery line by orchestrating the processes to do the checkpoint, in which a valid recovery line is immediately setup once the checkpoint finishes. An uncoordinated checkpointing allows the processes to choose the time of checkpoint independently, so the system has to select from the independently created local checkpoints to form a recovery line during recovery. This not only makes the recovery procedure complicated, but also suffers from *Domino Effect* [20]. Compared, coping with message logging, the uncoordinated approach is beneficial for large scale distributed systems

running thousands of processors, because only the crashed nodes need to rollback, the others can go through the failures [3]. On the other side, the coordinated protocol requires all processes to rollback upon failures but without the need of message logging, which is more economic for communication intensive parallel programs running in small and medium size environments.

There are two approaches to coordinated checkpointing: blocking and non-blocking. Typically, the blocking one synchronizes all processes at the checkpoint until the recovery line has been set up. A non-blocking system allows a process to resume execution right after it finishes the local checkpoint. Given the high overhead introduced by the blocking, the non-blocking approach gets preferred in practice.

As to implementation, some non-blocking coordinated checkpoint algorithms can only be implemented at the system level, meanwhile others can be implemented at the application level. A system-level library is implemented at the level of OS kernel or network system to gain high performance. An application-level checkpoint is managed by an external library, portable cross platforms. Our algorithm is designed

Type	Advantage	Disadvantage
Coordinated	Domino Effect Free; No Message Logging	Coordination Overhead Global Rollback
Uncoordinated	Independent Checkpoint Only Failed Process Rollback	Domino Effect Message Logging Overhead
Blocking Coordinated	Simple Protocol	High Blocking Overhead
Non-blocking Coordinated	Low Performance Overhead	In-transit and Orphan Messages
System-level	High Efficiency	Low Portability
Application-level	High Portability	Medium Performance

Table 1. Comparison of different designs.

to be suitable for application-level implementation. More comprehensive surveys can be found in [7, 11].

2.2.2. Chandy-Lamport Algorithm

Chandy-Lamport algorithm is one of the earliest and most widely used coordinated, non-blocking algorithms for the checkpoint of distributed systems. It is based on the following assumptions:

1. The system is comprised of finite numbers of processes and communication channels.
2. Communication is done by passing messages through the communication channels.
3. The latency of communication is finite and the communication is reliable [10].
4. The message passing behaves in the FIFO manner.

The algorithm can be summarized as follows. Let IN_p and OUT_p be the number of incoming and outgoing communication channels respectively, which process p holds. Then, each process p :

1. Creates a local checkpoint.
2. For $i=1$ to OUT_p , p sends a marker message along the outgoing channel i .
3. p continues its job.
4. For $i=1$ to IN_p , p logs the messages from the incoming channel i until a marker message is received along channel i .
5. When all processes finish the above steps, the recovery line is formed.

The reason why the *Chandy-Lamport* algorithm relies on the FIFO assumption is that the marker used in the algorithm acts as a fence to separate the message passing around the local checkpoint. Consider the scenario depicted in Figure 3. Process P1 sends three messages in the order: $\{M1, Marker, M2\}$ to P2. With the FIFO manner, the messages reach P2 in the same order as sent.

Assume that at the time when M1 reaches P2, P2 has already finished its local checkpoint. So that, M1 is logged as an in-transit message and M2 is an intra message according to the *Chandy-Lamport* algorithm. But, if the underlying communication layer does not behave in the FIFO manner, the arriving order may be different from the sending order. If the arrival order is $\{Marker, M1, M2\}$, M1 will be omitted from logging making the recovery line unrecoverable. More, if the arrival order is $\{M1, M2, Marker\}$, the logging of M2 results in an inconsistent recovery line.

Being the cornerstone of the Chandy-Lamport algorithm, the FIFO property of message passing may only be true when looking deeply into the details of MPI implementations: most MPI implementations define a low level channel, which seats on top of the underlying network. On low performance networks such as Ethernet, TCP is used, providing FIFO communications. The high performance NIC (Network Interface Card) also guarantees the FIFO property. Therefore, there are many checkpoint/recovery systems for MPI, which implement the pure Chandy-Lamport algorithm at the system level [1, 24, 26]. However, this property is not guaranteed by the MPI standard at the application level. Generally speaking, it is valid to receive messages in the reverse order to the sending one. While this situation may be rare, much more common are the situations when messages are sent and received out of sequence.

Although the priority rule in MPI stipulates some messages passing to obey the FIFO property (see 3.2), FIFO is not strictly required by the MPI standard. For instance, messages with different envelopes can be received in any order (different from the sending order). Among many variants of the Chandy-Lamport algorithm that

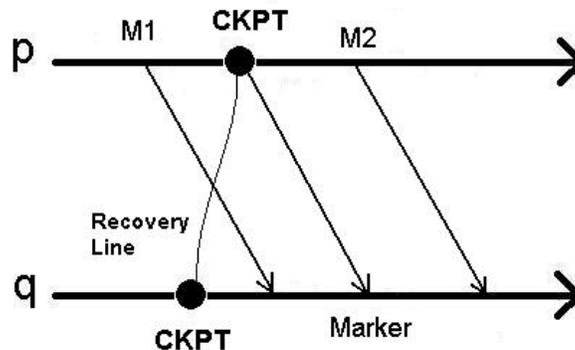


Figure 3. FIFO Message Passing

have been developed in recent years [4, 12, 13, 16, 17, 23], only two provide support for application-level non-FIFO message passing.

One is Time-Based Coordinated Checkpointing [19, 25]. It coordinates distributed processes by using loosely-coupled clocks. It assumes the existence of the maximal (T_{max}) and minimal (T_{min}) communication latencies as well as standard clock deviation T_{dev} . If the global time is T_{CKPT} , any process' clock will be in the interval $T_{CKPT} \pm T_{dev}$. So the maximal clock deviation between two processes is $2 \times T_{dev}$. Then the channel is cleared by caching all messages passed in the period $\{ T_{CKPT} - 2 \times T_{dev} - T_{max}, T_{CKPT} + 2 \times T_{dev} - T_{min} \}$. In a small network, this algorithm works pretty well. But with the growth of network scale, T_{max} , T_{dev} increases dramatically, making the message passing caching much longer, which impacts the performance significantly. Also, in a heterogeneous NoC, parameters T_{max} , T_{min} , T_{dev} actually cannot be accurately measured.

Another effort to tackle the non-FIFO is Message Tagging [4, 18]. The idea behind Message Tagging is that the system wraps some additional information onto the outgoing messages, which is called header. From the point of view of implementation, same with the checkpoint, the tagging can be done in two ways: the system-level approach tags the information by doing the modification at the network layer, which seriously damages the portability; the application-level tagging directly piggybacks the header onto the application messages. Usually, the header contains the sender's checkpoint index. Since the header is bound with the message, whenever the message arrives to its destination, the system retrieves the header and compares the receiver's current index with the sender's to classify whether the message is in-transit, orphan or intra. So, the header replaces the marker message to help the system identify messages without the FIFO requirement.

Our experience shows that the performance of the application-level message tagging fluctuates with the message size. Generally, more overhead is introduced with the increase of message length (see Section 5). Also, message tagging is slow in terms of outputting recovery line, since the system have to wait for logging an in-transit message until the process posts a call to receive it. In this case, as long as the process defers receiving the in-transit message, the overall recovery line cannot be committed. This may result in significant latency during failure-free execution. Also we note that the application-level message tagging may cause some unexpected execution results when encountering

the wildcard communication. For instance, a process may want to detect the length of the incoming message (by using `MPI_Status`) and then allocate space for receiving it. However, the message tagging will cause an incorrect message length, since the header binds more data onto the original message. One may argue to exclude the header size upon detecting. However, there is difficulty to do so, because of the highly implementation-dependent definition of the `MPI_Status` structure. Furthermore, the message tagging does not avoid coordination. Marker messages are still needed to broadcast the number of in-transit messages.

Given all these considerations, the message tagging is not likely to be an attractive option in practice.

3. Event Logging

3.1. Overview

In the following paragraphs, we present *Event Logging* for the application-level process coordination. First, it is necessary to differentiate Event Logging from Message Logging. Event Logging is a variant of the Chandy-Lamport algorithm that coordinates distributed processes to form a recovery line. Unlike message logging [2], it records only the message envelopes, without the message content, reducing much of the memory overhead introduced by message logging [21].

Also, Event Logging is different from the event logger [3], which is widely used in message logging systems for tracking the process casual dependency. Although the work of both Event Logging and event logger is to record message-passing events, the fundamental difference is that the former is a high level algorithm used for process coordination, while the later is a low level module that is built in the message logging system.

In *Event Logging*, every process keeps a log for sending and receiving events it performs. Upon occurrence of a new checkpoint, the send logs are exchanged between the sender and receiver. When the process gets another one's send log, it pairs it up with the local receive log to match the message envelopes. As the message envelope is logged at the time of the message passing (sending and receiving), the event log also keeps the message's *Happen Before* relation, which determines the type of the message: Intra, In-transit or Orphan. Then, when a pair of send and receive logs are matched, the system determines the category of the message.

3.2. Definitions and Assumptions

The process's execution is divided into a sequence of intervals separated by checkpoints. A checkpoint interval starts with any instruction following a local checkpoint and ends upon the completion of the next local checkpoint. Since our protocol is based on coordinated checkpoint, the local checkpoints of a recovery line have the same sequence number value.

With the concept of checkpoint interval, a send log is the collection of the outgoing message envelopes, denoted by $SEND_{p,i}$ (the send log of process p 's i -th checkpoint interval). Similarly, a receive log is $RECV_{p,i}$. We use $SEND_{p,i}\{R_{q,j}\}$ ($RECV_{p,i}\{S_{q,j}\}$) to denote the whole message passing event, in which process p sends out messages in its i -th checkpoint interval and the messages are received by process q in the j -th checkpoint interval.

Consider the example depicted in Figure 4. The send log of p 's i -th checkpoint interval is $SEND_{p,i}\{R_{q,i}, R_{q,i+1}, R_{q,i+2}, R_{q,i+3}\}$, which is simplified as $SEND_{p,i}\{R_{q,i}, R_{q,i+1}\}$, and the receive log of q 's i -th checkpoint interval is $RECV_{q,i}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}, S_{p,i+2}\}$, which equals to $RECV_{q,i}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$.

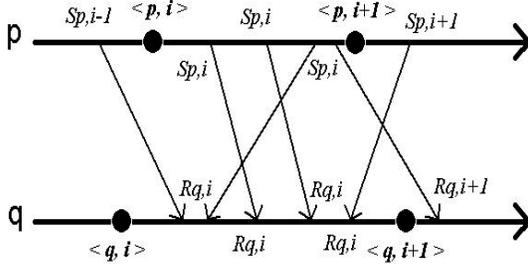


Figure 4. The p 's send event log and q 's receive log. $\langle p, i \rangle$ denotes p 's T_{CKPT} i -th checkpoint.

Obviously, between the send and receive logs, there exist the matching envelopes:

$$\begin{aligned} SEND_{p,i}\{R_{q,i}\} &= RECV_{q,i}\{S_{p,i}\}; & // \text{intra} \\ SEND_{p,i}\{R_{q,i+1}\} &= RECV_{q,i+1}\{S_{p,i}\}; & // \text{in-transit} \\ SEND_{p,i}\{R_{q,i-1}\} &= RECV_{q,i-1}\{S_{p,i}\}; & // \text{orphan} \end{aligned}$$

As the example shows, the goal of *Event Logging* is to match up the envelopes logged at the sender and receiver side in order to identify the in-transit and orphan messages.

To simplify the description of our algorithm, we make the assumption that any message passing finishes in no more than two successive checkpoint intervals. Later, we will show how to remove this

limitation for all kinds of communication environment. Under this assumption, the send and receive logs are cut down to $SEND_{p,i}\{R_{q,j-1}, R_{q,j}, R_{q,j+1}\}$ and $RECV_{q,i}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$.

Also, we emphasize that the Event Logging algorithm relies the Non-Overtaking property of MPI. Conceptually, one may think of successive messages sent by a process to another process as ordered in a sequence. Receive operations posted by a process are also ordered in a sequence. Each incoming message matches the first matching receive in the sequence [6]. For example, let a process P1 send three messages in the order {X1, X2, Y1} to another process P2 and X1, X2 have the same envelope X. With the non-overtaking property, if P2 posts only one receive matching X, X1 will always get picked.

However, the non-overtaking property only applies to messages with the same envelope. As in this example, Y1 could be at any place in the receiving order, since it has a different envelope from X1 and X2. Moreover, if P2 posts two receives both matching Y1, the first one will always be satisfied.

Note, that the notation $SEND_{p,i}\{R_{q,j-x}, \dots, R_{q,j-1}, R_{q,j}, R_{q,j+1}, \dots, R_{q,j+x}\}$ does not mean that the messages are sent in the receiving order. Also, we use $SEND_{p,i}\{-\}$ to denote a send log that cannot find a matching receive log, and $RECV_{q,i}\{-\}$ to denote an unmatched receive log.

3.3. Algorithm

The Event Logging consists of four parts: logging procedure, checkpoint procedure, identification procedure and recovery procedure. Let that the MPI program run N processes. Then each process will keep $2 \times (N-1)$ event logs.

3.3.1. Logging Procedure

For each process p , the logging procedure functions for each communication operations (send and receive). If p sends (receives) a message to (from) process q , then the logging procedure will go as follows: a new log node is created; the message envelope is filled into this node; the node is appended to $SEND_p\{R_q\}$ ($RECV_p\{S_q\}$).

3.3.2. Checkpoint Procedure

1. Upon receipt of a send log $SEND_q\{R_p\}$ from process q , if no checkpoint is in progress, process p goes to Step 2; otherwise it goes to Step 3.

2. p creates next local checkpoint (say, i -th checkpoint), sends $SEND_{p,i}\{R_x\}$ to each other process x of the program and then resumes the

normal execution.

3. p performs the identification procedure for q . Once p has completed identification of the messages received from q , all in-transit and orphan messages from q are saved as part of p 's local checkpoint.

4. After the message identification is finished for all other $N-1$ processes, the local checkpoint for process p will be created.

5. After all N processes have created their local checkpoints, the recovery line is formed.

3.3.3. Identification Procedure

Employed by the checkpoint procedure, the identification procedure detects in-transit, intra and orphan messages by comparing the send and receive logs ($SEND_{p,i}$ and $RECV_{q,j}$). Steps in this procedure are determined by the data structure used to save the event log. In its most simple form, link table, the identification procedure consists of the following steps:

1. For each node of $RECV_{q,j-1}$ ($SEND_{p,i-1}$), use Sequential Search to search in $SEND_{p,i}$ ($RECV_{q,j}$). When found a matching pair in $SEND_{p,i}$ ($RECV_{q,j}$), remove the pair from both $RECV_{q,j-1}$ ($SEND_{p,i-1}$) and $SEND_{p,i}$ ($RECV_{q,j}$).

2. For each node of $SEND_{p,i}$, use Sequential Search to search in $RECV_{q,j}$. When found a matching pair in $RECV_{q,j}$, remove the pair from both $SEND_{p,i}$ and $RECV_{q,j}$.

3. Upon completion the procedure, the unmatched send log $SEND_{p,i}\{-\}$ will consist of in-transit message envelope logs, and the unmatched receive log $RECV_{q,j}\{-\}$ will consist of orphan message envelope logs. They are both saved into p 's local checkpoint.

3.3.4. Recovery Procedure

The recovery procedure is summarized as follows:

1. Each process p reloads the execution state from the latest checkpoint $C_{p,i}$ and loads the in-transit and orphan message envelope log $SEND_{p,i}\{-\}$, $RECV_{q,j}\{-\}$.

2. Then it resumes the execution.

3. Upon receipt of a message with envelope M , p first checks whether $SEND_{p,i}\{-\}$ is empty. If not, it searches in $SEND_{p,i}\{-\}$ for a log of M . If found, it fills the receive buffer with the logged message M and removes the log from $SEND_{p,i}\{-\}$. Otherwise, it checks whether $RECV_{q,j}\{-\}$ is empty. If not, it searches in $RECV_{q,j}\{-\}$ for a log of M . If found, it receives the orphan message of envelope M in a temporary buffer and removes the log of M from

$RECV_{q,j}$.

3.4. Formal Analysis

Recalling the discussion in 3.2, if we can pair up the send log of a message with its receive log, we can discover the relation between the send and receive events and identify all in-transit and orphan messages. The following paragraph serves as a proof for Event Logging.

Theorem 1 *The algorithm is correct in the sense that it identifies all in-transit and orphan messages of the current checkpoint interval.*

For sake of simplicity, we consider an MPI program consisting of two processes. However, we argue that this scenario is universal for all cases, because the necessary and sufficient condition of the proposition that the recovery line of any MPI program is valid is that any two processes' sub-line of this recovery line is valid. This is because any message passing occurs between a pair of processes.

So our goal is to prove that Event Logging is capable to identify the intra, in-transit and orphan messages for any two processes. Let the current checkpoint interval of process p be $C_{p,i}$ and process q 's be $C_{q,j}$. According to the Happen Before relation, a message, M , which p sends to q in the current checkpoint interval is an intra message if and only if: $CKPT_i \rightarrow S(M) \rightarrow CKPT_{i+1}$ and $CKPT_i \rightarrow R(M) \rightarrow CKPT_{i+1}$. As to the Event Logging algorithm, it means that a pair of $SEND_{p,i}\{R_{q,j}\} = RECV_{q,j}\{S_{p,i}\}$ can be matched in the current checkpoint interval log. Message M will be an in-transit message if and only if $S(M) \rightarrow CKPT_i$ and $CKPT_{i+1} \rightarrow R(M)$. This means that a send log finds its matching receive log in the target's next checkpoint interval log, $SEND_{p,i}\{R_{q,j+1}\} = RECV_{q,j+1}\{S_{p,i}\}$. Finally, message M will be an orphan message if and only if $CKPT_i \rightarrow S(M)$ and $R(M) \rightarrow CKPT_i$. This means that a receive log finds its matching send log in the source's previous checkpoint interval log, $SEND_{p,i}\{R_{q,j-1}\} = RECV_{q,j-1}\{S_{p,i}\}$.

Although it is impossible to check future checkpoint interval logs, the trick is that the first checkpoint interval's logs $SEND_{p,i}\{R_{q,0}, R_{q,1}\}$ and $RECV_{q,0}\{S_{p,0}, S_{p,1}\}$ contain no in-transit or orphan messages. So we can easily match the intra message envelopes: $SEND_{p,0}\{R_{q,0}\} = RECV_{q,0}\{S_{p,0}\}$. To the unmatched send logs of p , $SEND_{p,0}\{-\}$, and the unmatched receive logs of q , $RECV_{q,0}\{-\}$, because the message passing must be completed in the next interval, we can conclude that the whole event of these unmatched event logs will be $SEND_{p,0}\{R_{q,1}\}$, $RECV_{q,0}\{S_{p,1}\}$. Then we can say that

all in-transit and orphan messages have been successfully identified in the checkpoint interval 0. This is the base of our induction.

Next, assume that at the checkpoint i -th interval $(C_{p,b}, C_{q,i})$ we have gotten the envelopes of the in-transit and orphan messages, $SEND_{p,i-1}\{R_{q,i}\}$ and $RECV_{q,i-1}\{S_{p,i}\}$. So, the event logs of $(C_{p,b}, C_{q,i})$ look like $SEND_{p,i}\{R_{q,i-1}, R_{q,i}, R_{q,i+1}\}$ and $RECV_{q,i}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$. Then, the algorithm removes the log of $SEND_{p,i-1}\{R_{q,i}\}$, $RECV_{q,i-1}\{S_{p,i}\}$. After the removal, the event logs of $(C_{p,b}, C_{q,i})$ look like $SEND_{p,i}\{R_{q,b}, R_{q,i+1}\}$ and $RECV_{q,i}\{S_{p,b}, S_{p,i+1}\}$, containing no logs related to $(C_{p,b}, C_{q,i})$. So, the same identifying method as for the first checkpoint interval can be applied. Finally, we get the in-transit and orphan messages of $(C_{p,b}, C_{q,i})$, $SEND_{p,i}\{-\}$ and $RECV_{q,i}\{-\}$, after removing the intra messages $SEND_{p,i}\{R_{q,i}\}=RECV_{q,i}\{S_{q,i}\}$. Same as $(C_{p,b}, C_{q,0})$, we can conclude that the final version of $SEND_{p,i}\{-\}$, $RECV_{q,i}\{-\}$ must be $SEND_{p,i}\{R_{q,i+1}\}$, $RECV_{q,i}\{S_{q,i+1}\}$.

Therefore, we conclude that for any checkpoint interval $(C_{p,b}, C_{q,i})$, where $i > 0$, all in-transit, intra and orphan messages will be identified.

A necessary note is that the algorithm pairs up send and receive logs by only matching envelopes. However, it is often that a process sends several messages with the same envelope to another process. Some messages could be orphan message, some could be intra message, and some might be still in fly at the time when the local checkpoint is taken. As the algorithm does not rely on the FIFO assumption, this could endanger the correctness of Event Logging seriously. However, the non-overtaking property removes this alert. According to it, the message passing with the same envelope obeys the following relation: If $x > y$ then $S_{p,i}(m_x) \rightarrow S_{p,i}(m_y)$ and $R_{q,j}(m_x) \rightarrow R_{q,j}(m_y)$. In other words, if messages a and b have the same envelope and a is sent earlier, the receiving of a must be ahead of b . Also, since the log is created along with the event (send or receive), we conclude that if a 's send log happens before b 's, a 's receive log must be ahead of b 's as well, either at some position before b in the same checkpoint interval, or in a previous interval. So, the event log of messages with the same envelope will be naturally ordered. As the log is created and compared sequentially, from the first passed message to the last one, the non-overtaking property guarantees that the matching does not overlap.

3.5. Extending the protocol

The above algorithm relies on the assumption

that message passing finishes in two successive intervals at most. However, in reality, it is possible that a message is received in a large latency. So, it is necessary to remove our limitation.

First, we exclude the danger of orphan messages, because none can send a message several checkpoint intervals after it is received by another one.

As to the in-transit message, the only requirement is that process can get it upon recovery, regardless how early the sending was. In other words, to recover from the i -th checkpoint interval, there is no difference between an in-transit message sent in the $(i-1)$ -th interval and the one sent in the $(i-4)$ -th. So we maintain a log for all the not-received in-transit messages and append the new in-transit-message send logs after it. All send event entries are kept until the message is picked up by the destination (then the entry will be removed from the log).

Our algorithm relies on no specific premise and can adapt to any communication demand. However, considering the balance between efficiency and programming flexibility, we leave users with the option of deciding for how many checkpoint intervals an in-transit message can be logged. By setting a parameter, the checkpoint protocol can be trimmed to the application's requirement.

4. Performance Analysis and Optimization

4.1. Analysis

In this chapter, we focus on the analysis and tuning of the performance of the logging and identification procedures. As to the checkpoint and recovery overheads, they are not Event Logging specific and general for all variants of the Chandy-Lamport algorithm.

The logging overhead is introduced by recording an envelope upon message passing. For any send (receive), the recording operation is the same: creating a new node, saving the envelope in the node and appending the new node to the log. So, the overall logging overhead is simply the sum of the costs of all logging operations. If the logging operation costs T_{log} , the number of message passing operations throughout the program's execution is N_m , then the logging overhead will be $T_{log} \times N_m$.

The identification overhead is incurred by the messages identification procedure. Consider the steps given in 3.3.3. For each item in the receive log, the algorithm repeats the search in the send

log trying to locate a matching envelope. In this sense, the identification overhead is mainly determined by the length of the send and receive logs. If $N_{q,j-1}$, $N_{q,j}$ and $N_{p,j}$ are the number of logs in $RECV_{q,j-1}$, $RECV_{q,j}$ and $SEND_{p,i}$ respectively, and T_m is the cost of a matching operation, the overhead expectation of the identification process using Sequential Search will be $T_m \times (2 \times N_{p,i} - 2 \times (N_{q,j-1} + N_{q,j}) + 3) \times (N_{q,j-1} + N_{q,j}) / 4$. In conclusion, the overhead caused by the message identification takes the most significant part in the evaluation of the Event Logging's performance. And the logging overhead is proportional to the number of message passing operations.

4.2. Optimization

Since the part of an MPI message envelope logged by each process includes 3 elements, which are *source-or-destination-rank*, *tag*, and *communicator*, a hash function can help to reduce the memory cost introduced by logging envelopes. However, the function must be a perfect hashing function. In other words, for any two different envelopes A and B, the hashing result must be different, $f(A) \neq f(B)$. Moreover, given that during the checkpointing procedure, processes need to exchange logs, the hashing can accelerate the communication, because of a much smaller size of the hashed log.

Also, the identification optimization strategy based on the hashing can be developed. As shown in 4.1, if a link table is used to save envelope logs, the matching process has to use Sequential Search to pair up send and receive envelopes. However, based on the non-overtaking property, the event log can be sorted by the hash value. Many well-known search algorithms can help to reduce the average search time. For example, by applying Binary Search, the average search length will be

$$\log_2 P_{N_{p,j+1}}^{N_{q,j-1} + N_{q,j}} - N_{q,j-1} + N_{q,j}.$$

In the case of multiple messages with the same envelope, an optimization technique may be to add an extra field to allow the event log to keep the number of occurrences of this envelope. For example, if message envelope M is used 100 times in the checkpoint interval, only one log of the message envelope needs to be kept by setting the counter field to 100.

Combining these two optimization techniques, the event log is reorganized in the form of sorted list with the value of each node being the hash value of a message envelope. Also, the node has another field, *NUM*, that holds the number of

messages passed with this envelope. Then, the identification procedure of Event Logging can be revised as follows:

1. For each node of $RECV_{q,j-1}$ ($SEND_{p,i-1}$), use Binary Search to search in $SEND_{p,i}$ ($RECV_{q,j}$). When found an identical value in $SEND_{p,i-1}$ ($RECV_{q,j}$), remove $Min(RECV_{q,j-1}.NUM, SEND_{p,i}.NUM)$ from both $RECV_{q,j-1}$ ($SEND_{p,i-1}$) and $SEND_{p,i}$ ($RECV_{q,j}$).
2. For each node of $SEND_{p,i}$, use Binary Search to search in $RECV_{q,j}$. When found an identical value in $RECV_{q,j}$, remove $Min(RECV_{q,j}.NUM, SEND_{p,i}.NUM)$ from both $SEND_{p,i}$ and $RECV_{q,j}$.

5. Implementation

Our current implementation for Event Logging algorithm consists of three packages: *MPI Wrapper Package* (MWP), *Message Identification Package* (MIP) and *Message Logging Package* (MLP).

5.1. MPI Wrapper Package

In our implementation, two main wrapper functions are $ELC_MPI_Send()$ and $ELC_MPI_Recv()$. In addition to sending and receiving a message, these two wrappers probe whether there is a pending checkpoint request in the MPI buffer. Depending on whether the checkpoint request is the first request detected, the process may take two different actions:

1. If the checkpoint request is the first one, the process creates a local checkpoint, broadcasts checkpoint requests containing this process's send event log to all other processes and then invokes the MIP. In general, MIP is responsible for identification of in-transit and orphan messages by using the Event Logging algorithm. After the identification completes, the receiving process will have the envelopes for all in-transit and orphan messages. The orphan message envelopes are written to the disk storage as part of the local checkpoint, and the in-transit message envelopes are handed over to MLP. Then the process adds the rank of the intercepted checkpoint request's source to a request table, which records the processes that have sent the checkpoint requests. After finishing this action, the process triggers MLP.
2. If the checkpoint request is not the first detected, process will directly start the MIP. The identified orphan message envelopes will be saved in checkpoint files. The in-transit message envelopes will be passed to MLP. Then the request's source

will be marked in the request table. The process counts the number of checkpoint requests it has received. When the process has gathered requests from all other processes, the process marks its local checkpoint as finished.

5.2. Message Identification Package

MIP is invoked to help the process to identify in-transit and orphan messages when the process receives a checkpoint request. According to the algorithm discussed in 3.3, the MIP tries to identify the messages by pairing up the send logs bound with the checkpoint request with the target's receive logs.

Let process q get a checkpoint request from process p within the i -th checkpoint interval ($i > 1$). In this scenario, q holds the event logs of the messages received from p , $RECV_{q,j}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$, and p 's send logs are packaged in the checkpoint request, $SEND_{p,i}\{R_{q,j-1}, R_{q,j}, R_{q,j+1}\}$. Also process q already has the logs of the in-transit and orphan messages of the last checkpoint interval, $SEND_{p,i-1}\{R_{q,i}\}$ and $RECV_{q,j-1}\{S_{p,i}\}$. MIP is invoked when process p intercepts send logs from process q , and hence the sender/receiver's ranks are known. Therefore, a pair of send/receive logs is matched if they have the same tag and communicator. Then, the identification process takes the following three steps:

1. Remove the in-transit message logs, $SEND_{p,i-1}\{R_{q,i}\} = RECV_{q,i}\{S_{p,i-1}\}$.
2. Remove the orphan message logs, $SEND_{p,i}\{R_{q,i-1}\} = RECV_{q,j-1}\{S_{p,i}\}$
3. Remove all intra message logs, $SEND_{p,i}\{R_{q,i}\} = RECV_{q,i}\{S_{p,i}\}$.

When the identification finishes, the remaining logs of $SEND_{p,i}\{-\}$ and $RECV_{q,i}\{-\}$ are the in-transit and orphan message envelopes in the current checkpoint interval.

5.3. Message Logging Package

MLP logs the in-transit messages using the envelopes identified by MIP. MLP is implemented in 2 different forms. The first one relies in the FIFO property of a lower layer of the MPI implementation, which guarantees that all in-transit messages will have been stored into the receive buffer, although may not have been picked up yet. So in this case MLP just posts a receive (`MPI_Recv`) for each in-transit message envelope.

The second version of MLP does not rely on the FIFO property of the lower communication level. In that case, MLP check in the incoming message

is in-transit or not. If so, the message will be logged. If not, nothing happens.

6. Experiments

6.1. Experiment Environment

In this section, we present the results of experiments with Event Logging. Four MPI programs are used: Gauss-Jordan method for solving systems of linear equations, Parallel NeuronSys, Monte Carlo Simulation and 1-D decomposition Matrix Multiplication. The experiments are carried out using a prototype application-level library libELC that enables the fault tolerance of MPI programs and based on the presented algorithms. One of the functions of the library is also to create portable local checkpoints for each individual MPI process following an approach similar to the Process Introspection proposed in [8]. Table 2 lists the machines used in the experiments.

The testing programs are run in three modes: the source mode, protocol mode, and checkpoint mode. In the source mode, original code is run. In the protocol mode, we apply libELC to the testing program, however no checkpoint is taken. In the checkpoint mode, both the Event Logging coordinating protocol is applied and checkpoints are created. Moreover, we add another value in the result tables, which is the expectation of the optimized program's execution time in the checkpoint mode. The expectation is calculated by the overhead model given in 4.1 and the program execution time in source mode. We compare this value with the experimental result to see whether the optimization works.

In the tests of Gauss-Jordan Solver, Parallel NeuronSys and Matrix Multiplication, four checkpoints are triggered by the function call `ELC_DoCKPT()`. Generally, we pick four random positions in the program to insert the calls. The Monte-Carlo Simulation program uses the time interval mechanism, when the checkpoint is triggered with a given frequency. Also, we vary the datasize and the number of processes for each test. The figures shown in the following tests are collected from a number of runs, discarding the outliers.

6.2. Testing Program

We have carried out the experiments of Event Logging with the following four MPI programs:

Machine	OS	CPUs (Mhz)
Csserver	Linux 2.4.20	4 @ 498
csultra01	SunOS 5.8	1 @ 440
csultra02	SunOS 5.8	1 @ 440
pg1cluster01	Linux 2.4.18	2 @ 1977
pg1cluster02	Linux 2.4.18	2 @ 1977
pg1cluster03	Linux 2.4.18	2 @ 1977
pg1cluster04	Linux 2.4.18	2 @ 1977
csa007bpc5	Linux 2.6.20	1 @ 930
csa007b3pc2	FreeBSD 5.2.1	1 @ 930

Table 2. Machine Configuration

1. The first experiment is an MPI program of Gauss-Jordan method for solving systems of linear equations, which is written by J. Meyer at University of Nebraska at Omaha. The linear system is evenly distributed by rows among N-1 processes, from where the results are collected by the MPI_Allreduce function call to the rank 0 process. We run the program on LAM/MPI 7.0.4 with three linear systems of different sizes: 4,000, 8,000 and 16,000.

2. Parallel NeuronSys is a neuron simulator program public available at <http://www.cs.usfca.edu/neurosyl/>. Generally, it is used to solve a system of ODE's modeling (Ordinary Differential Equation) a network of biologically realistic neurons on parallel computers. The current version uses fourth order Runge-Kutta method to solve the equation. Neurons are evenly distributed among processes and form a graph in which neurons excite and inhibit each other via their connections. Inter-process Communication contains five MPI_Allgather and one MPI_Gather function calls in each of a total of 10,000 iterations.

3. The third program does a Monte Carlo simulation of a system of hard disks. The fraction of the total area that is covered by disks (area fraction) is set to 0.5 and the user has control over the size of the system that will be simulated. The disks start from a triangular lattice and the simulation works in the master-slave pattern, in which the size of the system is determined by specifying the number of disks along an edge of the initial lattice.

4. In the 1-D Cyclic Decomposition Matrix Multiplication program, the master process (rank 0) behaves in a cyclic manner to distribute the matrix to the four slave processes, which does the multiplication job and returns the result to the master. The distribution unit is set to 4 columns so that the overall message number is Column/4 and the message size is 4 *Rows. More, we run this matrix multiplication program in the checkpoint mode using both Event Logging and Message

Tagging to compare the performance. The message tagging is implemented using derived datatype. In particular, the sending process encapsulates the original message with the header information into a temporary buffer, and calls MPI_Type_struct to create a derived datatype. On the other side, the target process receives the tagged message into a same size (original message plus the header) buffer space, retrieves the header information and hands over the original content to the user.

6.3. Results and Evaluation

1. Gauss-Jordan method: the overhead decreases from 22.20%, 3.83% to 1.95% with the linear equations' size changes from 4,000 to 8,000 and 16,000 (Figure 5). Simply put, the larger dataset increases the program execution time, but does not introduce extra message passing. Since the message envelope number stays the same, the overhead of envelope logging and identification becomes smaller with the increase of the program execution

2. Parallel NeuronSys: in this test, we observed the significantly reduced number of in-transit and orphan messages due to the accelerated execution by employing more processes. Much less in-transit and orphan messages mean that upon checkpointing, libELC spends much less time on the message identification and logging. As the result shows, the overhead decreases from 29.44% for 4 process configuration; while down to 9.94% for 8 processes; and 7.38% for 16 processes (Figure 6). It demonstrates the Event Logging algorithm scales well.

3. Monte-Carlo Simulation: in the simulation, we trigger a checkpoint in every 150 seconds. Also we use three sets of parameters to vary the test execution time. Although increasing input data size prolongs the program execution and causes more checkpoints, the performance still improves from 33.15% to 23.83% to 14.32% (Figure 7).

4. Matrix Multiplication: as mentioned above, the matrix multiplication program uses 1-D cyclic manner to decompose the matrix, which determines the message size Columns/4 and message number 4Rows is proportional to matrix size. In this case, we increase the matrix size from 512×512, 1024×1024, 2048×2048. Then the message size goes from 512 KB to 2 MB and 8 MB. As the result, we observe the overhead of Message Tagging approach increases along with the datasize increases: 4.85%, 7.40% and 8.05%; while the overhead of Event Logging remains about the same: 2.33%, 2.85% and 2.79% (Figure 8). Basically this result shows that the performance

of Message Tagging fluctuates with the message size. This is because Message Tagging approach needs to manage buffer space for tagging the header onto the message. However, Event Logging logs only the message envelopes, which will be affected by the message size. From the point of view of implementation, another advantage of Event Logging over Message Tagging is that Event Logging enables fast in-transit message logging. At the time the envelope identification finishes, Event Logging has the envelopes for the in-transit messages, so a process can simply post receive requests to log these in-transit messages (Recalling that FIFO manner is generally supported by the lower layer of MPI implementations). However, in Message Tagging, the process has to wait as long as the messages are received by the program. That is because the process has no knowledge of an in-transit message until it checks a message's header.

In general, we observe that the performance of Event Logging gets better with larger datasets. And the algorithm scales well in the experiments. Also, as the result of Monte-Carlo Simulation shows, using Time Interval mechanism, the overhead does not go up with the number of checkpoint created. Moreover, Event Logging performs better than Message Tagging in the Matrix Multiplication experiment. And the effect comes more significant with the increase of the message size. Another advantage of Event Logging is that it enables the fast log of in-transit messages. Finally, the experiment results are all in reasonable deviation from the expectation, which demonstrates the effect of the optimization skill given in 4.2.

7. Conclusion and Future Work

In this paper, we present a variant of Chandy-Lamport algorithm, Event Logging, to address the application-level non-FIFO problem of the coordinated checkpoint for MPI programs, which no existing approach provides adequate solutions given the heterogeneous network. Different from the previous works, this protocol coordinates the distributed processes by logging the message envelopes. Upon checkpointing, processes exchange the envelopes to identify the in-transit and orphan messages. The benefits of Event Logging are that it makes no assumption of the underlying MPI version, what makes it not only platform-independent, but also portable across various MPI implementations.

Event Logging overtops Message Tagging in term of recovery line commit. In Event Logging,

in-transit messages will be logged as soon as their envelopes have been identified. However, using Message Tagging process may face unpredictable latency for the program to receive all in-transit messages. Also, Event Logging performs better than the Message Tagging approach with the increase of message size. We have presented the skills to optimize the performance of Event Logging, which could be easily implemented and significantly reduce the overhead. Experiments demonstrated the high efficiency of Event Logging.

In the near future, an important improvement is to implement *Selective Checkpoint* [15] based on Event Logging. Furthermore, we plan to optimize the checkpoint by using compiler technologies to minimize the data that need to be saved [18, 19]. Integration of our protocol with mpC language [15] is also in our schedule.

Reference

- [1] A. Agbaria and R. Friedman. "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations". In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31. IEEE Computer Society, 1999.
- [2] L. Alvisi and K. Marzullo. "Message Logging: Pessimistic, optimistic, causal and optimal". *IEEE Transactions on Software Engineering*, 24(2):149–159, FEB 1998.
- [3] A. Bouteiller, F. Cappello, T. H'erauld, G. Krawezik, P. Lemarinier, and F. Magniette. "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging". In *Super Computing 2003*, 2003.
- [4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. "Automated application-level checkpointing of MPI programs". In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94. ACM Press, 2003.
- [5] K. M. Chandy and L. Lamport. "Distributed snapshots: determining global states of distributed systems". *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
- [6] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walker. "MPI: The Complete Reference". *The MIT Press*, 1996.
- [7] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. "A survey of rollback-recovery protocols in message-passing systems". *ACM Computing Surveys*, 34(3):375–408, 2002.
- [8] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw. Process introspection: "A heterogeneous

checkpoint/restart mechanism based on automatic code modification". *Technical Report CS-97-05*, Department of Computer Science, University of Virginia, May. 1997.

[9] M. P. I. Forum. "MPI: A message-passing interface standard". *Technical report*, 1994.

[10] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M.W. Sukalski. "A network-failure-tolerant message-passing system for terascale clusters". *Int. J. Parallel Program.*, 31(4):285–303, 2003.

[11] S. Kalaiselviand and V. Rajaraman. "A survey of rollback and recovery strategies for computer programs". *IEEE Trans Comput.*, 25:489–510, OCT 2000.

[12] J. L. Kim and T. Park. "An efficient protocol for checkpointing recovery in distributed systems". *IEEE Transactions on Parallel and Distributed Systems*, 4(8):231–240, AUG 1993.

[13] R. Koo and S. Toueg. "Checkpointing and rollback-recovery for distributed systems". *IEEE Transaction on Software Engineering*, SE-13(1):23–31, 1987.

[14] L. Lamport. "Time, clocks and the ordering of events in a distributed system". *Communications of the ACM*, 21(7):558–565, 1978.

[15] A. Lastovetsky. "*Parallel Computing on Heterogeneous Networks*". John Wiley & Sons, 2003.

[16] P.-J. Leu and B. Bhargava. "Concurrent robust checkpointing and recovery in distributed systems". In *Proc. Int. Conf. On Data Engineering*, pages 154–163, 1988.

[17] N. Neves and W. K. Fuchs. "Coordinated checkpointing without direct coordination". In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pages 23–31, SEP 1998.

[18] J. S. Plank, M. Beck, and G. Kingsley. "Compiler-assisted memory exclusion for fast checkpointing". *IEEE Technical Committee on Operating Systems and*

Application Environments, 7(4):10–14, Winter 1995.

[19] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. "Memory exclusion: optimizing the performance of checkpointing systems". *Software Practice and Experience*, 29(2):125–142, 1999.

[20] B. Randell. "System structure for software fault tolerance". *SIGPLAN Not.*, 10(6):437–449, 1975.

[21] S. Rao, L. Alvisi, and H. M. Vin. "Egida: An extensible toolkit for low-overhead fault-tolerance". In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[22] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. "The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing". In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[23] L. Silva and J. Silva. "Global checkpointing for distributed programs". In *Proc. IEEE 11th Symp. on Reliable Distributed Syst.*, pages 155–162, 1992.

[24] G. Stellner. "Cocheck: Checkpointing and process migration for MPI". In *IPPS*, pages 526–531, 1996.

BIOGRAPHIES

Zhao Peng is currently an MSc student in the Computer Science Department at University College Dublin, National University of Ireland. His main research interests are design of algorithms and tools for fault tolerance in heterogeneous networks.

Alexey Lastovetsky received the PhD degree from the Moscow Aviation Institute in 1986, and the Doctor of Science degree from the Russian Academy of Sciences in 1997. He is currently a lecturer in the Computer Science Department at University College Dublin, National University of Ireland. His main research interests are parallel and distributed programming languages and systems for heterogeneous environments. He is a member of IEEE Computer Society.

Appendix.

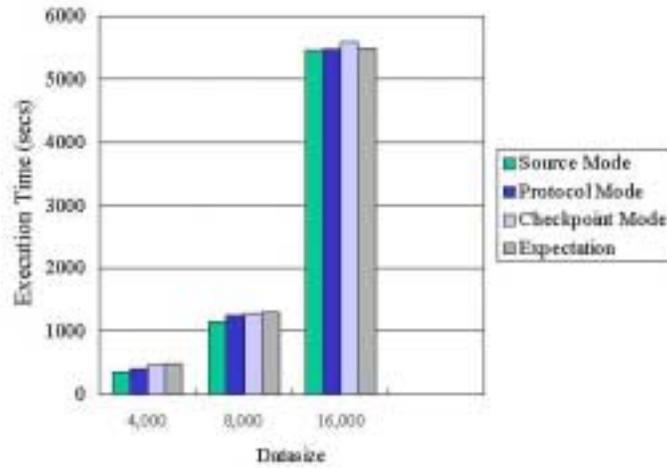


Figure 5. Experiments results of Gauss-Jordan method, in which the x-axis scale is the size of the linear system.

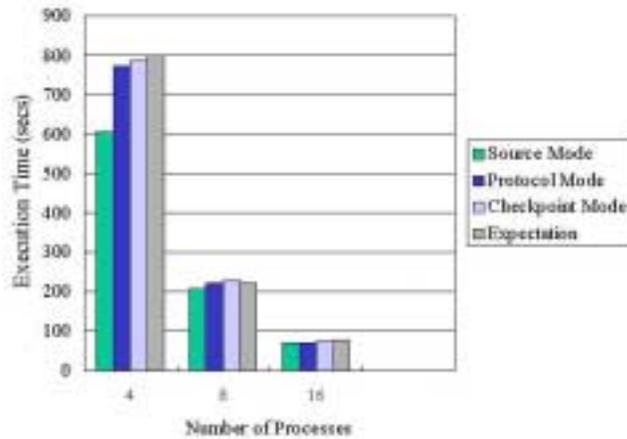


Figure 6. Experiments results of Parallel NeuronSys, in which tests are carried out with different numbers of processes.

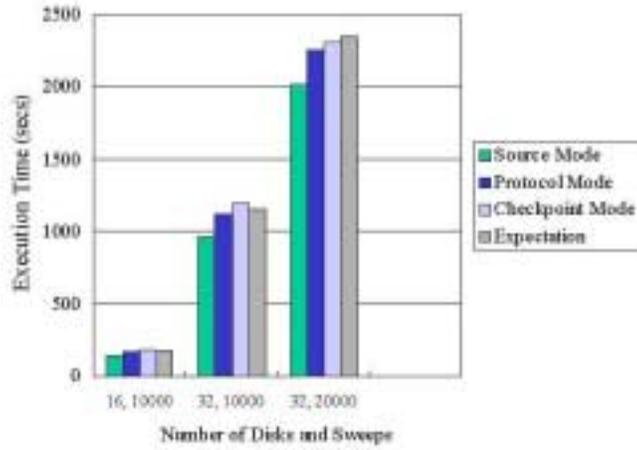


Figure 7. Experiments results of Monte-Carlo Simulation, in which the x-axis scale is the number of disks and sweeps in the simulation.

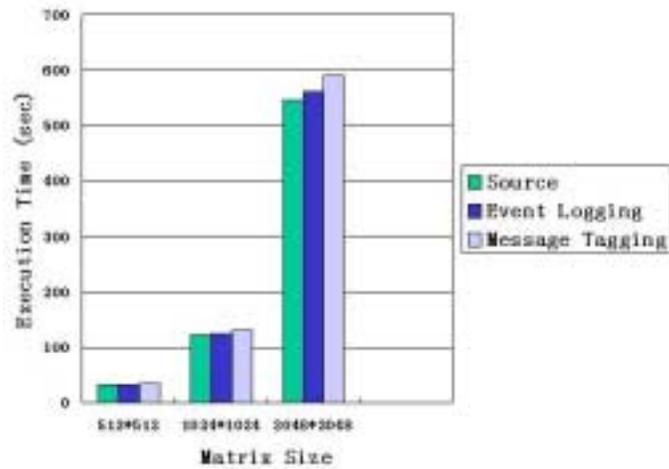


Figure 8. Comparison results of Matrix Multiplication, in which the message sizes in three different tests are 512KB, 2MB and 8MB.