libELC – A Portable Library Enabling Fault Tolerance of MPI Programs in Heterogeneous Environments

Peng Zhao B.Sc.

A thesis submitted for the degree of

M.Sc. in Computer Science

January 2005

Supervisor: Dr. Alexey Lastovetsky

Department of Computer Science

Faculty of Science

National University of Ireland, Dublin

Abstract

It is expected that the future large-scale problem solving environment, especially the Computational Grid, would be more heterogeneous, geographically distributed and independently administrated. The motivation is that the ever increasing deployment of heterogeneous Networks of Computers computers can easify solving the computation-intensive tasks. However, all these factors make the probability of resources failure reach a much higher figure than the traditional scenes. Fault tolerance, as an essential feature for long-running tasks, plays a key role in putting such systems into practice. This thesis investigates the technique to provide portable fault tolerance facility to the MPI programs running in the heterogeneous network.

It is observed that most existing fault tolerance mechanisms for MPI programs are not system-independent. They are either built on some particular platform or more often, implemented as plug-in to some specific MPI distributions. However, given the inherent heterogeneity of such environment, the first and foremost challenge in providing fault tolerance is the software portability. As a solution, this thesis presents a new coordinated checkpoint algorithm: Event Logging, which addresses the application-level non-FIFO message passing problem in Chandy-Lamport algorithm. It implements also libELC, a portable checkpoint/recovery library for C/MPI programs that uses Event Logging for the process coordination. Moreover, compared with the existing checkpoint/recovery systems for MPI, the ability to interoperate with various MPI implementations is considered as a huge advantage of libELC.

Experiments results and analysis presented in this thesis have demonstrated the efficiency of both the Event Logging algorithm and libELC library.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. Alexey Lastovetsky for having provided unfailing support, feedback and ideas over the course of my research. Without his belief in me none of this would have been possible.

I would also like to thank Prof. Li Zheying for introducing me to this topic in my undergraduate year. Thank you also to my proofreaders, Niki, Colm, David and Rem.

I would also like to acknowledge the Eurokom gang for happy memories and much devilment especially my roommates Colm and Niki, for filling in the ‘vast chasms’ in my technical expertise and foreign experience.

Outside of my ‘college cocoon’ there has been a wealth of support from friends and family; my parents, friends, college mates and everybody I met on this wonderful island.

In particular, apologies and special recognition must go to PingPing for (un)successfully feigning interest in the slim guy over the past 2 years, and for her love, patience and support.

Table of Concents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

High performance computing is an issue permanently discussed in computer science. It can be found in a number of fields, such as climate modelling, chemical/nuclear reaction, biologic/gnome analysis and oil exploration. Recent advances in software systems as well as the growing number of available higher-performance computing and networking hardware have made the use of metasystems, namely: networks of workstations, personal computers, and supercomputers as virtual, distributed-memory parallel machines a common approach in solving computation-intensive problems. The combination of distributed nodes within a single system is expected to rapidly replace the dedicated, centralized supercomputers and is expected to become eventually the main stream in the high performance computing community. Moreover, as the technology like Grid Computing suggests, the future of high performance computing would be more common and popular in heterogeneous network of computers.

The heterogeneous architectures and operating system platforms, working within a single high performance computing system, give rise to number of problems that are not present in the traditional homogenous systems. The complexity of both (a) varying architectural features, such as data representation and instruction sets, and (b) varying operating system features, such as process management and communication interfaces, must be masked from the application programmer. Further heterogeneity complicates existing problems in parallel and distributed systems. For example, data partition may depend on several factors: processor speed and architecture, operating system and network bandwidth. Despite the complexity and challenges involved in heterogeneous distributed computing it remains an active and promising area of research because it promises increased performance both by the use of a larger hardware and by mapping sub-tasks of a computation to the most appropriate available hardware, described in mpC [1] and HMPI [2].

However, as a common problem in network of computers, both the heterogeneity and independent administration of computation node increase the probability of failures. Unlike the RPC-based distributed system, the parallel processes that disperse in networks of computers are usually tightly coupled. If no fault tolerance is provided, then when one or several processes fail, the rest cannot survive to continue and the whole program crashes. In this sense, what is needed is a technique that would enable a system to perform fault tolerant procedures that can continue to execute even in the presence of a fault. Therefore, support for fault tolerance is an essential feature of heterogeneous networks of computers, because the execution time of parallel programs is long.

This thesis focuses on fault tolerance in heterogeneous networks of computers, providing the portable checkpoint/recovery facility to the MPI programs running in such environment. The solution suggested here is an innovative coordination checkpoint algorithm: Event Logging. This algorithm is designed and implemented together with corresponding software library libELC. Finally, the thesis reports experiment results that demonstrate the efficiency of Event Logging and libELC.

1.2 Heterogeneous Network of Computers

Networks of Computers (NoC) is the most general and popular architecture for parallel computing nowadays. Unlike the Symmetric Multiprocessors (SMP), NoC can be included in the volume of Distributed Memory Multiprocessors. The Distributed Memory Multiprocessors differentiate from Symmetric Multiprocessors by two major features. The first is that the former consist of a set of independent processors and the second is that they share no global memory space. Rather, this type of processors maintains local memory and is interconnected by the network.

Similar with other subsets of the Distributed Memory Multiprocessors (MPP), in NoC the processors communicate by passing messages. However, different from the traditional MPP, NoC is distributed, heterogeneous and autonomous.

	MPP	NoC
Equipment	*As MPP is specially manufactured for high performance computing, it adopts the homogeneous design: the identical system architecture, similar hardware performance and single image software. Unlike MPP, a typical NoC is a naturally evolving collection of computers.*	*Generally, a NoC is composed of various architectures: PCs, workstations, SMP servers, and even MPP supercomputers and clusters. As the result, the performance of these architectures varies significantly. NoC is heterogeneous.*
Deployment	*MPP is typically located in a small area, such as in a computing centre or research lab. The processors are interconnected by special high bandwidth networks, like Myrinet*	*NoC often consists of nodes widely distributed and connected with mixed network equipments.*
Admin	*MPP is often administrated by a small-dedicated group and dedicated for the high performance tasks.*	*NoCs are general-purpose computer systems, each node of which is administrated independently by the owner. This leads unpredictable during a job's execution.*

Table 1. Comparison of MPP and NoC

Table 1 is a detailed comparison between NoC with MPP. Compared with MPP, NoC has one major advantage — **scalability**. It is one of the goals of NoC to enable any computing device to join the pool. Further, the cost of building a NoC is much lower than that of building other parallel systems of similar scale. There is, however a downside of NoC, heterogeneity. The heterogeneity is the most significant and inherent feature of NoC. There are three aspects, in which the heterogeneity is evaluated.

- **Hardware**: *In hardware, heterogeneity is due to the variety of existing architectures and hardware standards. This problem is complicated further by the various hardware vendors. The term "compatibility of hardware" can be in*

certain cases vague and in other cases confusing. For instance, sometimes two devices may be interchangeable and compatible from the point of view of the user, but they could be built on toally different architectures. In this sense, writing portable codes across various machines is definitely the most necessary requirement for the heterogeneous NoCs.

- **Software**: *Software heterogeneity is evaluated in terms of Interoperability. Usually, interoperability problems occur when different implementations of the same software standard exist. An example, close to the topic of this thesis, is MPI. Currently there are two main versions of MPI: MPICH* [4] *and LAM/MPI* [5]. *As a well-known problem, these two distributions lack the capacity of interoperation, which requires all joining processes to be running with the same version: either MPICH or LAM/MPI. In this sense, a portable MPI program should not rely on any facility provided by the underlying implementation. Otherwise the program can be launched only at a node with the specified MPI version.*

- **Performance**: *The third heterogeneity of NoC results from the two previous aspects. In general, differences between hardware architecture and software implementation has lead to significant performance fluctuation among the machines in a network. Moreover, since the processors communicate by passing messages, the networks connecting them turn into the key factor affecting the overall system performance. This results in a totally different problem partition scheme for NoCs. In the homogeneous, regular environment such as MPP, the problem is usually evenly distributed. However in NoCs, the data should be partitioned proportional to the individual processor's speed and the network bandwidth among them* [1]. *So, in order to maximally utilize the resources, a good NoC-oriented parallel program should take into account the irregular machine performance.*

The work in this thesis is motivated by this analysis. The review presented here shows that heterogeneity is the most urgent and important issue to be addressed before NoCs can be put into general practice.

1.3 Fault Tolerance of Heterogeneous Network of Computers

As a result of heterogeneities, the probability of resources failure is much greater for a NoC. The primary reason is that there are different levels of reliability for each node. For example, commercial supercomputers are usually highly robust, while PCs are quite unstable. However to a parallel program running in NoC, the overall failure probability is decided by the failure probability of the most unreliable node. A second reason is that the network communication between separate processors in a NoC is unstable. Since the network equipment used in a NoC is not dedicatedly manufactured and maintained for high performance computing, communication failures occur often. Most commonly, communication faults cause the programs to crash. However there are some cases in which processes will be blocked by communication operations as long as the connection is faulty. Third reason is that since the machines in a NoC are administrated independently, any individual node might disconnect unexpectedly from the network. The machine could be (a) switched off, (b) rebooted (c) or rescheduled to a job with higher priority. As a result, the machine is no longer a working node in a NoC.

All the above arguments make fault tolerance a highly desirable feature for long running parallel programs on NoCs. There are four components that comprise fault tolerance: fault detection, fault location, fault masking and fault recovery. Fault tolerance can be provided to the parallel computing at three different levels [3]: hardware level, architecture level and software level. In the hardware and architecture levels, importance is given to fault detection and fault location. In the software level, fault tolerance policy usually emphasizes on the fault masking and fault recovery. By comparison, it is easier and more cost effective to provide software fault tolerance solutions at the software level than hardware solutions.

1.3.1 System Model

So far, it was argued that the aim of this thesis is to design and implement a software-level fault tolerant system for parallel programs in NoCs. However, the need to specify a parallel programming model is the target here as well as the type of faults

that are going to be the focus of this work. The following sections elaborate on these two aspects.

- **Parallel Programming Model**

First, we focus on the MPI programs [6]. Although there are some alternatives, MPI has the distinct scalability advantage with regards to NoCs. Conceptually, a MPI program can be thought as a set of independent processes running in separate address spaces. Processes are hosted on different machines, but are coupled by passing messages. MPI is built on the assumption that communication takes place within a known group of processes. Each group is assigned an ID. Each process within a group is also assigned a local ID. So, a pair of <*GroupID, ProcessID*> uniquely identifies a process. Messages are sent and received by the source and target process through explicit function calls. As to each individual process, the execution is defined by a sequence of *state intervals*, each started by a nondeterministic event. Execution during every state interval is deterministic, such that if a process starts from the same state and is subjected to the same nondeterministic events at the same locations within the execution, it will always yield the same output.

However, a process may fail. Generally there are two common types of failures in the community of parallel/distributed systems: **Fail-stop failure** [7] and **Byzantine failure** [8]. In the **Fail-stop failure** model, a faulty process loses its volatile state and stops responding to the rest of the system in such a way that its halting can be detected by other processes. In comparison, **Byzantine failure** incurs more serious problems, because the fault process may still communicate with the others and as a result it is possible to send malicious messages.

In this thesis, we concentrate on the **Fail-stop failure** of MPI programs. The work here is based on two assumptions. The first is that processes have access to a stable storage device that survives faults, such that state information saved on this device during failure-free execution is valid through process failures.

The second assumption is the existence of a secure and reliable transportation layer for the message passing. “Reliable” means that the communication latency is arbitrary but finite. In other words, a reliable communication layer guarantees that any message

will be delivered to the destination once it is sent out. This feature is generally supported by the network layer. Also "reliable" implies that correct semantics of message passing must be guaranteed by the underlying communication facility. In the context of MPI, this applies two rules:

1. *No message would be altered during the communication, which includes both the message envelope and the message content.*
2. *No message delivery will be dropped or duplicated during the transfer. In other words, once a message is sent out, the underlying communication layer guarantees the eventual delivery of that message.*

Any faults within such a communication layer are hidden from the upper-level MPI program, by either the network itself or by the MPI implementation [9]. These assumptions release our work from the worries about the communication faults, while helping us to concentrate on the more general software-level fault tolerance.

1.3.2 Design Goal

Given a heterogeneous network, the following sections list several criteria for designing the fault tolerance facility.

First of all, the fault tolerance mechanism must be portable. Given an environment comprised of various architectures and platforms, a system-independent solution has an unparalleled advantage. More, we emphasize that the portability also applies to the software. For MPI, we note that the fault tolerance approach should be built on top of the standard, which in particular makes no assumption about the underlying MPI implementations. The ability to interoperate with various MPI distributions is considered a huge benefit.

Secondly, the portability should not result in too great a performance penalty. In other words, the solution must provide similar performance to other lower level approaches.

Third, from the point of view of the end users, they may not want to deal with the details of the underlying fault tolerance mechanism. Therefore, the proposed design

should be transparent to the users. However, certain options should be exposed to allow advanced tuning of the fault tolerance.

1.3.3 Redundancy and Replication

Physical redundancy is the most straightforward and widely used fault tolerance technique [10]. By adding extra components, the fault can be made transparent to the rest of the system. The malfunctioned element is replaced with one of the substitutions. Usually one might think that the redundancy is implemented by means of hardware. However, software can also provide the redundancy, more often called **replication** [11].

The key technique of software **replication** is group membership. In the context of MPI programs, each process is associated with a group of backups. Upon the occurrence of a fault in a process, the backup group uses some election algorithm to choose the replacement. Therefore, if a MPI program consists of *N* processes and the failure probability of the i^{th} process is P_i $(0 \le i<N)$, without replication the overall survival probability of the MPI program is calculated by: $\prod_{i=0}^{N-1}(1-P_i)$. However, suppose that the i^{th} MPI process has B_i replications, which has the same failure probability P_i, then the MPI program's survival probability is changed to $\prod_{i=0}^{N-1}(1-P_i^{B_i})$.

Obviously, the improvement is

$$\prod_{i=0}^{N-1}(1-P_i^{B_i}) / \prod_{i=0}^{N-1}(1-P_i)$$

$$= \prod_{i=0}^{N-1}(1-P_i^{B_i}) /(1-P_i)$$

$$= \prod_{i=0}^{N-1}(\sum_{j=0}^{B_i-1} C_{B_i}^{j} P_i^{j}) \quad (C_{B_i}^{j} \text{ denotes the } j\text{-combination of } B_i)$$

Generally speaking, software replication gives a MPI program a higher probability of survival [12]. However, the downside of this type of fault tolerance technique is the resource consumption problem. In a minimal case, *N* process MPI program needs to keep *2N* processes running, while each MPI process maintains only one replication. In

particular a robust system built on software replication will consume a big amount of extra resources.

Furthermore, to implement the replication on message passing programs, a key technique is *Atomic Multicast*. Atomic multicast guarantees that a process always synchronizes with its backups. However, the synchronization incurs with significant overhead to the message-passing program. Currently, high performance MPI implementations are literally counting every cycle in an attempt to reduce latency as much as possible. As a result, adding overhead is unacceptable except for critical tasks which emphasize more on the system availability, rather than the parallel computationally extensive jobs which focus on minimizing the problem solving time.

1.3.4 Program Resilience

The research on hardware redundancy and software replication overlooks completely faults which are hidden from the user. As a rule, when a failure occurs the system tries to implicitly heal itself. The user need not and usually cannot detect and manage the failures. Although ultimately this process needs to be made transparent, the tradeoffs could be a high performance overhead and considerable cost. In this case, a lightweight approach, which is named **Program Resilience** [13], is taken to release the performance burden.

The program resilience approach emphasizes the survival of a partially malfunctioning program. However, it does not guarantee the correct semantics of the program execution. In this case, the program is able to continue running if failure happens. The task of adapting or restoring program state is left to the user. The user has the flexibility to choose different strategies to manage the errors in different failure scenarios. Therefore the program resilience approach has less overhead compared to the software replication.

A well-known example of program resilience in the context of MPI is taken in the FT-MPI library [14]. In FT-MPI, the state of a communicator is extended from {VALID, INVALID} to {OK, PROBLEM, FAILED}; while the process state is extended from the simple {OK, FAILED} to {OK, UNAVAILABLE, JOINING, FAILED}. The

enriched states give the user the opportunity to detect a failure within the time it runs. When the user detects a failure s/he may take different actions to repair a broken communicator: Shrink, Blank, Rebuild or Abort. Hence the program is able to survive through the faults.

However, in the very advantage of this approach is rooted its disadvantage: in a program resilience system, nothing is transparent. Therefore as a requirement, the program must be able to adapt itself to various faults. However in many scenarios, such an adaptive algorithm is very difficult to design. Consider an example of a SPMD program which consists of N processes, and each process holds a unique dataset. The data partition scheme determines that the failure of any process will result in the loss of this unique dataset. In such a case, execution cannot simply continue upon failures, even if the application may survive with the aid of resilience. In simple cases which are data independent, a solution may re-allocate the lost dataset to the remaining N-1 process. The N-1 will repeat the computation of the lost dataset. Unfortunately, these are isolated examples. In the more common cases there is a dependency between different datasets and there is not an easy solution to restore the state that executed before failures.

Also, it must be noted here that not all types of faults can be tolerated in the program resilience approach. Consider the example of an MPI program working in the Master/Slave pattern. As opposed to a slave process, master process failure cannot be easily ignored. Also in many cases there is a lower bound of the numbers for living processes. Above the value, program resilience may cut down the malfunctioned process to survive the execution. However, when this limit is reached, the program cannot simply resize itself so as to continue. Some other fault tolerance mechanisms, like employing new process, are needed as a backup solution. In this sense, a program resilience system cannot tolerate all faults that could occur.

To conclude, although program resilience has the huge advantage of less performance overhead and is a promising concept in the world of fault tolerance, it is far from its mature stage.

1.3.5 Checkpoint and Rollback Recovery

Fundamental to fault tolerance is recovery from an error. The whole idea behind the rollback recovery is to replay the failed program from some pre-saved points, where the state that executed before the failure can be restored. In this sense, there is no need to restart the failed program from its very beginning, but resume from some intermediate state. Resuming the execution from such intermediate points does reduce the execution time lost due to either software or hardware failures. To do so, it is necessary to create a checkpoint to capture the state of a running program and output the checkpoint onto a stable storage from time to time. To restore a failed program, the program's execution state is reloaded from the physical checkpoint file. Once the loading completes, the process is recovered and ready to resume the execution.

The significant advangtage of checkpoint/recovery is that it is a general method that can handle most kinds of failures. As result, the checkpoint/recovery has become the mainstream mechanism to provide fault tolerance for MPI programs. A substantial body of research demonstrates the utility and desirability of such a mechanism. The following is an overview of the applications that implement this method.

CoCheck is one of the earliest efforts to provide complete user-transparent checkpoint/recovery service to message passing application [15]. CoCheck follows the coordinated checkpoint flavour, using the Chandy-Lamport algorithm [16] for process coordination. It is noted that CoCheck is built on its own MPI implementation, known tuMPI. LAM/MPI [56] is a widely used MPI implementation. LAM/MPI has its built-in checkpoint/recovery functionality, which also uses the Chandy-Lamport algorithm for the coordinated checkpoint. Clip [17] is a user level coordinated checkpoint library dedicated to the Intel Paragon Systems. This library can be linked to MPI programs to provide semi-transparent checkpoint. The users need to explicitly invoke the checkpoint, but are not expected to manage the recovery. MPI-FT [18] is another fault tolerance library based on LAM/MPI, which uses the uncoordinated checkpoint mechanism with the aid of pessimistic message logging. MPICH-V2 [19] is one of the most recent fault tolerance solutions for the well established MPICH distribution. MPICH-V2 implements causal message logging, while using uncoordinated checkpointing to reduce the execution time lost. MPICH-

GF [20] is another checkpoint approach, which provides multi-protocol checkpointing to MPICH-G2 [21].

It is noted that a common of all these systems is that none is built on top of MPI. They are either platform-dependent or more often, implemented as plug-in for some specific MPI version. The only MPI-implementation-independent approach we notice is the C^3 system [22]. The merit of C^3 is that this system can interoperate with any MPI distribution. Our work shares the objectives of C^3. However we are dedicated to design a portable checkpoint/recovery facility for a heterogeneous network, which has not been addressed in C^3.

1.4 Conclusion and Thesis Outline

The purpose of this chapter is to identify the goal of the work presented here. It also introduces the main concepts and issues discussed further on. This is achieved by briefly examining the features of a heterogeneous NoC. Further, it identifies the needs and requirements of the fault tolerance for MPI programs in a given heterogeneous environment. Moreover, it introduces the three candidates for providing the fault tolerance facility: software replication, program resilience and checkpoint/recovery. It also highlights the advantages of the checkpoint/rollback-recovery approach over the other two, selects it as the method that is going to be used in this work.

The following chapters present our approach to provide checkpoint/recovery facility for an MPI program running on a heterogeneous network of computers. Chapter 2 defines the condition for creating checkpoints for MPI programs and examines the main checkpoint models; Chapter 3 presents a new coordinated checkpoint algorithm, Event Logging, which addresses the application-level coordination problem of the Chandy-Lamport algorithm; Chapter 4 describes libELC, an application-level checkpoint/recovery library for C/MPI program, using the Event Logging algorithm for the process coordination; Chapter 5 gives the performance experiments and analysis of Event Logging algorithm and libELC library; Chapter 6 concludes the work and list several routes for future development of the work.

Chapter 2

Checkpoint and Recovery for MPI

This chapter introduces the major approaches that are used in the literature. It is organized as follows: Section 2.1 defines the sufficient and necessary condition for creating checkpoints for MPI programs; Section 2.2 -- 2.4 introduce three different protocols for checkpointing MPI programs: Coordinated, Uncoordinated and Communication-induced, and finally Section 2.5 examines the advantage and the disadvantage of these three checkpoint models.

2.1 Creating Valid Recovery Line for fMPI

In order to create checkpoints for a parallel program, the first step is to create checkpoints for each individual process. Here are the types of checkpoints that are necessary for each individual process:

- Definition 1**: Local Checkpoint**
  The checkpoint of each separate process is called a Local Checkpoint.

Upon recovering, the parallel program state is restored from a set of local checkpoints, which is called a recovery line.

- Definition 2: **Recovery Line**
  A Recovery Line is a set of each process' local checkpoint that can be used to revert to a previous execution state of the parallel program. A Recovery Line is also called a Distributed Snapshot or Global Checkpoint.

However the recovery line of a MPI program is not as simple as merely performing a collection of the local checkpoints of each participating process. Since MPI employs messages in order to do the communication among multiprocesses, these messages are part of the state of the running program. As any communication has latency, there might be some messages in transit at the time that the individual process's state is

saved. Therefore, the checkpointing algorithm of MPI must capture each state of the communication.

Generally, there exist three types of messages in MPI: **intra message**, **in-transit message** and **orphan message**. Lamport proposed a relation called **Happens Before Relation** to indicate the partial order of the events in a distributed system [23]. This is an irreflexive, antisymmetric and transitive relation that can be applied to define these messages.

- Definition 3: **Happens Before Relation**
  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 the receiving event of M, then* $A \rightarrow B$.
  3. *If not* $(A \rightarrow B)$ *and not* $(B \rightarrow A)$*, then A and B are concurrent events.*

Given the definition of **Happens Before Relation**, the three types of messages can be defined as follows:

- Definition 4:
  1. **Intra Message***:* $CKPT \rightarrow Send \rightarrow CKPT'$ *and* $CKPT \rightarrow Recv \rightarrow CKPT'$
  2. **In-transit Message***:* $Send \rightarrow CKPT$ *and* $CKPT \rightarrow Recv$
  3. **Orphan Message***:* $CKPT \rightarrow Send$ *and* $Recv \rightarrow CKPT$

Where *CKPT* and *CKPT'* are two successive local checkpoints of a process.

Among these messages, intra message is harmless, because the passing of an intra message is completed upon checkpoint. No intra messages will exist in the communication channel upon checkpointing so the recovery line is preserved. However, the other two types: in-transit and orphan message can impose treats on the processes. If a failure occurs after the system has finished the recovery line $\{C_1, C_2\}$, the execution of the program is then restarted from that point. However, there are complications that can occur. Let take an example. If process P_1 has sent the message m_1 to P_2 without saving the communication state, and then the local checkpoint of P_2

is taken before it receives m_1. This can lead to the problem that P_2 will wait for m_1 after recovery but m_1 may be lost or discarded by the network during the program's failure. In this case P_1 would never send m_1 again. This problem is caused by in-transit message and is referred to as ***Unrecoverable*** (See Figure 1).

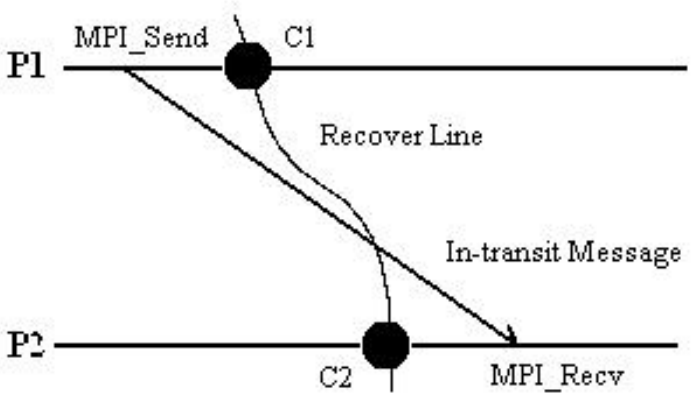

Figure 1. In-transit Message

Moreover, if as shown in Fig. 2, m_2 is sent after P_1's local checkpoint and is received before P_2's checkpoint, then upon recovery P_1 will re-send m_2 a message that has actually been received and saved in P_2's checkpoint. Although the execution of message sending can be recovered with the existence of orphan message, such a message does not only waste the buffer space, but also breaks the communication semantics. This type of error is referred to as ***Inconsistency***.

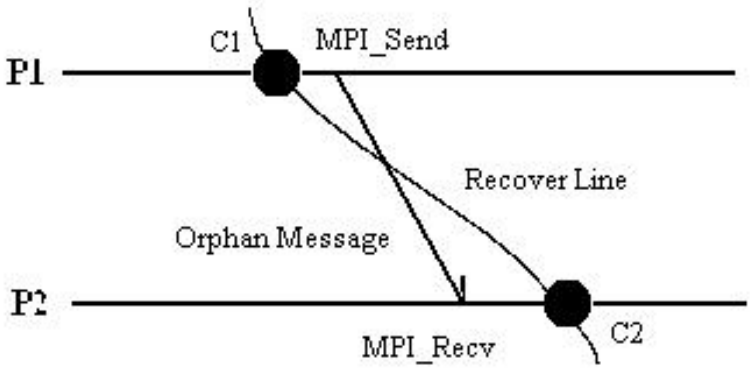

Figure 2. Orphan Message

Taking into account the arguments above, two conditions are prerequisite in order to achieve a valid recovery line, they are:

1. **Recoverable**: *no existence of in-transit message or the message can be regenerated after recovery.*
2. **Consistent**: *no existence of orphan message or the message can be eliminated after recovery.*

Given the condition to set up a valid recovery line for MPI programs, a checkpoint/recovery system can be categorized into one of three: ***Coordinated Checkpoint, Uncoordinated Checkpoint*** and ***Communication-induced Checkpoint***. The criterion for classification is based on the way in which a specific system manages the in-transit and orphan messages. Therefore, a coordinated effort is required in order for all processes to be synchronized upon creating the recovery line. There are differences in the approaches. The uncoordinated systems give processes the maximal autonomy in deciding when to create local checkpoints; while in a communication-induced checkpoint protocol, processes are allowed to create the local checkpoints independently. However sometimes additional checkpoints are forced in order to guarantee the eventual process of the recovery line.

2.2 Coordinated Checkpoint

The coordinated checkpoint is the most straightforward checkpoint mechanism for creating a recovery line for message passing programs. It orchestrates processes to create checkpoints to ensure that the communication channels are drained before the recovery line is setup. Similar with the checkpointing, the coordinated approach to failure of one process involves all other surviving processes to rollback to the latest recovery line.

2.2.1 Blocking Coordination

In its simplest form the coordinated checkpoint processes are synchronized before the local checkpoints in order to ensure a clean communication channel [24]. A coordinator broadcasts for every process a checkpoint request. When a process receives such a request, it stops its execution, flushes its communication channels, takes a local checkpoint, and sends an acknowledgment message back to the coordinator. After that the coordinator collects acknowledgments from all processes,

and broadcasts a commit message to complete the two-phase checkpoint protocol. Upon receiving the commit message, each process marks its local checkpoint as a new recovery line. Then the process resumes execution and exchanges messages with other processes. Although such an approach is implemented in certain programming models [25], the two-phase blocking operation incurs significant overheads during the failure free execution. This conflicts with the goal to achieve high performance.

Another type of coordinated checkpoint protocols is the ***Time-based Coordinated Checkpoint*** [26-31]. This method follows the blocking flavour, but requires no explicit barrier operation. In this approach, the synchronization is not made through explicit blocking, instead, all inter-process communication are cached around the checkpoint time. This allows ensuring that when the recovery line is formed no message is left in the communication channel. The time-based checkpoint protocol coordinates the processes by means of a clock. However, the existence of time deviation does not allow the clock of distributed processes to be accurately synchronized. Given a real time interval $[T_{start}, T_{end}]$ and the clock drift rate ρ, the clock time of any process will be in the interval of $[(1-\rho)(T_{end}-T_{start}), (1+\rho)(T_{end}-T_{start})]$, if the processes are launched at the exact time T_{start}. In this case, the maximal clock deviation of two distributed processes is $2\rho(T_{end}-T_{start})$. Furthermore, the checkpoint protocol assumes the existence of a maximal and minimal message delivery latency: T_{max}, T_{min}. As a result, if all processes start from the time T_{start} and agree to create a recovery line at T_{end}, the clean communication channel is taken as shown in Figure 3.):

1. *Cache all message passing from* $T_{end} - 2\rho(T_{end} - T_{start}) - T_{max}$ *;*

2. *Resume the message passing at* $T_{end} + 2\rho(T_{end} - T_{start}) - T_{min}$ *;*

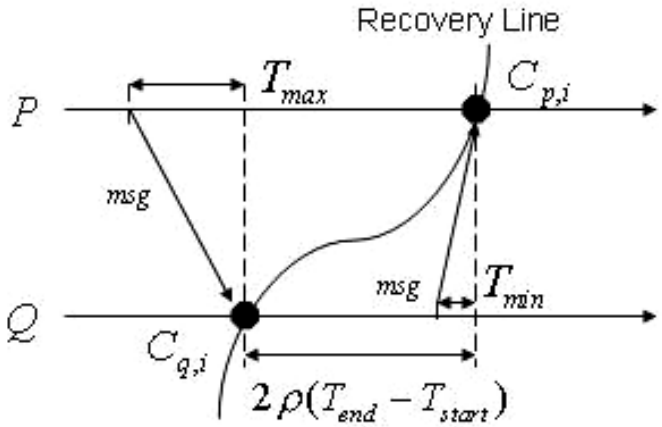

Figure 3. Time-based Coordination

Although the time-based coordination avoids the considerable overheads caused by the global barrier operation, this checkpoint protocol suffers from its scalability. In a small network, the algorithm works pretty well. However, when the network scales are taken into account, the clock-drift rate between two distributed processes, as well as the maximal message delivery latency, increase dramatically. It can even reach a degree in which the message passing caching period would be unacceptable. More important, in a heterogeneous network, it is actually not possible to accurately measure parameters such as clock-drift rate as well as maximal and minimal communication latency.

2.2.2 Non-Blocking Coordination with Chandy-Lamport Algorithm

The performance overheads in of the blocking coordination are difficult to avoid. That is why in practice a non-blocking scheme is preferred. A non-blocking scheme differs from the blocking scheme, because the former allows the process to resume its execution as soon as it finishes the local checkpoint. In the non-blocking scheme, although all processes are involved in creating the recovery line, processes are not firmly synchronized. In this sense, in-transit and orphan messages may exist in the communication channel at the time the local checkpoint is taken. However processes employ some external facilities to manage these messages, and thus to guarantee that the recovery line is consistent and recoverable. As a consequence, the non-blocking

coordination would have significant advantages on the checkpoint overheads, especially given the scalability concern.

Among the approaches proposed in order to manage the in-transit and orphan message, the Chandy-Lamport algorithm [16] is the most widely used algorithm. It is observed that it performs best when designing non-blocking coordinated checkpoint systems for message passing programs. We note that most existing checkpoint/recovery systems, which are built for message passing systems, employ the Chandy-Lamport algorithm as a foundation. Further, most of the algorithms that are proposed for checkpointing MPI programs by other researchers [26, 27, 32-37-58, 73], can be derived by relaxing various assumptions and by modifying the way each step is carried out.

As per Chandy-Lamport's model, 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 channel.*
3. *The Latency of communication is finite and the communication is reliable.*
4. *The message passing behaves in the FIFO manner.*

The algorithm can be summarized as follows. Let IN_x and OUT_x be the number of incoming and outgoing communication channels respectively, which process *x* holds.

1. *A process p is selected to initiate a new global checkpoint. P first creates a local checkpoint. For i=1 to* OUT_p *, p broadcasts a marker message along the outgoing channel i and then resumes its execution.*
2. *To any other processes: upon detecting the first marker message, a process q immediately creates a local checkpoint, and sends out its own markers along all outgoing communication channels it holds. After that, it resumes execution.*
3. *To each process x: for i=1 to* IN_x*, x logs the messages from the incoming channel i as in-transit messages, until a marker message is received along channel i. Once the process x receives marker messages along all incoming channels, it marks its logging completed.*

4. *When all processes finish logging, the recovery line is formed.*

It is noted that the Chandy-Lamport algorithm is naturally immune to orphan messages since the FIFO property guarantees marker messages always arrive and are detected before all post-checkpoint messages. So the recovery procedure of the Chandy-Lamport algorithm is simply to rollback all processes to their latest local checkpoints, and replay the in-transit messages at the receiver side.

At the time when the initiating process informs the rest of the processes about the new checkpoint, the Chandy-Lamport algorithm broadcasts marker messages along all communication channels. However it is unnecessary to send marker messages along every channel. Some marker messages can be safely eliminated along channels in which there is no message exchanged between the previous and the current checkpoint intervals [32-34]. The attempts to achieve this simplification result in ***Selective Checkpoint*** [35, 36] approach. In it not every process participates in the global checkpoint. Instead, a group of processes are selected to create a partial recovery line. After that this line can be safely patched onto the latest global checkpoint. This advances significantly the whole recovery line. Apart from reducing the number of marker messages, another benefit of selective checkpoint is the failure recovery cost. In the coordinated checkpoint model, the failure of any number of processes requires the rollback of all processes. However with the selective checkpoint, the rollback does not have to involve every process. If the failed process can be found in a partial recovery line, only the processes that participated in creating the partial recovery line are rolled back.

2.2.3 Summary of Coordinated Checkpoint

Being the most straightforward checkpoint/recovery protocol for message passing systems, the coordinated checkpoint protocol has the inherent advantage of its simplicity. This advantage is reflected both in the checkpointing and in the recovery. A disadvantage to this protocol is its performance. The non-blocking coordination introduces negligible overheads compared to a program, which saves its state during the failure free run. Although there are questions raised regarding the scalability of a coordinated checkpoint protocol, it has been demonstrated by [76] that it is the most practical approach given all performance considerations are balanced. As a

consequence, it is observed that most existing checkpoint/recovery systems for message passing programs either are directly built on the coordinated checkpoint model (the non-blocking flavour), or that they implement the coordinated checkpoint as a secondary fault tolerance mechanism.

2.3 Uncoordinated Checkpoint and Message Logging

Unlike the coordinated protocol which orchestrates all processes to setup the recovery line, the uncoordinated scheme allows processes to create the local checkpoint independently. The main potential is that each process may choose the most appropriate time to trigger its local checkpoint, preferably at the point of the process that has the least amount of data for checkpointing. Another benefit of the uncoordinated checkpoint is that the recovery procedure is also uncoordinated. In the coordinated checkpoint once a failure occurs, all participating processes need to rollback. However in the uncoordinated mode, when a failure occurs only the failed processes need to rollback to the latest local checkpoint and replay the execution so there is no need to interrupt other processes. Compared with the coordinated mechanism, this form promises more flexibility. The reason is that it introduces less checkpoint overheads than the coordinated form since it eliminates the need to exchange coordination messages.

However, usage of the uncoordinated checkpoint comes at a cost. First of all, the direct result of the lack of coordination is the possible danger of in-transit and orphan messages. Therefore an uncoordinated checkpoint system as a rule does not create a valid recovery line. The recovery line differs from the coordinated form in its usage of uncoordinated checkpoints. In this model the recovery line is actually constructed during the recovery. The system finds a set of local checkpoints that can be used for the rollback. However, this poses several problems. The first one is that every process needs to maintain multiple local checkpoints, rather than only the latest one. The second is that some of the local checkpoints may become redundant if they are not included in any recovery line and thus a garbage collection module is need. And finally, the third problem is that even though processes keep as many local checkpoints as they have created, they may still suffer from Domino Effect [37].

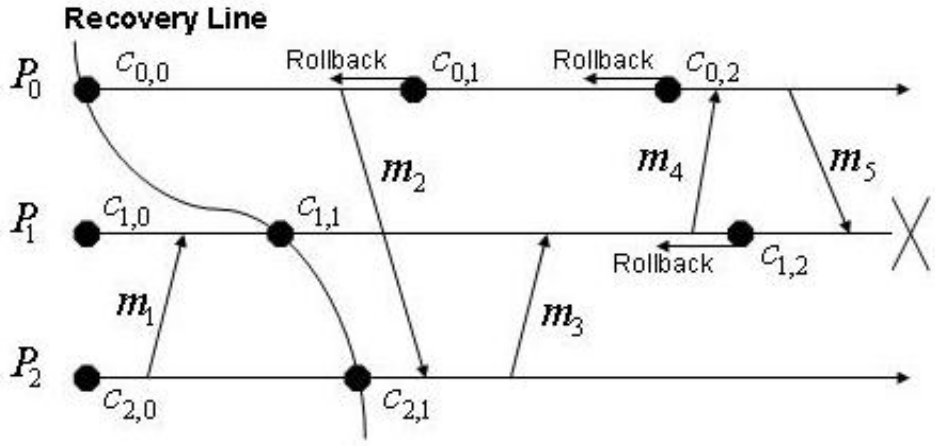

Figure 4. Domino Effect

Domino Effect is formulated as a cascaded rollback, which may continue and eventually causes the program to restart from its very beginning. Consider the example given in Figure 4, suppose that process P_1 fails after it has completed the local checkpoint $c_{1,2}$. Since processes create their local checkpoints uncoordinatedly, there is no naturally formed recovery line. To recover P_1, P_0 has to restart the execution from $c_{0,2}$ to receive message m_5. The rollback of P_0 further forces P_1 to step back to $c_{1,1}$ in order to replay message m_4, which also propagates to the rollback of P_2 to $c_{2,1}$ due to m_3. Then P_0 is rolled back to its beginning $c_{0,0}$, since m_2 is invalidated for the rollback of P_2.

To any checkpoint/recovery system, the possibility of the Domino Effect is intolerable and needs to be completely eliminated. There is extensive research focused on studying message logging [38]. Message logging relies on the *piecewise deterministic* (*PWD*) assumption [39]. Given a message passing program, a deterministic event refers to the receiving of an incoming message; while a nondeterministic event denotes a message sending operation. To an MPI process, the incoming message is the necessary condition for reproducing its outgoing message. Outgoing messages can be replayed only if the process gets the same incoming message during the recovery in the exactly same sequence as before the failure. Under the PWD assumption, the message logging protocol identifies all the nondeterministic events executed by each process. After that for each such event it logs a determinant that contains all

information necessary to replay the event. Thus the protocols can recover a failed process and replay its execution as it occurred before the failure.

From the way in which the message logging is implemented, a specific system can be categorized into one of the three flavours: ***Pessimistic Message Logging, Optimistic Message Logging*** and ***Causal Message Logging***.

2.3.1 Pessimistic Message Logging

Pessimistic Message Logging [40] presents the most rigid behaviour in the guarantee for the correctness of recovery, since it does not occupy time during the period when a failure occurs. In contrast, the pessimistic model is founded on the presumption that faults can occur at any time, regardless of whether an incoming message has been written or not onto the disk. Therefore it adopts a strategy in which every incoming message has to be logged onto the disk storage before it can be received. Hence, there is no need to calculate global recovery lines in the pessimistic model, since the failed process is able to reproduce its execution using the logged messages. However the rigid behaviour introduces considerable overheads. These are due to the synchronous logging operation during failure free execution. The tradeoff of pessimistic message logging becomes even more significant in communication-intensive programs.

To reduce the logging overhead, a common solution is to use ***Sender-based Message Logging*** [41]. The Send-based Message Logging has the advantage to its counterpart, ***Receiver-based Message Logging***, that all messages are logged in the sender's volatile memory rather saved in the receiver's stable storage. However, a disadvantage is that Sender-based Message Logging protocol tolerates only a single failure of a receiving process [42]. If both the sender and receiver have failed at the same time, the message necessary for the recovery of the receiver is logged in the sender's volatile memory, and, will be therefore lost. In such case, the rollback must be propagated in order to find a valid recovery line. However this solution can cause the Domino Effect.

Another optimization is to defer logging the incoming message until the process sends some outgoing messages [43]. This solution relaxes the pessimistic logging by

allowing the process to receive messages that have yet to be logged. Thus message logging and receiving are not performed in one atomic operation. This reduces overhead because several messages can be logged in one operation, reducing the frequency of synchronous access to stable storage.

2.3.2 Optimistic Message Logging

In the optimistic message logging, the protocol makes the "optimistic" assumption that no failure would occur before the messages are logged onto the stable storage [39]. In other words, the logging operation is asynchronous with the message passing. Messages are first saved in the memory (at either the sender or the receiver side) and then flushed periodically onto the stable storage. In terms of the logging overhead, the Optimistic message logging performs significantly better than the pessimistic one. However, such logging schemes face the danger of the rollback propagation. It is possible that at the point in which a failure occurs some of the messages have not been actually written to the stable storage. In this sense optimistic model must employ more elaborate recover procedures in order to calculate the recovery line.

A necessary step, in order to perform the rollback correctly, is to track the process dependency during the failure free execution. The process dependency can be formulated in the following model [44]: Let $C_{i,x}$ be the x^{th} checkpoint of process P_i, $I_{i,x}$ denotes the checkpoint interval between checkpoints $C_{i,x-1}$ and $C_{i,x}$. Assuming process P_i at interval $I_{i,x}$ sends a message m to P_j, it piggybacks the interval value $I_{i,x}$ onto the message. When P_j receives the message during interval $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$. Unlike the message content, the dependency information is recorded onto the stable storage synchronously with the message passing. Thus, even if a volatile message log is lost due to a failure, the dependency information will still be available to the recovering process. As a result, the rollback in the optimistic message logging model is based on the observation that if there is an edge from $C_{i,x}$ to $C_{j,y}$ and a failure forces $I_{i,x}$ to be rolled back, $I_{j,y}$ must be rolled back as well.

In particular, when a failure occurs the recovering process starts the rollback by trying to collect the dependency information from all other surviving process. Using the global dependency information, the recovering process calculates a valid recovery line by using the Reach-ability Analysis [44]:

1. *include all local checkpoints and all processes's current states (both surviving and failed) as a set CKPT;*
2. *for any two elements in the set CKPT,* $I_{i,x}$ *and* $I_{j,y}$*, if a dependency exist from* $I_{i,x}$ *to* $I_{j,y}$*, draw an edge from* $I_{i,x}$ *to* $I_{j,y}$*;*
3. *Mark the failed process;*
4. *while (at least one member of CKPT is marked)*
   {
   mark all elements in CKPT that can be reached by at least one dependency edge;
   delete the marked elements;
   }
5. *The last unmarked elements from each process in the set CKPT forms the recovery line.*

Then the failed process broadcasts the rollback information to other surviving processes. Upon receiving a rollback request, the surviving process quits the current execution and rolls back to the indicated local checkpoint.

2.3.3 Causal Message Logging

The causal message logging approach tries to combine the merits of the pessimistic and optimistic models [45]. It avoids the blocking of synchronous message logging operation, meanwhile guarantees no cascaded rollback, which is limited to the latest checkpoints for all processes.

In particular, causal message logging guarantees no domino effect by ensuring that the determinant (message receiving) of every nondeterministic event (message sending) that causally precedes the state of a process is either stable or it is available locally to that process [74]. Considering the example given in Figure 5, suppose that

message m_6 is lost due to the failure of process P_1. However to P_1, m_5 is a nondeterministic event (sending message), the determinant of which, m_2 m_4 have been logged by P_0, P_2 (suppose using Sender-based Logging). Therefore P_0 and P_2 are able to guide the recovery of P_1 by replaying the message m_2, m_4.

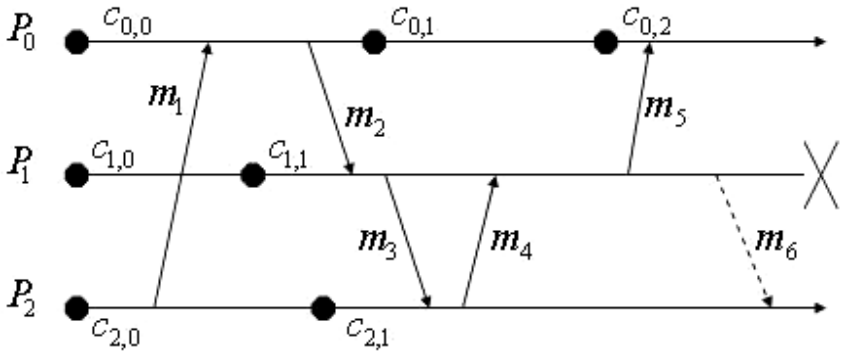

Figure 5. Causal Message Logging

Manetho [46] presents the first implementation of a causal message logging protocol. Each process maintains an *antecedence graph* which records the message passing events between processes. Upon sending a message, the sender process piggypacks its local antecedence graph on the message, which will be recorded at the receiver end.

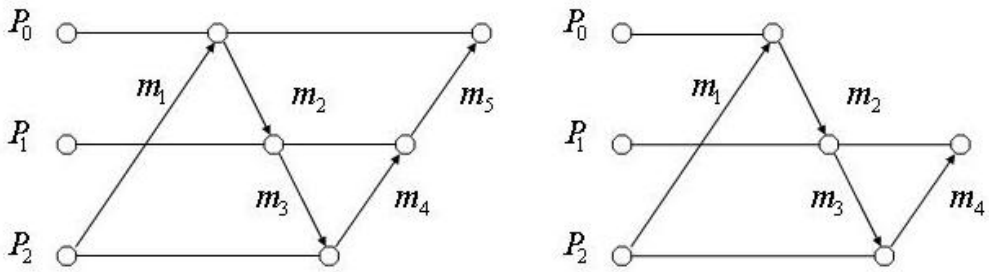

Figure 6. Antecedence Graph

Figure 6(a). shows the antecedence graph of process P_0 in Figure 5 after receving m_5. Figure 6(b) is the antecedence graph of process P_2 after sending message m_4. As the

figures show, the antecedence graph is propogated along with the message passing. At the time P_1 crashes, the antecedence graph provides P_0 with a complete history of the message passing events of the three processes, in which a node represents a message receiving event (except the first node of each process that represent the execution start), and the edges correspond to the message passing operation. Hence, the surviving process will know how to guide the failed process during recovery.

When a process sends a message to another one, it does not send the complete graph but an incremental piggybacking: all events preceding one initially created by the receiver need not to be sent back to it. Another algorithm has been proposed in [47] to reduce the amount of piggybacking on each message. It partially reorders events from a log inheritance relationship. Moreover it requires no additional piggybacking information. This allows having some information about the causality a receiver may already hold.

2.3.4 Summary of Uncoordinated Checkpoint

The uncoordinated checkpoint protocol was originally introduced at a time when the communication overhead far exceeded the overhead of accessing the stable storage. It was beneficial to try to retrench the network communication as much as possible. Moreover, at that time the memory available to run a program tended to be small. These tradeoffs naturally favoured the uncoordinated checkpoint schemes over the coordinated ones. However, current technology trends put these tradeoffs into a different perspective. With the significant increase of bandwidth in recent years, the overhead of coordination becomes negligible compared to the overhead of saving the program states [46-51]. Using techniques such as concurrent and incremental checkpoint, the overhead of either coordinated or uncoordinated checkpoint is essentially the same. Therefore, the uncoordinated checkpoint is not likely to be an attractive option in practice given the negligible performance gains. These gains do not justify (a) the complexities of finding a consistent recovery line after the failure, (b) the susceptibility to the Domino Effect, (c) the high storage overhead of saving multiple checkpoints for each process, and (d) the overhead of garbage collection. This leads to the conclusion that the coordinated checkpoint is superior to the uncoordinated scheme.

2.4 Communication-Induced Checkpoint

In a communication-induced checkpoint system [52], the creation of recovery line is uncoordinated. Processes are given the autonomy to choose when to save their execution states. In other words, they are independent to that of creating local checkpoints. However, to avoid the *Domino Effect* caused by uncoordinated checkpoints, this independence is in certain cases constrained by additional checkpoints that are forced in order to guarantee the eventual progress of the recovery line. The purpose of forced checkpoints is to break the communication and checkpoint patterns that may lead to invalid recovery line.

We observe two types of approaches to communication-induced checkpoint: ***Model-based Checkpoint*** and ***Header-based Checkpoint***. Model-based checkpointing protocols maintain checkpoint and communication structures that prevent useless recovery lines or achieve some even stronger properties [77]. Header-based protocols assign stamps to local and forced checkpoints such that checkpoints with the same stamp value at all processes form a valid recovery line.

2.4.1 Model-based Checkpoint

In the model-based checkpoint, the inter-process communication pattern is restricted so that the danger of in-transit and orphan messages is prevented from occurrence. In the simplest case, a process triggers a local checkpoint following every message passing operation to prevent the message becoming an in-transit or orphan one. In a more advanced case, the MRS model [53] limits the processes in a way, in which no message sending operation is allowed to be performed if there is any incoming message that has not been received. And additional checkpoints are forced between any consecutive sending and receiving operations, to ensure the validity of the recovery line. Obviously such communication pattern adds too many constraints in terms of programming and introduces siginificant overhead, what make it over-limited in the real world.

2.4.2 Header-based Checkpoint

The header-based checkpoint model piggybacks additional information onto the application messages to help the system identify the in-transit and orphan messages.

The simplest header would be one bit, which toggles between red (zero) and black (one) indicating the consecutive checkpoint intervals [37]. Upon detecting an incoming message with a header different from the local value, the receiving process will either trigger a new checkpoint or log the message.

In some cases no marker messages are needed, because the header information may fully replace the marker message for the process coordination. In these cases the coordination overhead is converted to the header overhead. However, in order for this to be achieved a process must be able to initiate local checkpoints on its own. In more common cases, coordination message are still necessary. In certain cases processes may not communicate with one another in previous checkpoint interval and therefore not be accessed by the new recovery line. As a consequence, most existing communication-induced checkpoint systems employ the header as well as the coordination message to form the global recovery line [33, 35].

2.4.3 Summary of Communication-induced Checkpoint

Theoretically, the communication-induced protocols are believed to have several advantages over the two other styles of checkpoint. For instance, it allows considerable process autonomy in deciding when to take checkpoints. Also they are believed to scale up well with a larger number of processes since they do not require the processes to participate in a global coordination. However, these advantages come at a price. First, the header information piggybacked on application messages occasionally induces processes to take forced checkpoints before they can process the messages, which may introduce unpredictable overheads. Second, processes have to pay the overhead of piggybacking information on top of application messages. Moreover, for each process several checkpoint files need to be maintained for the recovery procedure.

A study has shed some light on the behaviour of communication-induced checkpoint [54]. It presents an analysis of these protocols based on a prototype implementation and a validated simulation, showing that communication-induced checkpoint does not scale as well as expected. This is due to application messages whose occurrence within the execution of forced checkpoints makes it very difficult to predict the amount of stable storage that will be necessary for a particular program to run. Also,

this unpredictability affects the policies that govern where to force local checkpoints and makes communication-induced protocols cumbersome to use in practice. Furthermore, the study shows that the benefit of autonomy in allowing processes to take local checkpoints at their convenience does not seem to hold. In all experiments, a process takes at least twice as many forced checkpoints as local, autonomous ones.

2.5 Conclusion

This chapter examined the three main protocols for checkpoint/recovery of MPI programs: Coordinated, Uncoordinated and Communication-induced. Each of these protocols offer different advantages and tradeoffs with respect to the failure free execution, the number of processes needed to rollback, system complexity, algorithm scalability, the fault rate at which the algorithm remains valid, the latency of a recovery line commit, the storage overhead caused by checkpointing and rollback extent. A summary of the comparison between the three checkpoint protocols is presented in Table 2. Considering all factors the non-blocking coordinated checkpoint is evaluated to be the most practical approach for the MPI programs running in a heterogeneous network of computers.

	Coordinated		Uncoordinated			Communication-induced	
	Blocking	**Non-blocking**	**Pessimistic**	**Optimistic**	**Causal**	**Model-based**	**Header-based**
Failure-free Overhead	High	Very Low	Very high	Normal	Low	High	Unpredictable
Process Rollback	All	All	Failed	Failed	Failed	All	All
Scalability	Poor	Normal	Good	Good	Normal	Very Good	Poor
Algorithm Complexity	Very Simple	Simple	Simple	Complex	Very Complex	Simple	Simple
Fault Rate	Low	Medium	Very High	Low	High	High	Low
Recovery Line Commit	Very Low	Low	N/A	N/A	N/A	Very Low	High
Storage Overhead	Latest Checkpoint	Latest Checkpoint	Latest Checkpoint	All Checkpoints	Latest Checkpoint	Latest Checkpoint	Latest+Forced Checkpoints
Rollback Extent	Latest Checkpoint	Latest Checkpoint	Latest Checkpoint	Unpredictable	Latest Checkpoint	Latest Checkpoint	Latest/Forced Checkpoint

Table 2. Comparison of Different Checkpoint Protocols

Chapter 3

Event Logging: Application-level Coordinated Checkpoint for MPI

3.1 Introduction

The logging operation has been traditionally presented by message logging as an assistant mechanism to allow the use of uncoordinated checkpointing with no domino effect. However a system may also combine logging with coordinated checkpointing, yielding several benefits with respect to performance and simplicity [55]. These benefits include those of coordinated checkpointing —such as the simplicity of recovery and garbage collection, and those of message logging —such as fast output commit. Most prominently, this combination obviates the need for flushing the volatile message logs to stable storage in a sender-based logging implementation. Thus, there is no need for maintaining large logs on stable storage, resulting in lower performance overhead and simpler implementations. The combination of coordinated checkpointing and message logging has been shown to outperform one with uncoordinated checkpointing and message logging [55]. Therefore, the purpose of logging should no longer be to allow uncoordinated checkpointing. Rather, it should be the desire for the coordinated checkpoint to manage the in-transit and orphan messages.

This chapter presents Event Logging, an application-level coordinated checkpoint algorithm for MPI programs running in a heterogeneous network. The main contribution of the Event Logging algorithm is that this algorithm is applicable to various MPI implementations as well as different heterogeneous platforms.

In general, the Event Logging Algorithm asks both the sender and receiver processes to keep logs of the message envelopes, which are exchanged at the checkpoint stage to identify the in-transit message and orphan message envelopes. Once the process finds out these “trouble” messages, it saves the orphan message envelopes to avoid the inconsistency upon recovery. It also uses the in-transit message envelopes to log

the in-transit messages. Upon recovery, the process replays the logged in-transit messages and discards the repeated orphan messages.

In this chapter we concentrate on describing the algorithm. The implementation details are left to the next chapter.

3.2 Background

3.2.1 Problem Space

Mentioned in Section 2.2.2, the Chandy-Lamport algorithm requires the communication to operate in the FIFO manner. Simply put, FIFO is a property that asks the communication channel to behave like a tunnel, where the out-of-sequence message sending/receiving is forbidden. In other words, once a message is sent, it enters a tunnel in which the message must stay at its position. Despite the possibility of different routing paths over the network, the messages leave the queue always in the same order as they enter it.

The reason why FIFO is mandatory for the Chandy-Lamport algorithm is that the marker used in the algorithm acts as a fence to separate the message passing around the local checkpoint. Considering the following scenario in Figure 7: P_0 sends three messages in the order: $\{m_1, marker, m_2\}$ to P_1. With the FIFO manner, the messages reach P_1 in the same order as they are sent. Also, at the time m_1 reaches P_1, P_1 has already finished its local checkpoint. So that m_1 is logged as an in-transit message and m_2 is an intra message according to the Chandy-Lamport Algorithm. However, if the underlying network does not behave in the FIFO manner, the arriving order might be different from the sending order. If the arrival is $\{marker, m_1, m_2\}$, m_1 will be omitted from logging and make the recovery line unrecoverable. Furthuremore, if the arrival is $\{m_1, m_2, marker\}$, the logging of m_2 leads the recovery line inconsistent.

Being the cornerstone of the Chandy-Lamport Algorithm, the FIFO manner is a true statement of behaviour of message passing when looking deeply into the details of MPI implementations: most MPI implementations define a low level channel, which sits on top of the underlying network. On a low performance network such as

Ethernet, TCP is used, provides the FIFO property; while most high performance NICs (Network Interface Card) provide the FIFO reliable communications. As a consequence, there exist many checkpoint/recovery systems for MPI programs built on the pure Chandy-Lamport algorithm [15, 19, 20, 56]. However, this statement is false when looking at MPI from the top. From the point of view of programming it is valid to receive messages even in the reverse order of the sending. While this situation may be rare, a much more common situation is that messages are sent and received out of sequence. Although the MPI standard defines a priority rule to regulate the message sending/receiving at the application-level, called Non-overtaking property [6], it still cannot meet the requirement of FIFO communication.

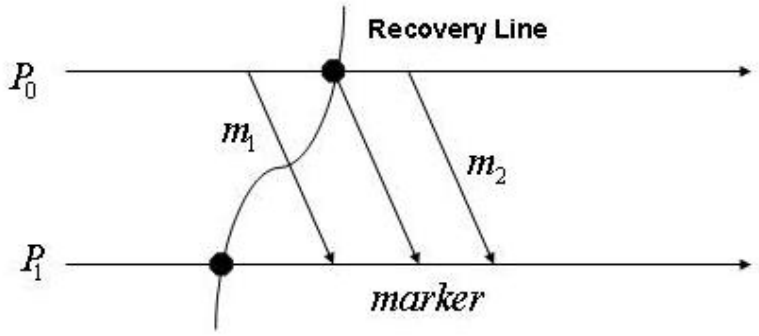

Figure 7. FIFO Message Passing

"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" [57]. A message in MPI is labelled by the envelope <*rank, tag, communicator*> (where *rank* is replaced by *target* on sending or *source* on receiving). If there are several messages with the same envelope in the receiving buffer, some rules must be used to solve the problem of determining which one should be picked up by a matching receiving post. With the non-overtaking property, MPI guarantees the message sequence and correct communication semantic.

For example, if one process sends two successive messages $\{m_1, m_2\}$ that have the same envelope to another process. The two messages, on the receiver side, will be picked up in the same sending order $\{m_1, m_2\}$, not $\{m_2, m_1\}$. On the other hand, if one process posts two receiving calls with the same envelope and there is only one matching message in the buffer, it is always the first receiving post that gets satisfied

even if it is a non-blocking operation. In this scenario where two processes both send a message with the same <*tag, comm*> to the same destination, these two messages are only different at the *source* rank. The destination process posts a matching receiving with the wildcard *MPI_ANY_SOURCE*, making it match both messages.
The non-overtaking property does not apply to such case, since there is no sending order between these two messages. Therefore, the receiving post may pick either of them, depending on which one arrives first. A note is that this only applies to the single-threaded environment. As MPI sets no rules of multithread, different MPI implementations may have different explanations of the non-overtaking message passing property in their own multithreaded features.

We argue that FIFO is different from the non-overtaking property. Principally, non-overtaking applies only to the messages with identical envelopes. To the messages with different envelopes, the program is free to receive them in any order. However FIFO is more restrictive since it requires all messages to be received in their sending order.

As the Chandy-Lamport algorithm works under the assumption of FIFO, there are problems when bringing the algorithm to the checkpoint of MPI because the FIFO assumption is not strictly required by the MPI standard at the application level. This one reason why most existing checkpoint/rollback-recovery systems built on Chandy-Lamport algorithm, implement the algorithm in a non-portable way. These systems have to resort to the help of a low-level layer, which provides the FIFO guarantee, to implement the Chandy-Lamport algorithm. It is observed that either these systems are built on their own MPI implementations or use some special network protocol. However, given the portability concern, such platform-dependent systems would not be popular in a heterogeneous network. In this sense, the proposed solution must be able to release the application-level non-FIFO constraint, so as to cope with different MPI implementations.

3.2.2 Existing Approaches

Unfortunately, although many variants that improve on the Chandy-Lamport algorithm have been developed in recent years [26, 27, 32-37-58, 73], only a few had provided supports to the application-level non-FIFO problems.

A straightforward approach is to coordinated processes used loosely synchronized clocks [26-31]. However, as discussed in Section. 2.2.1, time-based coordination suffers from its scalability, which is one of the main issues in a network of computers.

Message Tagging is another approach to address the application-level FIFO problem [37, 51, 59]. The idea behind Message Tagging is that the system wraps some additional information (header) onto the outgoing messages, which specifies the sending order. On the receiver side, the process receives the incoming messages, and unwraps the header information, so that system can obtain the message sending order so as to identify the in-transit and orphan messages.

The implementation for Message Tagging can be classified into two categories: System-level and Application-level. With System-level Message Tagging, the wrapping is done at the network layer, which obviously does not fit our goals. As to application-level Message Tagging, the header information is piggybacked directly on the MPI messages.

In general, we observe three approaches to implement application-level message tagging:

1. **Header Messag**e: In this approach, the header is not actually included in the original message. Instead, the system sends another extra message following the outgoing message to pass the header information. We call the original message *Host Message* and the assistant message *Header Message*. Sample code looks like:

```
MT_MPI_Send(buffer,…)
{
        MPI_Send(buffer,…);          // send the host message
```

```
        MPI_Send(header,…);          // send the header message
}
```

Note, the header message must be sent/received using the same envelope of its host message, to guarantee that a header message is always received immediately after its host. Since the system needs to perform an extra communication operation with every message, this approach doubles the program's communication volume. Moreover, because the header message uses the same envelope as the host message, all header messages are passed via user-specified communicators, not a global, independent communicator dedicated for the coordination. So, checkpoints in this pattern may be significantly delayed, when the target process postpones receiving messages.

2. **Buffered Mode**: The second option is to use buffered mode communication in MPI. Since it is impossible to expand the application buffer, the checkpoint system needs to copy both the message and the header into an external buffer, and send the buffer instead of the original message. A sample is:

```
MT_MPI_Send(msg,……)
{
    // allocate a temporary buffer
    buf=malloc(header_size+msg_size);
    // copy the header into the buffer
    MPI_Pack(header,header_size…,buf,…);
    // copy the message into the buffer
    MPI_Pack(msg,msg_size…,buf+msg_size…);
    // send the temporary buffer
    MPI_Send(buf,header_size+msg_size,MPI_PACKED,….);
    // release the buffer
    free(buf);
}
```

It is noted that all messages use the same datatype MPI_PACKED for communication.

3. **Derived Datatype**: MPI allows users to construct composite data structures from the simple types. Facilitated by this function, the sender and receiver processes agree to build up a temporary datatype upon communication, which is comprised of both the original message and the header, and use the temporary datatype to pass the message. The wrapper function looks like:

```
MT_MPI_Send(msg,….)
{
        buf=malloc(header_size+msg_size);
        // copy the header into the buffer
        memcpy(buf,header,header_size);
        // copy the message into the buffer
        memcpy(buf+header_size,msg,msg_size);
        // build a temporary datatype;
        MPI_Type_struct(….,temp_type);
        MPI_Type_commit(temp_type);
        // send the buffer as temp_type datatype
        MPI_Send(buf,1,temp_type,…);
        // free the temporary type
        MPI_Typ_free(temp_type);
        // de-allocate the buffer
        free(buf);
}
```

Although the buffered mode and derived datatype options are differerent from the point of view of programming, we argue that in effect, they are similar. The derived datatype has the advantage on performance. However, the common downside of the two solutions is that they are not completely safe. For example, MPI_Status is a commonly used structure in MPI, which lets users access the information about the incoming message. But with message tagging, the information is changed. For example the length property includes not only the size of the original message, but also the header as well. Consider the code executed by the receiver below:

```
MPI_Probe(source, tag, comm., &status);
```

MPI_Get_count(&status, MPI_INT, &count);
MTC_MPI_Recv(buf, count, MPI_INT, source, tag, comm.,&status)

Suppose the incoming message contains 4 integers and the header is one additional integer, then the total size is 5 integers, in which the variable count equals to 5. However, if the target process wants to receive the message correctly, the value of count should be 4 when passing it to the call of *MTC_MPI_Recv()* function. Otherwise, the header is received as part of the message. In the worse case, if the message and the header use different datatypes, the call of *MPI_Get_count()*simply returns *MPI_UNDEFINED* as the parameter *count*. One may argue that the solution is to exclude the header size. However, we note the difficulty to do so at the application level, due to the highly implementation-dependent definition of *MPI_Status*.

Also, as the experiment results in Section 5.7 shows, the performance of the derived datatype approach fluctuates with the message size. This is due to the system needing to manage extra buffer space for tagging the header information. With the increase of the message size, the memory operation costs get more and more significant.

Moreover, message tagging may be very slow to output a recovery line. This is because that the header is bound with the message and the system is able to intercept an in-transit message only when the process tries to receive this message. It is possible that, as long as the process does not post the receiving call, the system cannot log the in-transit messages. Hence the recovery line would never be completed.

Finally, we notice that the message tagging approach still needs coordination message to work properly. The coordination message used in message tagging is to tell processes the number of the in-transit message it needs to log. Otherwise, a process would have no idea whether there is any in-transit message left.

For all these reasons, message tagging is not as attractive as it looks. An appropriate solution should have better performance, and fast recovery line commit. And the most important, it must be totally compliant with the MPI standard.

3.3 Event Logging

In the following paragraphs, we present *Event Logging* for application-level process coordination. First, it is necessary to differentiate Event Logging from Message Logging [38]. In particular, *Event Logging* is a variant of the *Chandy-Lamport* algorithm that coordinates distributed processes to form recovery lines. Unlike message logging, it records only the message envelopes, without the message content, reducing much of the memory overhead introduced by message logging [60].

Also, it is noted that Event Logging is different from the event logger [19], which is widely used in message logging systems for tracking the process causal dependency. Although the function of both Event Logging and the event logger is to record the message passing events, the fundamental difference is that the former is a high level algorithm used for process coordination, while the latter is a low level module that is built into the message logging system.

In *Event Logging*, every process keeps a log for the sending and the receiving events it performs. When a new checkpoint occurs, send logs are exchanged between sender and receiver. When a process gets another's send logs, it pairs it up with the local receive logs to match the message envelopes. As the message envelope is logged at the same time as 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 finds out which category the message is in.

3.3.1 Definitions and Assumption

A process's execution is divided into a sequence of intervals separated by checkpoints. A checkpoint interval starts with any instruction following a local checkpoint, ending upon the completion of the next local checkpoint. A checkpoint interval includes all statements between two successive local checkpoints. Each checkpoint interval is assigned a unique sequence number, which is equal to the number of local checkpoints that have been completed by the process. Since our protocol is based on coordinated checkpoint, the local checkpoints that compose a recovery line have the same sequence number value.

For any message, the message is sent in a checkpoint interval of the source, and received in another one of the target, regardless of whether these two intervals belong to the same recovery line. We mark the sending event as $S_{p,i}(m)$, which meaning the message *m* is sent in the process *p*'s i^{th} checkpoint interval ($S_{p,i}$ is the general form for any message); and the receiving event as $R_{q,j}(m)$, means the message is received in *q*'s j^{th} checkpoint interval (simplified as $R_{q,j}$). Also, a send log is the collection of the outgoing message envelopes, noted by $SEND_{p,i}$ (the send log of process *p*'s i^{th} checkpoint interval). Similarly, a receive log is $RECV_{q,j}$. Suppose that *F* is the set of the valid message envelopes, which consist of *n* elements: $F = \{f_1, f_2, ..., f_n\}$. The send and receive logs are defined as $SEND_{p,i} = \{x^* \mid x \in F\}$ and $RECV_{q,j} = \{x^* \mid x \in F\}$. Also, $SEND_{p,i}^{\ x}$ ($RECV_{q,j}^{\ x}$) denotes the x^{th} message envelope in a send (receive) log.

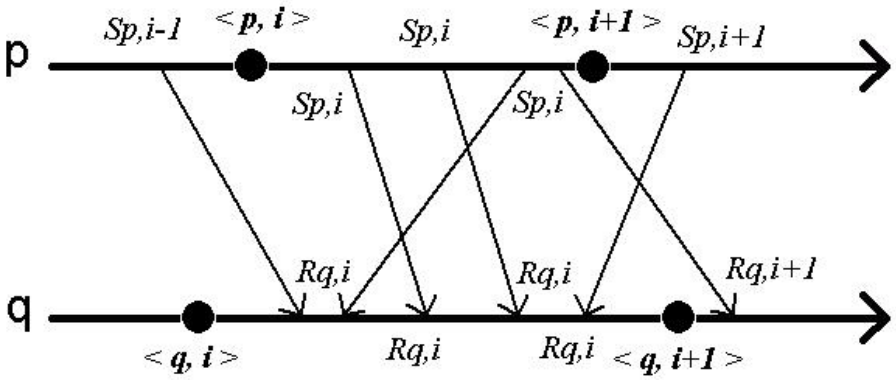

Figure 8. *p*'s send log and *q*'s receive log,

<*p,i*> denotes *p*'s ith checkpoint

Given the existence of orphan, intra and in-transit messages (See Figure 8), we use $SEND_{p,i}\{R_{q,0}, R_{q,1}, ..., R_{q,j}, ..., R_{q,x}\}(x \geq j \geq 0)$ to denote the general relation between the sending and receiving events. As $R_{q,j}$ denotes a receiving event happens on process *q* during the j^{th} checkpoint intervals, the expression means that an outgoing message can be received anytime during the program life. Also the corresponding receive log of *q*'s j^{th} interval is denoted

by $RECV_{q,j}\{S_{p,0}, S_{p,1}, ..., S_{p,i}, ..., S_{p,x}\}(x \geq i \geq 0)$. Obviously there exist some matching relations between these two logs:

$$SEND_{p,i} \cap RECV_{q,i} = SEND_{p,i}\{R_{q,i}\} = RECV_{q,i}\{S_{p,i}\}; \quad // \textit{intra messages}$$

$$SEND_{p,i} \cap RECV_{q,i+x} = SEND_{p,i}\{R_{q,i+x}\} = RECV_{q,i+x}\{S_{p,i}\}; \quad // \textit{in-transit messages}$$

$$SEND_{p,i} \cap RECV_{q,i-x} = SEND_{p,i}\{R_{q,i-x}\} = RECV_{q,i-x}\{S_{p,i}\}; \quad // \textit{orphan messages}$$

Note, a send log that has no matching receive log is written as $SEND_{p,i}\{-\}$, and an unmatched receive log is $RECV_{q,j}\{-\}$. However, since the message passing is reliable, these are not the final states. The counterparts must be found somewhere in the following checkpoint interval logs.

In order to simplify the depiction of our model, we make the following assumption. Later, we will show how to remove such a limitation for all kinds of communication environment.

- **Assumption**: *The following discussion assumes any message passing finishes in no more than two successive checkpoint intervals.*

Then, the send and receive logs are of the form like: $SEND_{p,i}\{R_{q,i-1}, R_{q,i}, R_{q,i+1}\}$ and $RECV_{q,i}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$.

Note, the format given above of send event log $SEND_{p,i}\{R_{q,i-1}, R_{q,i}, R_{q,i+1}\}$ doesn't mean that the messages are sent in the receiving order. However, with the Non-overtaking manner, the receiving order of the messages with the same envelope is the same as the sending order.

3.3.2 Algorithm

The aim of Event Logging algorithm is to coordinate distributed processes by identifying the evnelopes of in-transit and orphan messages. A general idea to identify such messages in Event Logging is to ask every process to log all envelopes it has

sent and received. Upon checkpointing (using the Chandy-Lamport algorithm), the send logs are bounded with the marker messages to be exchanged between processes. When detecting a marker message, the process unpacks the incoming seng logs, and compares them with the local receive logs to identify the in-transit and orphan messages.

To identify the in-transit and orphan messages, we start from the first checkpoint interval $\{C_{p,0}, C_{q,0}\}$. For the first interval, a trick is that there are no orphan message envelopes in the send log of process *p*: $SEND_{p,0}\{R_{q,0}, R_{q,1}\}$ and no in-transit message envelopes in the receive log of process *q*: $RECV_{q,0}\{S_{p,0}, S_{p,1}\}$. So it is fairly easy to identify the in-transit and orphan message envelopes by removing the intra message logs:

$$
\begin{aligned}
& SEND_{p,0} - SEND_{p,0} \cap RECV_{q,0} \\
&= SEND_{p,0}\{R_{q,0}, R_{q,1}\} - SEND_{p,0}\{R_{q,0}, R_{q,1}\} \cap RECV_{q,0}\{S_{p,0}, S_{p,1}\} \\
&= SEND_{p,0}\{R_{q,0}, R_{q,1}\} - SEND_{p,0}\{R_{q,0}\} \\
&= SEND_{p,0}\{R_{q,1}\} \\
&= SEND'_{p,0}
\end{aligned}
$$

And

$$
\begin{aligned}
& RECV_{q,0} - SEND_{p,0} \cap RECV_{q,0} \\
&= RECV_{q,0}\{S_{p,0}, S_{p,1}\} - SEND_{p,0}\{R_{q,0}, R_{q,1}\} \cap RECV_{q,0}\{S_{p,0}, S_{p,1}\} \\
&= RECV_{q,0}\{S_{p,0}, S_{p,1}\} - RECV_{q,0}\{S_{p,0}\} \\
&= RECV_{q,0}\{S_{p,1}\} \\
&= RECV'_{q,0}
\end{aligned}
$$

The in-transit $SEND'_{p,o}$ and orphan $RECV'_{q,0}$ message envelopes are kept by process *q* until the next checkpoint $\{C_{p,1}, C_{q,1}\}$. For the second checkpoint interval, however, the send log of *p* looks like $SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}\}$ and the receive log of *q* is $RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}\}$. Then we can clear these logs by:

$$
\begin{aligned}
& SEND_{p,1} - SEND_{p,1} \cap RECV'_{q,0} \\
&= SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}\} - SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}\} \cap RECV_{q,0}\{S_{p,1}\} \\
&= SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}\} - SEND_{p,1}\{R_{q,0}\} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}\} \\
&= SEND'_{p,1}
\end{aligned}
$$

And

$$
\begin{aligned}
& RECV_{q,1} - SEND'_{p,0} \cap RECV_{q,1} \\
&= RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}\} - SEND_{p,0}\{R_{q,1}\} \cap RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}\} \\
&= RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}\} - RECV_{q,1}\{S_{p,0}\} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}\} \\
&= RECV'_{q,1}
\end{aligned}
$$

As it shows, after the clearance, the event logs of the second checkpoint interval look similar to the initial interval. So, the same algorithm can be applied to identify the in-transit and orphan message envelopes for this recovery line:

$$
\begin{aligned}
& SEND_{p,1} - SEND'_{p,1} \cap RECV'_{q,1} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}\} - SEND_{p,1}\{R_{q,1}, R_{q,2}\} \cap RECV_{q,1}\{S_{p,1}, S_{p,2}\} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}\} - SEND_{p,0}\{R_{q,1}\} \\
&= SEND_{p,1}\{R_{q,2}\} \\
&= SEND''_{p,1}
\end{aligned}
$$

And

$$
\begin{aligned}
& RECV_{q,1} - SEND'_{p,1} \cap RECV'_{q,1} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}\} - SEND_{p,1}\{R_{q,1}, R_{q,2}\} \cap RECV_{q,1}\{S_{p,1}, S_{p,2}\} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}\} - RECV_{q,1}\{S_{p,1}\} \\
&= RECV_{q,1}\{S_{p,2}\} \\
&= RECV''_{q,1}
\end{aligned}
$$

Simple as the above algorithm looks, it demonstrates that as long as no failure occurs, it is feasible to identify the in-transit and orphan message envelopes for any checkpoint interval. When a fault occurs, the recovery automatically guarantees the event log of the first post-recovery checkpoint interval cannot contain in-transit and orphan message envelopes of the previous interval, $SEND_{p,i}\{R_{q,i}, R_{q,i+1}\}$ and $RECV_{q,i}\{S_{p,i}, S_{p,i+1}\}$. Then after recovery, the same protocol can be applied to start a

new event logging cycle. Hence, valid recovery lines can always be created throughout the life cycle of the MPI program.

3.3.3 Formal Analysis

In order to prove the correctness of the Event Logging algorithm, we need to prove two cases. The first case is that there are no messages with the same envelope in the log. In other words, each message is labelled uniquely. The second case is that there are some messages in the log with the same envelope.

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

- **Case 1: no messages with the same envelope**

First, since the messages' envelopes logged are unique, there must be one and only one matching send/receive pair for any event. In other word, there are two possible results of the matching: the counterpart of a send (receive) event can be found in the current or previous receive (send) log. The matching process fails if its counterpart cannot be found. Recalling the in-transit, orphan and intra messages defined by Lamport's Happen Before Relation, the Event Logging algorithm is just trying to identify messages by judging the message's Happen Before Relation.

Suppose the current checkpoint interval of process p is $C_{p,i}$ and process q's is $C_{q,j}$, according to the Happen Before Relation, a message m, which p sends to q in the current checkpoint interval can be is identified as an intra-message if and only if:

$$C_{p,i-1} \rightarrow S_{p,i}(m) \rightarrow C_{p,i} \ And \ C_{q,j-1} \rightarrow R_{q,j}(m) \rightarrow C_{q,j};$$

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: $SEND_{p,i}\{R_{q,j}\} = RECV_{q,j}\{S_{p,i}\}$.

The message, M, can be considered to be an in-transit message if and only if:

$$C_{p,i-1} \rightarrow S_{p,i}(m) \rightarrow C_{p,i} \ And \ C_{q,j} \rightarrow R_{q,j+1}(m);$$

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 the message, m, can be considered an orphan message if and only if

$$C_{p,i} \to S_{p,i+1}(m) \ And\, C_{q,j-1} \to R_{q,j}(m) \to C_{q,j};$$

This means that a receive log finds its matching send log in the source's previous checkpoint interval log: $SEND_{p,i+1}\{R_{q,j}\} = RECV_{q,j}\{S_{p,i+1}\}$.

Although it is impossible to check the future checkpoint interval logs, the trick is that the first checkpoint interval logs contain no in-transit and orphan messages of any previous checkpoint interval: $SEND_{p,0}\{R_{q,0}, R_{q,1}\}$ and $RECV_{q,0}\{S_{p,0}, S_{p,1}\}$. So we can easily clear the intra messages p sends q ($SEND_{p,0}\{R_{q,0}\} = RECV_{q,0}\{S_{p,0}\}$). for the unmatched send logs of p ($SEND_{p,0}\{-\}$) and the unmatched receive logs of q ($RECV_{q,0}\{-\}$), because any event log must have a counterpart somewhere and the message passing must be completed in the next interval, we conclude the final version of these unmatched event logs are $SEND_{p,0}\{R_{q,1}\}$, $RECV_{q,0}\{S_{p,1}\}$. We say the in-transit and orphan messages have been successfully identified. This is the cornerstone of the following *Proof by Induction*.

Then, assuming at the checkpoint intervals $C_{p,i}, C_{q,j}$, we have identified in-transit $SEND_{p,i-1}\{R_{q,j}\}$ and orphan $RECV_{q,j-1}\{S_{p,i}\}$ messages of $C_{p,i-1}$, $C_{q,j-1}$, the event logs of $C_{p,i}$, $C_{q,j}$ look like $SEND_{p,i}\{R_{q,j-1}, R_{q,j}, R_{q,j+1}\}$ and $RECV_{q,j}\{S_{p,i-1}, S_{p,i}, S_{p,i+1}\}$. The algorithm can remove the counterparts of them $RECV_{q,j}\{S_{p,i-1}\}$ and $SEND_{p,i}\{R_{q,j-1}\}$ from the event logs of $C_{p,i}, C_{q,j}$. After the removal, the event logs of $C_{p,i}$, $C_{q,j}$ will look like $SEND_{p,i}\{R_{q,j}, R_{q,j+1}\}$ and $RECV_{q,j}\{S_{p,i}, S_{p,i+1}\}$, containing no logs related with $C_{p,i-1}, C_{q,j-1}$. So, the same identifying method of the first checkpoint interval can be applied. Then we get the in-transit and orphan messages of $C_{p,i}, C_{q,j}$: $SEND_{p,i}\{-\}$ and $RECV_{q,j}\{-\}$ after removing the intra messages $SEND_{p,i}\{R_{q,j}\} = RECV_{q,j}\{S_{p,i}\}$. As for the $C_{p,0}, C_{q,0}$, it is concluded that the final version of $SEND_{p,i}\{-\}$ and $RECV_{q,j}\{-\}$

must be $SEND_{p,i}\{R_{q,j+1}\}$ and $RECV_{q,j}\{S_{p,i+1}\}$. Therefore, we conclude for any checkpoint interval $C_{p,i-1}$, $C_{q,j-1}$ $(i \geq 0, j \geq 0)$, all in-transit, intra and orphan messages can be identified.

Therefore, **Case 1: no messages in the events log have the same envelope** is proved.

- **Case 2: some messages in the log with the same envelope**

Based on what we have learned in Case.1, now we prove a more complex case, for which there are some messages in the event log that have the same envelopes. Recalling the event log definition $SEND/RECV = \{x^* \mid x \in F\}$ and $F = \{f_1, f_2, ..., f_n\}$, we mark the x^{th} message of the envelope *f* with f^x, and *x* messages of the envelope *f* with *f*x*. For example, if process *p* sends *n* messages of the same envelope *f*, in the i^{th} checkpoint interval, the send log will be $SEND_{p,i} = \{f^1, f^2, ..., f^n\}$. Also according to the Non-overtaking message passing property, the message passing of the same envelope obeys the following precedence rule: If $x<y$ then $S_p\{f^x\} \rightarrow S_p\{f^y\}$ and $R_q\{f^x\} \rightarrow R_q\{f^y\}$. These kinds of message logs must keep the same order. For the messages with different envelopes, we argue that the logging is independent. In other words, this kind of message can be logged in any order. We call this the *Logging Independency Principle* (LIP).

With LIP, the event log can be converted into the forms: $SEND_{p,i} = \{f_1^*, f_2^*, ..., f_n^*\}$ and $RECV_{q,j} = \{f_1^*, f_2^*, ..., f_n^*\}$ (called *LIP Transformation*). Considering the example, where $F = \{f_1, f_2, f_3\}$ and process *p* sends the messages with the envelopes in the following order: f_1, f_1, f_3, f_2, f_3, the corresponding send log is $SEND_{p,i} = \{f_1, f_1, f_3, f_2, f_3\} = \{f_1^1, f^2{}_1, f_2^1, f_3^1, f_3^2\} = \{f_1 * 2, f_2 * 1, f_3 * 2\}$. The LIP transformation would not lead to a wrong identification result because it is the adjustment of the log order in an interval. A log would not be placed into another interval after it. As the message is classified by the Lamport's Happen Before Relation, a log's place adjustment keeps the original relation. So the identification of messages of different envelopes is safe with LIP Transformation.

As we have proved that Event Logging is capable of identifying messages with different envelopes, we simplify the problem by considering the case that all messages are delivered using the same envelope: $SEND / RECV = \{f^*\}$. Thus, due to the Non-overtaking property, we can get the following rule: If $x<y$ then $S_p\{f^x\} \rightarrow S_p\{f^y\}$ and $R_q\{f^x\} \rightarrow R_q\{f^y\}$, we argue that the message identification precedence of $SEND_{p,i} = \{f^1, f^2, ..., f^n\}$ does not intersect. In other words, for all messages with the same envelope f, the orphan message sending must happen before the intra message sending, which precedes the in-transit message sending. This argument is true because if there is an intra message f^x sent before some orphan message f^y, there exists the following relations: $C_{p,i} \rightarrow S_{p,i}\{f^x\} \rightarrow S_{p,i}\{f^y\}$ and $R_{q,j-1}\{f^y\} \rightarrow C_{q,j} \rightarrow R_{q,j}\{f^x\}$. However, this conflicts with the Non-overtaking property. Also, if an in-transit message f^x precedes some intra message f^y, we get: $S_{p,i}\{f^x\} \rightarrow S_{p,i}\{f^y\} \rightarrow C_{p,i}$ and $R_{q,j}\{f^y\} \rightarrow C_{q,j} \rightarrow R_{q,j+1}\{f^x\}$. This is also impossible. To prove this case, we apply the same technique used in Case 1:

To the first recovery line $C_{p,0}$, $C_{q,0}$ the event logs look like $SEND_{p,0}\{R_{q,0}, R_{q,1}\} = \{f*n\}$, $RECV_{q,0}\{S_{p,0}, S_{p,1}\} = \{f*m\}$. Since the intra message passing stays before any others, the *Max(n,m)* logs must be the intra message logs. If $n>m$, the remaining *(n-m)* logs in $SEND_{p,0}$ are in-transit message logs. Otherwise, the remaining *(m-n)* logs in $RECV_{q,0}$ are orphan message logs.

Then, suppose that the event logs of $C_{p,i}$, $C_{q,j}$ are $SEND_{p,i} = \{f*n\}$ and $RECV_{q,j} = \{f*m\}$ and we have identified the event logs of $C_{p,i-1}$, $C_{q,j-1}$: $SEND_{p,i-1}\{R_{q,j}\} = \{f^x\}$ and $RECV_{q,j-1}\{S_{p,i}\} = \{f^y\}$. The identification of $C_{p,i}$, $C_{q,j}$ goes as follow:

1) *Remove the first y logs from* $SEND_{p,i}$, *and the first x logs from* $RECV_{q,j}$. *Then they look as* $SEND_{p,i}\{R_{q,j}, R_{q,j+1}\} = \{f^{n-y}\}$ *and* $RECV_{q,j}\{S_{p,i}, S_{p,i+1}\} = \{f^{m-x}\}$.

2) *Remove |(n-y)- (m-x)| logs from both* $SEND_{p,i}$ and $RECV_{q,j}$.

3) *If (n-y)>(m-x), the remaining (n-y)-(m-x) logs,* $SEND_{p,i}\{-\} = \{f^{(n-y)-(m-x)}\}$ *are in-transit message logs. Otherwise, the (m-x)-(n-y) logs* $RECV_{q,j}\{-\} = \{f^{(m-x)-(n-y)}\}$ *are orphan messages logs.*

Then we conclude that Event Logging is capable of the identification of the Case 2.

3.3.4 Removal of the 2-interval restriction

In our algorithm, we suppose all message passing will finish in two successive intervals at most. However, in fact, it is possible that a message passing encompasses a larger latency, even though rarely. Therefore it is necessary to remove such a limitation.

To illustrate the reason, let's repeat the description of our algorithm. However, a change is that, without the 2-interval restriction, the event logs of process *p* and *q*'s first checkpoint interval will be: $SEND_{p,0}\{R_{q,0}, R_{q,1}, ...\}$ and $RECV_{q,0}\{S_{p,0}, S_{p,1}, ...\}$. Then the algorithm is revised:

$$
\begin{aligned}
&SEND_{p,0} - SEND_{p,0} \cap RECV_{q,0} \\
&= SEND_{p,0}\{R_{q,0}, R_{q,1}, ...\} - SEND_{p,0}\{R_{q,0}, R_{q,1}, ...\} \cap RECV_{q,0}\{S_{p,0}, S_{p,1}, ...\} \\
&= SEND_{p,0}\{R_{q,0}, R_{q,1}, ...\} - SEND_{p,0}\{R_{q,0}\} \\
&= SEND_{p,0}\{R_{q,1}, ...\} \\
&= SEND'_{p,0}
\end{aligned}
$$

And

$$
\begin{aligned}
&RECV_{q,0} - SEND_{p,0} \cap RECV_{q,0} \\
&= RECV_{q,0}\{S_{p,0}, S_{p,1}, ...\} - SEND_{p,0}\{R_{q,0}, R_{q,1}, ...\} \cap RECV_{q,0}\{S_{p,0}, S_{p,1}, ...\} \\
&= RECV_{q,0}\{S_{p,0}, S_{p,1}, ...\} - RECV_{q,0}\{S_{p,0}\} \\
&= RECV_{q,0}\{S_{p,1}, ...\} \\
&= RECV'_{q,0}
\end{aligned}
$$

The in-transit $SEND'_{p,o}$ and orphan $RECV'_{q,0}$ message envelopes are kept by process *q* until the next checkpoint $\{C_{p,1}, C_{q,1}\}$. For the second checkpoint interval, however,

the send log of p looks like $SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}, ...\}$ and the receive log of q is $RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}, ...\}$. Then we can clear these logs by:

$$
\begin{aligned}
& SEND_{p,1} - SEND_{p,1} \cap RECV'_{q,0} \\
&= SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}, ...\} - SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}, ...\} \cap RECV_{q,0}\{S_{p,1}, ...\} \\
&= SEND_{p,1}\{R_{q,0}, R_{q,1}, R_{q,2}, ...\} - SEND_{p,1}\{R_{q,0}\} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}, ...\} \\
&= SEND'_{p,1}
\end{aligned}
$$

And

$$
\begin{aligned}
& RECV_{q,1} - SEND'_{p,0} \cap RECV_{q,1} \\
&= RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}, ...\} - SEND_{p,0}\{R_{q,1}\} \cap RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}, ...\} \\
&= RECV_{q,1}\{S_{p,0}, S_{p,1}, S_{p,2}, ...\} - RECV_{q,1}\{S_{p,0}\} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}, ...\} \\
&= RECV'_{q,1}
\end{aligned}
$$

As it shows, after the clearance, the event logs of the second checkpoint interval look similar to the initial interval. So, the same algorithm can be applied to identify the in-transit and orphan message envelopes for this recovery line:

$$
\begin{aligned}
& SEND_{p,1} - SEND'_{p,1} \cap RECV'_{q,1} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}\} - SEND_{p,1}\{R_{q,1}, R_{q,2}\} \cap RECV_{q,1}\{S_{p,1}, S_{p,2}\} \\
&= SEND_{p,1}\{R_{q,1}, R_{q,2}\} - SEND_{p,0}\{R_{q,1}\} \\
&= SEND_{p,1}\{R_{q,2}\} \\
&= SEND''_{p,1}
\end{aligned}
$$

And

$$
\begin{aligned}
& RECV_{q,1} - SEND'_{p,1} \cap RECV'_{q,1} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}, ...\} - SEND_{p,1}\{R_{q,1}, R_{q,2}, ...\} \cap RECV_{q,1}\{S_{p,1}, S_{p,2}, ...\} \\
&= RECV_{q,1}\{S_{p,1}, S_{p,2}, ...\} - RECV_{q,1}\{S_{p,1}, ...\} \\
&= RECV_{q,1}\{S_{p,2}, ...\} \\
&= RECV''_{q,1}
\end{aligned}
$$

As it demonstrates, the algorithm can still identify the envelopes of the in-transit and orphan messages. The only difference is that for the i^{th} checkpoint interval, the process needs to keep a log of the in-transit and orphan message envelopes that happened in the previous (*i-1*) intervals (if such exist). The clearance of the i^{th} interval's logs uses all (*i-1*) intervals'.

Note that for the in-transit message the process must be able to get these messages upon recovery, no matter how early the sending takes place. In other words, to recover from the i^{th} checkpoint interval, there is no difference between in-transit messages sent in the $(i-1)^{th}$ interval to these sent in $(i-x)^{th}$ interval. We maintain a log for all the unreceived in-transit messages and append the new in-transit message envelopes onto it. All sending event entries are kept until the message is picked up by the target (then the entry will be removed from the log). In this way, our algorithm relies on no specific premise and can adapt to any communication demand.

Also, the Event Logging algorithm is safe though the fairness property is not guaranteed by MPI. Simply put, not every message in MPI must be received. The unfaireness allows that some messages may be never picked up even if they have reached the target process. In such a situation a message would eventually become an in-transit message. And according to non-overtaking property, this message must be received before passing any other messages between this pair of processes with this particular envelope. Although Event Logging is still able to identify and log such in-transit messages, we note that this may not be an expected scenario from the point of view of both Event Logging and the user.

3.4 Analysis and Optimization

3.4.1 Analysis

Being a checkpoint/recovery algorithm, Event Logging has the following three overheads: logging overhead, checkpoint overhead and recovery overhead. To optimize the performance of Event Logging, we examine them separately.

Logging overhead is introduced by recording the envelope during the message passing. For a single send (receive), the overhead is generally constant: creating a new node, saving the envelope in the node and linking the new node with the log. However, over the whole life of a program, the total logging overhead is determined by the program's communication volume. Suppose the logging overhead of a single message is $T_{\log}$, and there are in total N messages exchanged, the total logging overhead can be calculated by: $T_{\log} * N$. Further N can be substituted by $N_{CKPT} * T_c * f$,

where T_c is the checkpoint interval and f the message passing frequency is, N_{CKPT} is the number of checkpoints taken during the program execution.

Checkpoint overhead is comprised of the cost of identifying messages and the cost of creating a local checkpoint. To a process, the time of creating a local checkpoint is proportional to the size of data that needs to be saved. We denote this cost using function $T_{local}(datasize)$. As to the message identification cost, the matching process is as:

```
for x=0 to NumberOf( RECV_{q,j-1}{S_{p,i}} )
{
  for y=0 to NumberOf( SEND_{p,i}{R_{q,j-1}, R_{q,j}, R_{q,j+1}} )
  {
     if ( RECV_{q,j-1}^x{S_{p,i}} == SEND_{p,i}^y{R_{q,j-1}, R_{q,j}, R_{q,j+1}} )
     {
        remove RECV_{q,j-1}^x{S_{p,i}} and SEND_{p,i}^y{R_{q,j-1}, R_{q,j}, R_{q,j+1}};
        quit y loop;
     }
     else {
        y=y+1;
     }
  }
  x=x+1;
}

for x=0 to NumberOf( RECV_{q,j}{S_{p,i}, S_{p,i+1}} )
{
  for y=0 to NumberOf( SEND_{p,i}{R_{q,j}, R_{q,j+1}} )
  {
     if ( RECV_{q,j}^x{S_{p,i}, S_{p,i+1}} ) == SEND_{p,i}^y{R_{q,j}, R_{q,j+1}} )
     {
```

remove $RECV_{q,j}{}^{x}\{S_{p,i}, S_{p,i+1}\}$ *and* $SEND_{p,i}{}^{y}\{R_{q,j}, R_{q,j+1}\}$*;*

quit y loop;

}

else {

$y=y+1;$

}

}

$x=x+1;$

}

Note, no previous in-transit message envelopes will be logged $SEND_{p,i-1}\{R_{q,j}\} = RECV_{q,j}\{S_{p,i-1}\}$.

Using $N_{q,j-1}$ to denote *NumberOf(* $RECV_{q,j-1}\{S_{p,i}\}$ *)*, $N_{q,j}$ to denote *NumberOf(* $RECV_{q,j}\{S_{p,i}, S_{p,i+1}\}$*)*, $N_{p,i}$ to denote *NumberOf(* $SEND_{p,i}\{R_{q,j}, R_{q,j+1}\}$*)* and T_m denotes the matching time cost, the overhead of the above matching process is calculated by $T_m \frac{(2N_{p,i} - 2(N_{q,j-1} + N_{q,j}) + 3)(N_{q,j-1} + N_{q,j})}{4}$.

As for the recovery overhead, it is introduced by recovering the execution from the local checkpoint, plus replaying the in-transit messages and discarding the orphan messages upon the recovery. Suppose that restoring the process's execution state costs T_r, replaying an in-transit message costs T_i, discarding a repeated orphan message costs T_o, and there are N_i in-transit messages and N_o orphan messages saved in the local checkpoint. Also suppose N_r messages will be received after the process recovery. The recovery overhead is $T_r + T_i N_i + T_o \sum_{i=1}^{n} \frac{(N_o - e)(N_o - e + 1)}{2(N_r - i) + (N_o - e)\frac{(N_r - i) - (N_o - e)}{N_r - i}}$, where

$$e = \sum_{x=1}^{Min(i-1, N_r)} \frac{x C_{i-1}^{x} C_{N_r - i + 1}^{N_r - x}}{C_{N_r}^{N_o}}.$$

In general, among the three kinds of overheads, the checkpoint overhead caused by matching event logs takes the biggest part. Besides, as the logging overhead impacts the normal message passing operation, the user of Event Logging is advised to shorten the checkpoint interval. As for the recovery overhead, it is highly unpredictable and depends on the program itself. In the next section, we show a technique to minimize the logging overhead and checkpoint overhead.

3.4.2 Performance Tuning

Since the message envelope in MPI is a term consisting of three elements: <*rank, tag, comm*>, a hash function helps to reduce the memory overhead introduced by the logging. However, the function must be a perfect hashing function. In other words, for any two different message envelopes A, B, the hashing result is different: $f(A) \neq f(B)$. Moreover, considering that upon checkpointing the processes need to exchange logs, the hashing accelerates the exchange, because after hashing the log size is reduced.

Also, a matching optimization strategy based on the hashing can be developed. Shown in Section 3.4.1, and relying upon the fact that the event log is an unsorted link table, the matching process employs Sequential Search to pair up the send and receive logs. Recalling the Logging Independency Principle (LIP) discussed in Section 3.3.3, if there is a perfect hash function that generates numerical output, the original event log can be sorted by the hash value. For a sorted table, some well-known searching algorithms can help to reduce the average search length. For example, by applying Binary Search, the average search length is reduced to.

$$\log_2 P_{N_{p,i+1}}^{N_{q,j-1}+N_{q,j}} - (N_{q,j-1} + N_{q,j})$$

Furthermore, in cases where there are messages with the same envelope, an optimization technique is adding an extra flag to the event log specifying the number of the messages. For example, if message *m* has been sent *100* times in the checkpoint interval, the number flag is set to *100* and only one log of message envelope is kept, instead of creating *100* logs.

3.5 Conclusion

This chapter presents a new coordination checkpoint protocol Event Logging, which is based on the widely used Chandy-Lamport algorithm. Event Logging addresses the non-FIFO message passing challenge, which is the key to implement the coordinated checkpoint at the application-level.

Event Logging combines the merits of coordinated checkpoint and message logging. It improves the performance by logging only the message envelopes without the actual content, which is the main source of the overheads introduced by message logging.

This chapter also gives a proof of the correctness of the Event Logging algorithm. Analysis and performance tuning strategy are discussed as well.

Chapter 4

libELC – Application-level Checkpoint/Recovery Library for MPI

This chapter describes the design and implementation of libELC, an application-level checkpoint/recovery library for the C/MPI programs in a heterogeneous network. libELC is built from scratch to ensure the portability.

4.1 Overview

libELC is written in ANSI C language, employing Event Logging algorithm to provide the portable checkpoint/recovery facility for the C/MPI programs running in heterogeneous network of computers

Being a checkpoint/recovery tool for message passing programs, libELC is comprised of three parts: the uniprocess checkpoint/recovery module, the multiprocess coordination module and the message replay module. The uniprocess checkpoint/recovery module deals with the portable checkpoint/recovery for each individual MPI process. Given the portability concern of the heterogeneous network, the local checkpoint is created by using the application-level checkpoint approach, which saves and restores the process execution state in a platform-neutral manner. The multiprocess coordination module is responsible for orchestrating the parallel processes to form valid recovery lines, by using the Event Logging algorithm. Finally, the message replay module is active only at the time when the program restarts from a previous checkpoint, to replay the message passing.

4.2 Uniprocess Checkpoint/Recovery Module

4.2.1 Background and Challenges

Checkpointing a running program is usually considered as a low-level operation [61, 62]. In particular, programs live in the form of a process that is an entity that actually runs in its own space address in the operating systems. A process is usually composed of a unique identifier (PID), a register set, an address space and/or other certain

resources such as file descriptor, network connection, peripherals, etc. In this sense, a straightforward checkpoint approach is to capture the states of these physical elements. In the homogeneous case, checkpoint/restart mechanisms can simply and directly manipulate the state of a process without semantic analysis of that state. For example, the state of a UNIX process is simply the contents of its address space, plus the process control blocks (register values, file descriptor table, etc.). These entities are already conveniently available to the UNIX kernel, making the internal state of a UNIX process trivial to checkpoint. As long as the process is restarted on the same kind of UNIX system and processor on which the checkpoint was produced, the contents of the address space need not be interpreted by the kernel to restore the process. We call such a checkpoint/recover mechanism *native process checkpoint*.

From the point of view of implementation, the native process checkpoint facility can be classified into to two categories: system-level and library-level. The system-level solution such like EPCKPT [64], CRAK [63], CHPOX [65], provides the checkpoint capacity by building it into the operating system kernel. The patched operating system is able to save the state for a running process upon detecting some requesting signals. Usually the system-level approach does not require the user to manually trigger the checkpoint. Instead checkpoints will be generated in a preset period. Therefore, the checkpoint and recovery are completely transparent from the point of view of the users.

Different from that, the library-level approach is usually managed by an external library. Users are required to link theirs codes with the checkpoint library. In this case, programs may call the checkpoint library during the execution to save their states. Some popular examples of this kind of solution include libckpt [62], Esky [67], Condor [66], etc. Compared with the system-level approach, though the library-level approach may not be completely transparent to the programmer, it does not ask for modifications on the hosting platforms, which makes it lightweight and more preferred.

However, both of these two approaches suffer from the lack of portability given a heterogeneous environment. As discussed above, the idea behind the system-level checkpoint is based on the physical composition of a process. However, the address

space and kernel process control information would be meaningless if used to restart the process on a hetergeneous operating system implementation or architecture. Differences in data format, instruction sets, address space sizes (e.g. 32-bit vs. 64-bit addressing), and memory space structures will make the saved state completely unrecognizable at the restart time. In this sense, no platform-dependent solution would be popular in the NoC community.

Unfortunately the presence of inherent heterogeneity in a network of computers significantly complicates the design of a portable process checkpoint/restart mechanism. The additional complexity inherently introduced by heterogeneity is the main reason why few designs for such a portable facility have been developed to date. In this sense the main challenge for the design of the uniprocess checkpoint/recovery model in libELC is to develop a portable checkpoint library for heterogeneous processes.

4.2.2 Application-level Checkpoint

4.2.2.1 Basic Idea

Taking a different point of view from the native process checkpoint, the application-level approach [68-70] inspects the logical composition of a program, instead of the physical elements: data segment, stack, register, etc. In this case a program's execution state is co-determined by the program execution flow and program state.

A program's execution flow can be examined as a sequence of function calls. Supposing a C program that starts from *main()*, does some operations, and calls *function1()*, where the execution flow transfers to. Inside *function1()*, some operations are performed, and after these operations finish the execution flow returns to *main()*. Then, some work is done in *main()* until *function2()* is called. Similar to the call to *function1()*, the execution flow is passed to the called routine, i.e. *function2()*. *In function2()*, a call of checkpoint *ckpt()* is issued and the program state is saved.

In this example, at the time *ckpt()* is called, the program's execution flow is $main() \rightarrow function1() \rightarrow main() \rightarrow function2() \rightarrow ckpt()$. However, to restore the program running from *ckpt()*, *function1()* is needless since no previous execution

should be repeated. Therefore, the correct execution flow is $main() \rightarrow function2() \rightarrow ckpt()$. Upon recovery, the execution flow is reconstructed by skipping all instructions and directly jumping to *ckpt()*. For a C program, the jumping can be implemented by labelling every function call and using a *GOTO* statement. Suppose a new process is restarted from *main()*. Before executing any instructions, the process examines the saved failure-free execution flow and finds the next call is *function2()*. Then the process directly jumps to *function2()* by the statement:

GOTO LABEL_function2;

After the process is switched to *function2()*, it finds the target *ckpt()* and jumps to it:

GOTO LABEL_ckpt;

When the process re-enters *ckpt()*, it has known that the program is recovering. Then the process reloads the program state from the checkpoint file and resumes the execution. However, this requires that all function calls in the program must be uniquely labelled, *LABEL_function_name: function_name()*. Also, a flow table can be setup to examine the execution flow. Upon being called the corresponding function label is inserted onto the flow table. When returning from a function, the label will be removed from the flow table, indicating the execution flow has returned to the calling routine:

```
LABEL1:     EnterCall(LABEL1);
            function_call_1();
            QuitCall();
```

Hence, at the time a checkpoint is issued, the flow table will contain the exact program exection flow.

The program state can be seen as the composition of the values of program variables, the content of dynamically allocated memory (Heap) and other program-related properties, such like I/O descriptor, current working directory, etc. In a word, all these elements are presented in the form of memory blocks. So if the checkpoint library can locate all memory blocks and know their sizes, it is able to save and restore their states. In this sense, an application-level checkpoint inspects the events including variable definition, heap management and I/O operations. Upon detecting such an action, the checkpoint library records the related information in a dedicated stack,

called a shadow stack, to locate the corresponding memory blocks. The recorded information (typically the memory block address and size) serve as an index for the checkpoint library to save and restore the program state. For example, consider an integer definition statement in the C language: *int i*. The shadows stack records *i*'s address and size by inserting the following statement immediately after *i*'s definition:

PushOntoStack(variable_address,size);

Upon checkpointing (recovery), the process traverses the shadow stack, using the address to locate and save (restore) every memory block.

As it shows, the idea behind the application-level checkpoint/recovery is to apply some transformation onto the source code. The following pseudo code is a complete example of the function modification:

- **Original Code:**

```
function_name(parameter)
{
        //variable definition
        type variable_name;
        ……
        //execution statement
        …       …
        function_call_1();
        ……
        function_call_2();
        ……
        return;
 }
```

- **Modified Version:**

```
Function_name{parameter}
{
        // variable definition
        type variable_name;
        PushOntoStack(&variable_name,sizeof(variable_name));
```

```
    ……

    // execution jumping code
    switch (execution flow table)
        case (LABEL1): goto LABEL1;
        case (LABEL2): goto LABEL2;

    // execution statements
    ……
    LABEL1:     EnterCall(LABEL1);
                function_call_1();
                QuitCall();
    ……
    LABEL2:     EnterCall(LABEL2);
                function_call_2();
                QuitCall();
    ……
    return;
}
```

Since the program execution flow and program state is saved and reconstructed at the application level, this approach is completely system-independent. It is applicable to any platform as long as the ANSI C language is supported.

The idea of application-level checkpointing can be found in a number of efforts [68-70], among which the *PORCH* project [68] presents the most comprehensive implementation. *PORCH* ships with a C pre-processor, which is responsible for automatically transforming the C codes into a checkpoint-able version. Also, a universal portable data format is developed in *PORCH* to store the checkpoint data. However, *PORCH*, as well as most of the above application-level checkpoint systems, focuses on the uniprocess checkpoint/recovery. To our best knowledge, there is not an application-level checkpoint library so far developed for MPI programs, which provides a totally portable checkpoint/recovery facility in heterogeneous environments.

The closest effort to libELC is the C^3 system [22]. C^3 also uses application-level coordinated checkpointing. However, the difference between libELC and C^3 is that C^3 uses Message Tagging approach to coordinate the distributed processes; while libELC employs Event Logging algorithm. More important, C^3 does not aim at the portable checkpoint/recovery in a heterogeneous network. As a consequence, the checkpoint and recovery in C^3 can be only made on platforms that use the same memory layout.

4.2.2.2 Save and Recover Execution Flow in libELC

Section 4.2.2.1 explains the idea of the application-level checkpoint and recovery of the program execution flow. However, the technique described is the most simple and basic case. libELC takes a similar but more efficient approach.

Apart from recording the program execution flow outside of the function call like:

```
LABEL1:  EnterCall(LABEL1);
         function_call_1(....);
         QuitCall();
```

libELC modifies the function argument definition, registering the function itself in the flow table upon calling.

```
LABEL1:  function_call_1(...., LABEL1);
```

In other words, the execution flow is recorded by the called function, instead of the calling one. The definition modification reduces most of the modification work: the definition change is made only once, rather than for every function call.

Two interfaces are provided to support the definition modification: *OnCallEnter(), OnCallReturn()*. A simple example is:

- ***Original Code:***

```
Function_name(parameter)
{
        // variable definition
        ……
```

```
        // execution statements
        ……
        return;
}
```

- **Modified Version:**

```
Function_name(parameter, LABEL)
{
        OnCallEnter(LABEL);
        // variable definition
        …….
        // execution switch code
        ……
        // execution statements
        ……
        OnCallReturn(LABEL);
        Return;
}
```

One of the notable changes is the function parameter: the modified function has an additional *LABEL* argument. When calling the modified function, the calling label is passed in the parameter *LABEL*, which is further handed over to *OnCallEnter(LABEL)*. The function *OnCallEnter()* records *LABEL* in the flow table. On return, *LABEL* will be removed by the call of *OnCallReturn()*. In this sense, a function call is modified as follow:

- **Source Code:**

```
function_name(...);
```

- **Modified Version:**

```
LABEL_function_name:        function_name(..., LABEL_function_name);
```

The modification of function definition not only simplifies the programming, but also benefits the inline calls. An inline call is the composition of multiple function calls.

Typically, there exist two kinds of inline call: composition by operator and composition by parameter. A typical composition-by-operator inline call looks like:

*result=function1()+function2()*function3();*

And the composition-by-parameter looks like:

result=function1(function2(), function3());

To the inline calls, however, using only one *LABEL* parameter is not enough to specify the function execution sequence. For example, consider the following modified inline call:

LABEL: result=function1(function2(LABEL),function3(LABEL),LABEL);

Supposing a checkpoint is taken in the call of *function2(LABEL)*. According to the discussion above, *LABEL* is saved along with the flow table during the checkpoint. However, since all three calls share the same *LABEL* parameter, upon recovery the calling function would not know which function it should jump to. Therefore the program has to re-evaluate the whole expression. However, if the compiler is right-prior, *function3()* will be re-executed, which should not be repeated at all. Also, as the compiler allocates some temporary registers to save the intermediate result, it is impossible to access the return value of *function3()* upon creating a checkpoint in *function2()*. Thus, in order to guarantee the correct recovery, the checkpoint/recovery system needs to track the exact execution flow and save all the intermediate result for the inline calls.

The typical approach to this problem is to use intermediate variables to decompose the inline call. The call is disassembled into multi-statements, each of which contains a function call. Functions' return values are deposited in the intermediate variables, which are reassembled eventally to perform the original logic of the inline call. As each single function call has its own *LABEL* parameter, the system is able to tell the exact execution information. A decomposition example is shown as:

```
LABEL1:    t_result1=function3(LABEL1);
LABEL2:    t_result2=function2(LABEL2);
LABEL3:    result=function1(t_result_1,t_result_2,LABEL3);
```

As it shows, such a solution completely changes the code layout. Although the same program semantic is maintained, it makes the code much more difficult to read. On

the contrary, the proposed definition modification can avoid these problems by introducing another additional parameter *FID*, which indicates the evaluation order of the expression. Thus, instead of decomposing, the inline call is transformed into:

```
LABEL: result=function1(function2(LABEL,1),
                        function3(LABEL,2),
                        LABEL,0);
```

With the aid of *FID*, the system is able to select the time for creating a checkpoint. Basically, in the composition-by-parameter case, the choosing criterion is the outmost function call. In the above statement, the checkpoint is taken in *function1()*. This is because when starting *function1()*, the intermediate return value of *function2()* and *function3()* have been available as the parameters of *function1()*, which can be saved in *function1()*. By contrast, if the checkpoint were created in *function2()*, the process would be unable to retrieve the return of *function3()*. As to the composition-by-operator inline call, however, no call need be selected for checkpointing, since all returns are intermediate. For example, considering the following statement:

```
result=function1()+function2+function3();
```

None of these three functions is able to access the returns of the other two, so the checkpoint can be delayed until the next function call, in which *OnCallEnter()* will perform the postponed checkpoint.

During the recovery, libELC first switches to the *LABEL* statement and starts re-evaluating the expression. Upon entering a call, the function checks whether the parameter *FID* is 0. If not, the function quits without executing any instruction. Thus, the correct checkpoint/recovery of the program execution flow is guaranteed.

4.2.2.3 Save and Recover Program State in libELC

As mentioned in 4.2.2.1, the application-level checkpoint employs a dedicated shadow stack for all program variable, heap memory and I/O descriptors. In *libELC*, however, these elements are managed separately.

Case 1: Variables

In *libELC*, three interfaces are designed for the atomic datatype, pointer and structure:

```
void OnVarDef(void *pAddr,int size);
```

*void OnPtrDef(void **pAddr,int count);*

*void OnStrDef(void *pAddr,int size,char *pStrName);*

void OnStructureDef(StructureDesc descTemp);

Given a variable definition of basic type: type variable_name, the function is called with the arguments, e.g.: *OnVarDef(&variable_name,sizeof(type))*. In this way, the checkpoint library knows where is the variable (*&variable_name*) and how many bytes it spans (*sizeof(variable_name)*). Then, upon checkpointing, the *sizeof(variable_name)* bytes beginning from *&variable_name* will be written to the disk file, which contains the variable value. Note, in a heterogeneous NoC, some machines may use the little endian convention (PCs, Sun Sparc), so the variable bottom address is *&variable_name+sizeof(variable_name)-1*; however, some other architectures may use big endian design, in which case the variable bottom address is *&variable-sizeof(variable_name)+1*. libELC addresses this problem by detecting the running platform, and following the corresponding convention to calculate the variable bottom address.

Unlike other types, a pointer is a special kind of variable, which stores not a plain value but a memory address. Since the memory allocation strategy is different across platforms, the address of a variable could be different on different platforms in different runs. For example, in Linux:

int i;

*int *p=&i;*

The value of pointer p could be *0xbfffe5b4* in the first run, and *0xbffff834* in the second (Our experiments show that most systems including Windows, Solaris, BSD, AIX, use static memory allocation, whereas Linux adopts the dynamic allocation). So, in order to save (restore) pointers in a portable way, the pointers cannot be treated like "Save the value" as with other program variables. Instead, libELC replaces the address with a portable logical representation.

It is a rule of both the operating system and programming language that it is illegal for a pointer to point to a memory space out of the program's scope. In other words, a

point can only point to somewhere in the memory declared by the program. So, the pointer is transformed into the representation <*memory_block, offset*>. *memory_block* is a chunk of memory allocated either explicitly or implicitly by the program. Basically, there are three types of allocation: program variables, heap memory and function (Since a pointer can almost point to anything in a program, Section. 4.2.2.4 is a dedicated section discussing the checkpoint and recovery of pointers). *memory_block* is denoted by its ID: *mid*, and offset is calculated by *labs(pointer_value-&memory_block)*. There also is the problem of little endian or big endian memory addressing. The selection of + or – is determined by the running platform.

In general, upon defining a pointer, libELC issues a call to *OnPtrDef()*, in which the address of the pointer is recorded. During the checkpoint, libELC finds the pointer, retrieves its value, and performs a search to locate which memory block it points to. Then the pointer is represented by <*mid, labs(pointer_value-&memory_block)*>. To recover it, after the memory block is restored, the pointer is reset to the +/- *labs(pointer_value-&memory_block)* byte from *&memory_block* (details in Section. 4.2.2.4).

Among the three interfaces provided for variables, *OnStrDef()* is the most complicated. The reason is that a structure is a collection of variables: it may contain basic type variables, pointers, or even structures. When inline pointers exist, the saving and restoration of structure cannot be like the basic types. However, that requires the checkpoint library to locate the pointer elements defined in the structure. In other words, the checkpoint library needs to know the offsets of the pointers. Then, the location is calculated by *&structure_variable+/-pointer_offset*. To record the structure's definition, libELC provides a data type *StructureDesc* and an interface: *OnStructureDef()*. Upon defining a structure, StructureDesc is filled with the structure's information and passed to the following call of *OnStructDef()*. For example, some user-defined structure may look like:

```
struct Node {
        int ID;
        struct Node *next;
```

```
        struct Property prop;
}
```

StructureDesc is initialized and *OnStructDef()* is called like:

```
StructureDesc descTemp;

strcpy(descTemp.name,"struct Node");
descTemp.size=sizeof(struct Node);
descTemp.ptrCount=1;
descTemp.ptrOffset=(void *)malloc(descTemp.ptrCount*sizeof(void *));
descTemp.ptrOffset[0]=sizeof(int)-1;
descTemp.strCount=1;
descTemp.strOffset=(void *)malloc(descTemp.strCount*sizeof(void *));
descTemp.strOffset[0]=sizeof(int)+sizeof(struct Node *)-1;
descTemp.strName=(char *)malloc(descTemp.strCount*sizeof(char)*
                        MAX_STRUCT_NAME_LENGTH));
strcpy(descTemp.strName[0],"struct Property");

OnStructureDef(descTemp);
```

descTemp.name saves the structure's name and *descTemp.size* is the structure size. The element *ptrOffset* is a pointer to an integer array, which contains the offsets of each pointer defined in the structure type. *ptrCount* specifies the length of this array. Simliarly, the array pointer *strOffset* tells the offsets of every inline structure element, and *strCount* is the array's length. *OnStructureDef()* uses recursive calls to resolve the in-line structure elements. So that, by employing the *StructureDesc* data type and *OnStructureDef()* function, libELC is able to know the number and offsets of the pointer elements in the user-defined data types. Upon defining a variable of these data types, the function *OnStrDef()* uses the data type knowledge recorded by *OnStructureDef()* to locate the pointer elements. For each pointer element, *OnStrDef()* calls *OnPtrDef()* in order to translate the memory address to its representational format at the checkpointing time. Considering a variable definition of the type - *struct Node*:

struct Node new_node;

Then the function *OnStrDef()* is called as:

OnStrDef(&new_node,sizeof(new_node), "struct Node");

Inside of the call, *OnStrDef()* searches for the name *"struct Node"*. When found, it retrieves the information of the pointer elements' number and offsets, and calculates the pointer address by *&new_node+/-pointer_offset*. Then, the address is passed to *OnPtrDef(&new_node+/-pointer_offset)*, and saved for the translation.

Case 2 Heap Memory

Besides the program variables, heap memory is another important element in the program state. Unlike the static variables, the operating system uses the heap to allocate memory for a run-time request. In the C language, heap management is made by calling the following functions: *malloc()*, *calloc()*, *realloc()* and *free()*. Since the execution of these function calls is determined at the run-time, libELC defines the following wrapper functions to detect the dynamical allocation of the heap:

*void *ELC_malloc(size_t size);*
*void *ELC_callac(size_t nmemb, size_t size);*
*void *ELC_realloc(void *ptr, size_t size);*
*void ELC_free(void *ptr);*

These wrappers provide the same behaviour as the original functions. However, one more job they do is to record the address and size of the allocated memory block. For example, *malloc()* returns the head address of a memory block from the heap. In the wrapper *ELC_malloc()*, the head address and block length *size* are pushed onto the shadow stack. Upon checkpointing, libELC locates the memory block and saves it by writing size bytes to the disk. Moreover, when the user tries to de-allocate a memory block, *ELC_free()* removes the corresponding entry from the shadow stack. *ELC_realloc()* updates the shadow stack with the memory adjustment.

However, such wrapper functions take no consideration of pointers. They are incapable of saving and restoring the pointers in the heap memory (called heap pointers). This results from the different memory architectures of different systems. In order to deal with the pointers in the heap, following each wrapper function call,

libELC inserts a call to the corresponding variable definition interface. If the allocated heap memory is pointer type *void **ptr=(void **)ELC_malloc(4*sizeof(void *))*, *OnPtrDef()* is called after the allocation with the parameter *OnPtrDef(ptr, 4*sizeof(void *))*. If the memory is allocated in some user-defined type *struct structure_name *ptr=(struct structure_name *)malloc(sizeof(struct structure_name))*, *OnStrDef()* is called as *OnStrDef("struct structure_name", ptr, sizeof(struct structure_name))*. In both cases, libELC records the locations of all the allocated heap pointers, which will be translated, saved and recovered separately from the heap memory. In this way, libELC creates the portable checkpoint/recovery to the dynamically allocated memory.

Case 3 Program-related Properties

Besides the variables and heaps, the program state is also influenced by some other properties, such as I/O operations, file systems and signal handlers, which are necessary parts of the checkpoint and recovery. However, to create portable checkpoints for programs running across heterogeneous platforms, the source code must be also portable. In other words, even if restarting the program execution on another machine with a different architecture, the recovery procedure should be no more than copying the program source, data files and the checkpoint files to the target machine, compiling the source to generate a local executable and simply re-launching the executable. In such a case, the program should contain no system-specific resources (like signal handlers) or environment-related property (working directory, process identifier PID). In this sense, the main concentration in this section is to deal with the program I/O state.

Basically, there are three types of I/O: file (file I/O), terminal (standard I/O) and sockets. As the I/O operation is so ubiquitous, it is critical to reconstruct the I/O state upon recovery. Generally, all these three I/O types are built on files and file descriptors. A file descriptor is just a link between the process and the corresponding source. File descriptors associated with normal files should be reattached to those files when the application is restarted. File descriptors associated with a terminal (for standard IO) should be attached to the terminal upon recovery. Among the three, sockets pose the most serious challenge, because the recovery of a socket connection involves two machines. Fortunately, to the depth libELC concerns, MPI provides

sufficient communication facility. The socket is managed by the MPI runtime environment implicitly: once the MPI program restarts, the connection will be re-established automatically.

To recover the access of a normal file, the flag set of the file descriptor, the access mode and the file offset must be restored. libELC provides the following wrapper functions:

*FILE *ELC_fopen(const char *path, const char *mode);*
*void OnFileDef(FILE *fp);*
*int ELC_fclose (FILE *fp);*
int ELC_fcloseall();

The function *ELC_fopen()* records the filename and access mode. *OnFileDef()* translates the stream returned by *ELC_fopen()* to file descriptor. Upon checkpointing, the file descriptor value is associated with the filename and saved into the checkpoint file. In order to restore the file access, libELC uses *freopen()* to redirect the file descriptor to the file. As to the file offset, the system calls *ftell()* to get the current offset, saves the offset in the checkpoint file, and uses *fseek()* to relocate after recovery. The function *ELC_fclose()* will notify the system to remove the corresponding entry; *ELC_fcloseall()* clears all the file entries. However, we emphasize that it is the user's responsibility to guarantee the file and directory are in place upon recovery.

Moreover, if the program changes something in the file after a checkpoint, the checkpoint/recovery library should be able to rollback the file to the state of the checkpointing time. Since the file operation is usually non-volatile, libELC choose a coarse-grain solution for the rollback: when checkpointing, libELC backups all open files. The backup is done by closing the files, copying and reopening them. To rollback, the system just simply replaces the original file with the backup.

As to the checkpoint/recovery of the file descriptor associated with terminal, libELC judges which file descriptor is connected to the terminal (by calling *isatty()*) and the name of the connected terminal (by calling *ttyname()*).Similarly to normal files, the file descriptor value is saved with the terminal name and attribute (using *tcgetattr()*).

Upon recovery, the system reattaches the same file descriptor onto the terminal, and resets the terminal attribute using *tcsetattr()*.

4.2.2.4 Pointer

Being the most knotty part, pointers pose two main challenges: (1) as a pointer can point to almost anywhere in the program memory space, the pointer translation must care for various scenarios; (2) since the pointers are saved in the order of their definitions, there arise problems when a pointer of pointer is recovered before its reference. This section is dedicated to illustrate how libELC tackles the above problems.

First, the value of a pointer is the address of one of the following three types of memory block: variables, dynamically allocated memory (heap) and functions. The difference is which segment the memory block stays in. Generally, the variables are stored in the stack, where the memory allocation is done automatically by the operating system upon defining a new variable. On the other hand, heap is allocated at run time by explicit memory requests. The address of a function is static, which is always allocated at the text segment during the compilation.

To the variables and dynamic memory, every memory block is assigned a unique ID, *mid*. The assignment is done together with the allocation of the memory block. To the variable, the assignment time is the time of defining a variable; to the heap, the ID is assigned when calling *ELC_malloc()* and *ELC_calloc()*. The result of such a scheme is that the ID assignment is sequential, in the same order of the variable definition and heap allocation. Moreover, the assignment is dynamical: on the return of a function, the IDs of the function's local variables are reclaimed along with the de-allocation of the variable memory, and will be reassigned for the following program variables. However, the IDs of memory blocks in different segments, stack and heap, are maintained independently (stack- *sid*, heap- *hid*). As variables stay in the stack, the logic representation of a variable pointer is <*sid, offset*>, and the pointer of dynamically allocated memory is represented as <*hid, offset*>.

For function pointers, libELC employs another way to represent them. Basically, the address of a function is determined by the compiler. In other words, the memory

allocation for user-defined functions is static. Therefore, it is feasible to calculate the memory address of a function by using the offset from the function to a known position in the text segment. The *main()* function is such an obvious example. Given the address of the main function &main, the function pointer is positioned by <*pointer_value* - *&main*>. On recovery, the system uses the offset to reset the pointer. However, to be compatible with the variables and dynamic memory, the function pointer is also set into the format <*ID, offset*>. But, all function pointers use the same default ID 0. Then a function pointer is represented as <*fid=0, offset=pointer_value-&main*>.

Another challenge the pointer poses is the checkpoint/recovery order. Supposing a program defines two pointers in the order: *ptrA, ptrB*. The first pointer *ptrA* is a pointer of pointer, which contains the address of the second pointer *ptrB*. Since the pointer is a type of user-defined variable that locates in the stack, libELC assigns the *sid 1, 2* to the two pointers. During the checkpoint, *ptrA* is represented as <*sid=2, offset=0*> and saved before *ptrB*. On recovery, *ptrA* is also reset before *ptrB*. This creates the problem that *ptrA* cannot find its pointee, because *ptrB* has not been recovered at this time. Such a relation is called *Recovery Dependency*. According to Recovery Dependency, a pointer can be recovered only when its pointee has been restored (the dependency of the pointer is satisfied).

To satisfy the dependency, libELC requires the recovery to be performed in the following sequence:

1) *Recover all variables except pointers (without the inline pointers in the structure types);*
2) *Restore the heap memory (without the inline pointers);*
3) *Reset the function pointer;*
4) *Recover all other pointers except the pointer of pointer;*
5) *Finally, recover the pointer of pointer.*

Moreover, the restoration procedures of program states are postponed until the recovery of execution flow finishes. To illustrate the reason, consider an example: a pointer *ptr* in the function *A()* points to a heap block *heap* allocated in function *B()*.

The execution flow is $A() \rightarrow B() \rightarrow ckpt()$. So on restart, when recovering function *A()* and the pointer *ptr*, function *B()* and *heap* have not yet been restored. This breaks the dependency between the pointer *ptr* and the heap block *heap*, because *ptr* is recovered before its pointee. Thus, libELC must perform the execution flow recovery before the program state recovery, in order to satisfy the dependencies.

4.2.2.5 A Note on Portability

libELC can only provide the portable checkpoint/recovery facility for C/MPI programs, which are themselves portable. The checkpoint of a non-portable MPI program cannot be used for recovery on a heterogeneous machine. In other words, the MPI program cannot have any information or system calls, which are runtime environment related.

Also it is necessary to note here that the application-level checkpoint approach has not been as mature as the system-level checkpoint mechanism. As a consequence, the current implementation of libELC faces several restrictions that are common to application-level checkpoint systems:

1) **Union Type**: *Currently libELC is unable to save and restore the value of a union-type variable. However, to the best knowledge of the author, PORCH* [22] *provides support to the union type. And such support will be included in the next version of libELC.*

2) **Register Variable**: *According to the ANSI C standard, a variable can be defined as register variable. Such variables will be allocated on some registers, rather than in the memory space. libELC cannot give support to the checkpoint and recovery of register variables.*

3) **Variable Definition**: *In a C program, variables can be defined anywhere starting with a curly bracket {}. However, in libELC, no variable is allowed to be defined during the execution. In other words, variable definitions must be placed in front of execution statements. Other wise, undefined errors may be caused by the GOTO statements upon recovery. Also, if the source program contains some*

static variables (or global variables), corresponding calls to OnVarDef() should be inserted in the main() function.

4) **Universal Data Format**: *The current libELC version does not address the problem of heterogeneous data representation. It assumes that the checkpoint file generated on one machine can always be recognized on another platform. Obviously this is not a strong argument given a heterogeneous environment.*

4.3 Multiprocess Coordination Module

This section describes the technique used in the libELC coordination module to implement the Event Logging algorithm. The coordination module is comprised of three packages: MPI Wrapper Package (MWP), Message Identification Package (MIP) and Message Logging Package (MLP).

4.3.1 MPI Wrapper Package

A wrapper function is a popular technique for extending the system feature. By including the original function, a wrapper provides users the additional functionality in a transparent way. Users have no knowledge about how the function is implemented, and do not need to worry about the details of using the extra service. In order to hide the checkpoint/recovery and multiprocess coordination procedure from the users, libELC provides a set of wrappers for the MPI routines.

Although making most of the checkpoint/recovery details transparent, libELC still exposes one explicit function to let users invoke the checkpoint: *ELC_DoCKPT()*. The user may insert calls to *ELC_DoCKPT()* at the place they want to start a checkpoint. It does not require all processes in the program to call this checkpoint function at the same place. libELC is responsible for broadcasting the checkpoint requests in the program. If multiple checkpoint requests are detected by a process at the same time, only one is taken, the others will be ignored. However, if two processes run the same code and issue a same checkpoint request but at the different time due to performance difference, the two checkpoint requests will be taken separately. In other words, two recovey lines will be created.

ELC_DoCKPT(): every calling process *p* performs a local checkpoint, broadcasts a checkpoint request to each of all other processes and initializes a request table. The checkpoint request is packaged with *p*'s send event log. For example, supposing process *p* is in the i^{th} checkpoint interval. When calling *ELC_DoCKPT()*, *p* creates checkpoint $C_{p,i}$ and send the event log $SEND_{p,i}\{R_{q,j-1}, R_{q,j}, R_{q,j+1}\}$ to all other processes $(q = 1,2,...,n, q \neq p)$. As soon as it returns from *ELC_DoCKPT*, the calling process resumes the normal execution.

```
ELC_DoCKPT()
{
        do a local checkpoint;
        for i=1 to all processes
        {
                if ( process i is not the calling process )
                        broadcast a checkpoint request to process i;
        }
        initialize a request table;
}
```

To the other processes that have not called *ELC_DoCKPT()*, they will be introduced to the checkpoint by the wrapper *ELC_MPI_Send(), ELC_MPI_Recv()*. In these two wrappers, before performing the original communication, the wrapper probes whether there is pending checkpoint request. If yes, the checkpoint requests will be received. Depending on whether this checkpoint request is the first request being intercepted in the current checkpoint interval, the process may take two different actions:

- **Action 1**: If the checkpoint request is the first one, the process behaves in the same manner as when *ELC_DoCKPT()* is called: it creates a local checkpoint, broadcasts checkpoint requests containing this process's send event log to all others and then invokes the MIP (details in Section. 4.3.3). In general, MIP is responsible for identifying the in-transit messages and orphan messages by using Event Logging algorithm. At the time the identification completes, the process will have the envelopes for in-transit and orphan messages. However, note that

the identification is taken only on the message passed between the checkpoint-requests-sending process and this checkpoint-request-receiving process. The orphan message envelopes are written onto the disk as a 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 rank of the processes that have sent the checkpoint requests. After finishing this action, the process triggers MLP.

- **Action 2**: If the checkpoint request is not the first one detected, the process directly starts the MIP. The identified orphan message envelopes are saved in the checkpoint files. The in-transit message envelopes are passed to MLP, which saves the logged in-transit messages into the checkpoint files. Then the request's source is marked in the request table. The process counts the number of checkpoint requests it has received. When the process finds it has gathered requests from all other processes, the process marks its local checkpoint as finished.

In all, the wrapper function *CKPT_MPI_Recv()* looks like:

```
CKPT_MPI_Recv (incoming message envelope)
{
    probe whether there is pending checkpoint request (CKPT_Request);
    if (yes)
    {
        receive the CKPT_Request;
        if this CKPT_Request is the first one
        {
            do a local checkpoint;
            for i=1 to N
            {
                if (i!=my rank)
                    send a CKPT_Request with send logs to process i;
            }
            invoke the MIP;
```

```
            save the orphan message envelopes to the checkpoint file;
            pass the in-transit message envelopes to the MLP;
            mark the CKPT_Request source onto the request table;
        }
        else {
            invoke the MIP;
            save the orphan message envelopes to the checkpoint file;
            pass the in-transit message envelopes to the MLP;
            save the logged in-transit messages into the checkpoint files;
            mark the CKPT_Request source onto the request table;
            count the number of CKPT_Request;
            if (got CKPT_Request from all other processes)
                mark the local checkpoint finished;
        }
        MPI_Recv(incoming message envelope);
    }
```

4.3.2 Message Identification Package

MIP is invoked to help the process identify the in-transit and orphan messages at the time when the process intercepts a checkpoint request. Recalling the algorithm discussed in Section. 3.3.2, the MIP tries to identify the messages by pairing up the send logs bound with the checkpoint request with the target's receive logs.

Supposing process q gets a checkpoint request from process p during the i^{th} checkpoint interval ($i>1$). In this scenario, q holds the event logs of the messages received from p: $RECV_{q,i}\{R_{p,i-1}, R_{p,i}, R_{p,i+1}\}$, 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: $SEND_{p,i}\{R_{q,j-1}\}$ and $RECV_{q,i-1}\{R_{p,i}\}$. Also, in MPI a message is labelled by the tern <*rank, tag, comm*>. MIP is invoked when process p intercepts the send logs from process q, so in this case the sender/receiver's ranks are known: p and q, and the message envelope is reduced

to <*tag, comm*>. Thus a pair of send/receive logs are matched if the two logs have the same envelope.

Suppose $N_{q,i,r}$ is the number of the logs in $RECV_{q,i}$; $N_{p,i,s}$ is the log number of $SEND_{p,i}$; $N_{p,i-1,s}$ is the log number of $SEND_{p,i-1}$; $N_{q,i-1,r}$ is the log number of $RECV_{q,i-1}$; $SEND_{p,i}^{x}$ denotes the x^{th} message envelope in $SEND_{p,i}$; and $RECV_{q,i-1}^{y}$ denotes the y^{th} envelope in $RECV_{q,i-1}$. Then the matching process takes the following three steps:

1) Clear the in-transit message logs: $SEND_{p,i-1}\{R_{q,i}\} = RECV_{q,i}(S_{p.i-1})$:

```
for x=0 to N_{p,i-1,s}
{
    for y=0 to N_{q,i,r}
    {
        if (SEND^x_{p,i-1} == RECV^y_{q,i})
        {
            remove SEND^x_{p,i-1} and RECV^y_{q,i};
            quit loop y;
        }
        else {
            y=y+1;
        }
    }
    x=x+1;
}
```

(2) Remove the orphan message logs: $RECV_{q,i-1}\{S_{p,i}\} = SEND_{p,i}(R_{q,i-1})$**.**

```
for x=0 to N_{q,i-1,r}
{
        for y=0 to N_{p,i,s}
```

```
        {
            if ( RECV^x_{q,i-1} == SEND^y_{p,i} )
            {
                remove RECV^x_{q,i-1} anew SEND^y_{p,i} );
                quit loop y;
            }
            else {
                y=y+1;
            }
        }
        x=x+1;
    }
```

(3) Remove the intra message logs: $RECV_{q,i}\{S_{p,i}\} = SEND_{p,i}(R_{q,i})$ **.**

```
for x=0 to N_{p,i,s} - N_{q,i-1,r}
{
    for y=0 to N_{q,i,r} - N_{p,i-1,s}
    {
        if ( SEND^x_{p,i} == RECV^y_{q,i} )
        {
            remove SEND^x_{p,i} and RECV^y_{q,i} ;
            quit loop y;
        }
        else {
            y=y+1;
        }
    }
    x=x+1;
}
```

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

4.3.3 Message Logging Package

MLP logs the in-transit messages using the envelopes identified by MIP. MLP is implemented in two different forms. The first 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 they may not have been picked up yet. So in this case MLP just posts a receive (MPI_Recv) for each in-transit message envelope.

```
ELC_Logging()
{
        For each in-transit message envelope
        {
                MPI_Recv(in-transit message envelope);
                save the in-transit message into the checkpoint file;
        }
        return;
}
```

The second version of MLP does not rely on the FIFO property of the lower communication level. In that case, MLP checks if the incoming message is in-transit or not. If so, the message will be logged. If not, nothing happens.

4.4 Message Replay Module

Being a necessary part of the library, the message replay module helps the MPI program to reconstruct the previous communication state from the recovery line, by replaying the message passing. Considering of the difference between in-transit and orphan messages, the replay module consists of two elements: in-transit message replay and orphan message replay.

4.4.1 In-transit Message Replay

Shown in Figure 1, an in-transit message is a message that is sent before the checkpoint, but received after it. This results in the problem that the message is lost after restarting from the checkpoint, since the target process' checkpoint does not include this message and the source process will not repeat the sending. So the recovery procedure must restore the communication state by either making the source resend the message, or making the message re-available in the receiving buffer. Given that in checkpointing, the in-transit message is logged at the receiver side, libELC chooses the latter scheme. However, being a portable application-level checkpoint/recovery library, libELC cannot access neither the OS I/O buffer nor the MPI internal buffer to restore the in-transit message. On the contrary, the message has to be fed to the receiving call without system's intervention.

The solution adopted in libELC is the wrapper function. Upon recovery, the process loads the in-transit messages from the checkpoint and pushes them into a pending message queue (PMQ). For each receiving call after the recovery, the wrapper function first checks whether PMQ is empty. If not, the function tries to find a matching message in the queue. When it finds the first matching message, the wrapper copies the message content to the user-specified address, removes the message from the queue and returns without performing the real MPI receiving. Otherwise, the wrapper function will call MPI to do the receiving. Note, *CKPT_MPI_Recv ()* is used instead of the original *MPI_Recv()* (shown in Section. 4.3.2):

```
IN_MPI_Recv(receiving buffer, incoming message envelope)
{
        if (PMQ is not empty)
        {
                search PMQ;
                if (find a matching message)
                {
                        copy the message content to receiving buffer;
                        remove the matching message from PMQ;
                        return;
```

```
            }
        }
        CKPT_MPI_Recv(receiving buffer, incoming message envelope);
        return;
    }
```

4.4.2 Orphan Message Replay

As shown in Figure 2, an orphan message is sent after the source's local checkpoint but received before the target's local checkpoint. However, if the recovery line consists of these two checkpoints, it leads to the scenario in which the source process resends the orphan message after recovery, but the target would no longer need it. Generally, the existence of orphan message is not as serious as the in-transit message since the orphan messages just wastes the buffer space mostly. But in some cases, the orphan message may break the program's communication semantics.

Consider a case where process p sends two messages of the same envelope m in the order m_1, m_2 to process q. Moreover, before these two sends, p has just performed a local checkpoint $C_{p,i}$. Furthermore, assume q receives the message in the order m_1, m_2. In contrast to the sender, q triggers its local checkpoint $C_{q,i}$ between the two messages. In this scenario, m_1 becomes an orphan message included in q's local checkpoint. Then, after recovering from $< C_{p,i}, C_{q,i} >$, p repeats these two sends, making m_1, m_2 both available to q. But q will only post one receiving request, intending to get m_2. According to the non-overtaking property of MPI, the receiving request is satisfied with m_1, not m_2. As q gets another message, its subsequent execution might be changed.

In order to guarantee the correct communication semantics, the checkpoint/recovery system must clear any dangers caused by the orphan message. Basically, there exist two approaches: banning the resending at the sender side; or discarding the resent orphan message from the receiving buffer. Since the event log of orphan messages is identified at the receiver side, libELC chooses to discard at the receiver side.

Upon trying to receive a message after recovery, the process first checks whether there is a matching entry in the orphan message queue (OMQ). If so, the process retrieves a message from the buffer, then discards it. The process repeats the above actions until no matching entry can be found in OMQ. Then the process is able to get the "real" message.

```
ELC_MPI_Recv(receiving buffer, incoming message envelope)
{
        while (there is a matching entry in OMQ)
        {
                MPI_Recv(receiving buffer, incoming message envelope);
        }
        IN_MPI_Recv(receiving buffer, incoming message envelope);
        return;
}
```

Note, this is the final version of the wrapper function in libELC for *MPI_RECV()*.

4.5 Support More Feature of MPI

4.5.1 Collective communication

Besides point-to-point mode, the other important communication pattern of MPI is the collective. According to MPI, a collective operation requires all processes in the communicator to call it to finish. A natural result is that a valid recovery line must not have a collective operation bisecting it. For example, if *p* executes a barrier call after its local checkpoint and *q* calls the barrier operation before its local checkpoint, the recovery line comprised of these two local checkpoints is not valid. Since, upon recovery, *p* will repeat the barrier but *q* won't, that makes *p* become a zombie process. To supporting the collective communication, we wrap the original collective routines, adding a selective procedure before it.

- **Required Data:**
  seq=the local current checkpoint sequence
  sum=the sum of checkpoint sequence of all processes,
  nProc=the number of processes

- **Selective Procedure:**
    1. *A reduction with the sum option is executed.*
    2. *As each checkpoint is assigned a monotonically increasing sequence. The sum of this sequence under a coordinated checkpoint algorithm should be N times of the number of involving processes, where N is the last local checkpoint sequence. The process compares the expression: Sum/nProc with seq. However, there are 3 possibilities:*
        i. *Sum/nProc=seq; this means no checkpoint is in progress at the time of the execution of this collective call.*
        ii. *Sum/nProc>seq; some other processes have started a new checkpoint, but this process has not received any checkpoint request.*
        iii. *Sum/nProc<seq; this process has started a new checkpoint, but still some processes have not completed it.*
    3. *If the result is i and iii, no checkpoint is started; otherwise, those processes that get the result of Sum/nProc>seq, need to create a checkpoint immediately. As in the point-to-point case, after the local checkpoint finishes, the process sends out current logs to all other processes.*

4.5.2 Non-standard-mode Point-to-point Communication

Besides the standard mode point-to-point communication *MPI_Send*/*MPI_Recv*, MPI provides other several patterns to users to satisfy their specific requirement. Among them, however, buffered, synchronous and ready options need no special treatment. The things we care about are the non-blocking mode, *MPI_Sendrecv* (same as *MPI_Sendrecv_replace*) and *MPI_Send_init*/*MPI_Recv_init*.

Non-blocking mode: According to the semantic of non-blocking communication, message might have not been sent or received even if the routine has returned success. Upon checkpointing, there are naturally two might results for a non-blocking call: finish or not. As the send (*MPI_Isend*), we don't need to worry about whether the sending has completed or not before checkpoint. Regardless whether it is not, the *MPI_Isend* operation will not be repeated after recovery. So the message must be an

in-transit one to the target process. The send event log of non-blocking send is created in the same way as for the blocking routine. Once the target logs the message in its checkpoint, there need not be any further concern about a replay of the send. Regarding the non-blocking receive (*MPI_Irecv*), things are different. Supposing the target process posts a non-blocking receive request, and then a checkpoint is taken. There are two possibilities. First, the receiving buffer has been filled with the incoming data before the checkpoint. In this case, the data will be kept in the checkpoint and receiving completes before the checkpoint. On the other hand, if the checkpoint occurs before the buffer changes, data will not be saved. For this problem, our solution is that after checkpoints finish, the system still keeps an eye on the operation (*MPI_Irecv*). When the receiving completes (probe or wait return true), we update the checkpoint file with the current buffer's contents (This is what the target needs). Therefore, the checkpoint will always contains the required data upon recovery. Regarding the other routines, such like *MPI_Sendrecv*/*MPI_Sendrecv_replace*, they are just composed of the basic non-blocking operations. For *MPI_Send_init*/*MPI_Recv_init*, the event log will be created with the call of *MPI_Start* or *MPI_Startall*.

Of note is the ***MPI_Request*** object. Since it is used to detect the completion of corresponding communications, we have to rebuild it after recovery. However, according the discussion above, a much simpler way is to return queries of the legacy request as true, since both for the sending and receiving, the data will be saved into the checkpoint file when the communication is done.

Here, we don't need to worry about the correct semantics of application communication. If another blocking point-to-point operation is executed between this non-blocking operation and the checkpoint, the return of blocking call has already implied the completion of non-blocking, according to MPI non-overtaking property.

4.5.3 Communication Wildcard

A tougher feature of MPI is the wildcard used in point-to-point communication. A receiver may specify *MPI_ANY_SOURCE* for the value: source, and/or *MPI_ANY_TAG* for the value: tag, indicating that any source and/or tag are

acceptable, reduces the message-passing result of wildcards sometimes run-time dependent.

So far, relying on the non-overtaking message-passing property of MPI, we have successfully supported *MPI_ANY_TAG* by logging the in-transit messages according their relative sending order. In the preceding deductions, we already get the in-transit message list: $SEND_{p,i}\{R_{q,i+1}\}$, and this log is just built up by the sending sequence in the source's current checkpoint interval. Even if there is a message in the list that has the same envelope with the sending after the checkpoint and the target process uses *MPI_ANY_TAG* to receive, it is always the first matching message in this list is picked up.

To implement *MPI_ANY_SOURCE*, a simple case is: if p and q send two messages to r, which have the same envelope (except the value of source), and r issues a receive with wildcard: *MPI_ANY_SOURCE*, the target process chooses to receive the message according to the relative sending order: the result of process *r*'s receiving post (*MPI_Recv*) fully depends on which message was sent first. Thus when *p*, *q* and *r* restart the execution from a recovery line before this message passing, the result might be different from execution in the absence of a failure. Unfortunately, MPI lacks a mechanism to retrieve the sending order of messages from different sources, making it impossible to reproduce the exactly same internal message-passing state among multiple processes, which is necessary to support the wildcard.

However, since any receiving call with wildcard will return one definite message finally and we can get the details of the message's envelope (*rank, tag*) after the wildcard receiving, a non-deterministic receiving event can be converted into a deterministic result. Put simply, after the receiving of an in-transit message with *MPI_ANY_SOURCE* completes, we retrieve the matching message's property from the *MPI_Status* variable specified in the call and log this receiving as a deterministic event without wildcard. Moreover each process maintains a counter to record the receiving sequence number in the current checkpoint interval (the n^{th} message received in the current interval). Upon receiving an in-transit message, the sequence number will be logged with message content and message envelope. After recovery,

the process may decide to receive the message either from the MPI buffer, or get it from the in-transit message log by comparing the receiving sequence counter with the logged message's sequence number.

4.5.4 Derived Datatype

Besides the default basic datatypes bound with C and FORTRAN, MPI allows the programs to dynamically define/destroy their own datatype to facilitate transmitting complex data structures. As a very important feature of MPI, our approach should include the support of such datatypes.

Upon a message passing, MPI matches the send and receive with not only the message envelope, but also the datatype specified at two sides. As to the derived datatype, MPI does the comparison by checking the parameter and construction mode (contiguous, index, vector and structure, etc). For example, a derived type that contains only one integer is actually as same as the default *MPI_INT*. So, even if the sender and receiver construct the same derived datatype with different names, MPI still can make the correct matching. As the solution to the derived datatype, when the process passes a message of non-default types, the field *Datatype* of the log entry will be filled with a reference which points to the entry in the list that records the currently existing derived datatype. Since the recovering of MPI program is in charge of restoring the opaque MPI property (derived datatype, communicator, group, etc), we don't need to worry that the receiving of a derived-datatype message after recovery will encounter an undefined-datatype error.

4.6 Conclusion

This chapter presents the implementation detail of libELC, an application-level checkpoint/recovery library for MPI programs running in a heterogeneous network. The main challenge for developing a portable checkpoint/recovery facility is handling the portable uniprocess checkpoint/recovery technique and the application-level process coordination. For the former problem, libELC adopts the application-level checkpoint technique, which examines the running state of a program from its logic composition, rather than the physical elements. The saving and restore of the execution state is done by apply transformation onto the source code. The obvious benefit of the application-level checkpoint is the outstanding applicability of our library on any platforms that give supports to ANSI C language.

To the process coordination problem, libELC employs the Event Logging algorithm. The implementation of Event Logging uses wrapper functions. The advantage of Event Logging algorithm is its inter-operablity with various MPI implementations.

Also, note that libELC is the first checkpoint/recovery library so far that is built on top of the MPI standard and provides a portable checkpoint/recovery facility in a heterogeneous network. Although C^3 [22] also uses the application-level checkpoint technique, the implementation is not totally portable due to the lack of pointer translation.

Chapter 5

Experiments and Evaluation

This chapter is dedicated to experimental evaluation of the libELC library and the Event Logging algorithm.

5.1 Experiment Environment

Since the main goal of the design of Event Logging and libELC is portability across various MPI implementations, we chose two popular MPI distributions for the following experiments: MPICH-1.2.6 and LAM/MPI 7.0.4, running on various machines and OSes.

Table 3 shows the configuration of the heterogeneous network of computers used in the tests:

Machine	OS	CPUs (Mhz)
csserver.ucd.ie	Linux 2.4.18-10bigmem	4@498
csultra01.ucd.ie	SunOS 5.8	1@440
csultra02.ucd.ie	SunOS 5.8	1@440
pg1cluster01.ucd.ie	Linux 2.4.18-10smp	2@1977
pg1cluster02.ucd.ie	Linux 2.4.18-10smp	2@1977
pg1cluster03.ucd.ie	Linux 2.4.18-10smp	2@1977
pg1cluster04.ucd.ie	Linux 2.4.18-10smp	2@1977
csa007b4pc5.ucd.ie	Linux 2.6.0	1@930
csa007b3p2ucd.ie	FreeBSD 5.2.1	1@500

Table 3. Machine configuration.

Note, not all of these machines were involved in every test. The configuration of each test run is listed with the used machine and number of processes. Also, all computers were connected by 100 Mbits Ethernet with switches enabling parallel communications.

In general, each test program was run in three modes: source mode, protocol mode, and checkpoint mode. In source mode, the original program was executed. While in

the protocol mode, we applied the libELC protocol to the test program, however no checkpoint is taken. And in the checkpoint mode, not only was the libELC protocol applied, physical checkpoints were also created.

Except for the Monte-Carlo simulation and 1-D decomposition Matrix Multiplication experiments, in the other three programs (Gauss-Jordan method, Laplace Solver and Parallel NeuroSys), four checkpoints were triggered by the checkpoint function: *ELC_DoCKPT()*. Generally, we picked four positions in the program to insert the calls. The Monte-Carlo simulation program used the Time Interval mechanism to create the checkpoints. However, in the 1-D decomposition Matrix Multiplication experiments, no checkpoints are taken (See Section 5.7).

Also, we used a range of different data sets and numbers of processes for each test. Moreover, the figures shown in the following tests are collected from a number of runs, discarding the outliers.

5.2 Performance Model

In this section, we present a performance model of libELC. Although the model has included all possible factors that may affect the performance of libELC, it is necessary to claim that, for different programs, different parameters and modifications need to be applied to the performance model.

First, we give an expression that defines the relation between a program's execution time and the input data size and performance volume:

$$T_{exe} = \frac{f(DataSize)}{\sum Speed_i}$$

In which, T_{exe} is the program execution time, $DataSize$ is the total volume of the input data, $f(DataSize)$ is the computation volume, and $\sum Speed_i$ is the total process capacity. Note $f(DataSize)$ may be different functions for different programs. Also in a heterogeneous network, the data may be not evenly distributed. Instead, the data volume allocated for each process depends on the process performance. However, if the data volume is kept proportional to the process speed, the cost of each sub task is

equal to the overall program execution time, although this optimization may not be practicable in some cases.

Second, we calculate the overhead introduced by libELC. Generally, the overhead of libELC consists of two parts: Protocol Overhead (*PO*) and Checkpoint Overhead (*CO*). The protocol overhead (*PO*) is mainly caused during the program execution: a process needs to maintain its shadow stack, which inspects the program's execution flow; locates the variable address and records the heap allocation/release. Also it needs to log message envelopes upon the sending/receiving. The checkpoint overhead (*CO*) is the cost of process coordination to and checkpoint creation. Note that in the following experiments, the overhead of the protocol mode (with the libELC protocol but without triggering checkpoints) is *PO* and the difference between the results of the checkpoint mode and the protocol mode is *CO*. Generally, *PO* is proportional to the number of show stack operations, and it is expressed by:

$$PO = C_p * N_{op}$$

In which, C_p is the average cost for each show stack operation, and N_{OP} is total number of operations that a process performs during its execution. Moreover, N_{OP} is associated with three parameters: the number of messages N_M , the number of variables in the program N_{VAR} and the complexity of data structure used in the program f : $N_{OP} = N_M + f * N_{VAR}$ (Thus $PO = C_p * (N_M + f * N_{VAR})$). The reason we concern about the data structure complexity is that in most cases, a simple composite datatype costs the system more much time to analyze and locate its members, than managing the same number of common variables (See Section. 4.2.2.3). Our experience shows that this is a key performance factor of the uniprocess checkpoint.

As to *CO*, the overhead of creating checkpoint files depends mainly on the checkpoint data size and the I/O system performance, in which the checkpoint data size is usually a function of the program's input data size: $g(DataSize)$. Furthermore, the coordination overhead relies on the in-transit and orphan message number.

$$CO = N_{CKPT} * (\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG});$$

In which, N_{CKPT} is the number of checkpoints, and IO_Speed is the speed of the I/O system, C_m is the cost on identifying and logging in-transit and orphan message envelopes, and N_{MSG} is the total number of in-transit and orphan message.

Given the above expressions, we can define libELC's performance model as:

$$Overhead = \frac{CO + PO + T_{exe}}{T_{exe}} - 1$$

$$= \frac{PO + CO}{T_{exe}}$$

$$= \frac{N_{CKPT} * (\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG}) + C_p * (N_M + f * N_{VAR})}{\frac{f(DataSize)}{\sum Speed_i}}$$

Note the above expression denotes libELC's overhead in the checkpoint mode. As to the overhead of the protocol mode, it is simplified to:

$$Overhead = \frac{PO + T_{exe}}{T_{exe}} - 1$$

$$= \frac{PO}{T_{exe}}$$

$$= \frac{C_p * (N_M + f * N_{VAR})}{\frac{f(DataSize)}{\sum Speed_i}}$$

As mentioned above, this model is generic but very basic. In the following four tests, we will study it with more case-specific parameters.

5.3 Test 1: Gauss-Jordan method for solving systems of linear equations

The first experiment is an MPI implementation of the Gauss-Jordan method for solving systems of linear equations, which was written by J. Meyer at University of Nebraska at Omaha. The linear system is evenly distributed by row among N-1 processes, from which the results are collected to the rank 0 process by the MPI_Allreduce function call.

Four checkpoint calls are inserted in the program: the first checkpoint is taken after the rank 0 process completes the linear equation initialization; the second checkpoint is called when the master process distributes the equations to the other N-1 processes; the third call is made during the solving procedure (the halfway); the last one happens when the solving is finished.

We ran the program on LAM/MPI 7.0.4 with three different linear equation sizes: 4,000, 8,000 and 16,000. The program was tested with a four processor configuration (Table 4).

Machines used	Number of Processes
csserver.ucd.ie	1
csultra01.ucd.ie	1
csultra02.ucd.ie	1
csa007b4pc5.ucd.ie	1

Table 4. Process configuration in Gauss-Jordan experiments.

5.3.1 Size: 4,000

In the protocol mode, libELC introduces an overhead of about 12.25%, which includes the cost of logging message envelopes, recording program execution flow and inspecting program state. However, in the checkpoint mode, the overhead increases to 34.45%, which is mainly caused by the I/O operations.

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	431.8440	357.5612	415.3793
2	361.7256	406.4668	450.9052
3	376.1245	408.6260	513.4520
4	307.3733	386.1419	492.6780
5	269.9916	459.8315	365.2496
Average.	349.4118	392.2147	469.7841
Overhead		12.25%	34.45%

Table 5. Gauss-Jordan experiment results for datasize: 4,000.

5.3.2 Size: 8,000

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	1099.844	1368.3343	1229.488
2	1215.651	1174.8286	1175.125
3	1193.014	1246.8299	1438.118
4	1274.838	1152.7711	1395.019
5	1017.824	1288.9251	1215.101
Average.	1160.234	1246.338	1285.170
Overhead		7.42%	11.25%

Table 6. Gauss-Jordan experiment results for datasize: 8,000.

When the problem size grows up to 8,000, the overhead of the protocol drops to 7.42%. The slight improvement of the protocol-mode result is due to the increase of data size, which causes more operations on saving the program state. A significant decrease in the overhead of the checkpoint mode was observed (to 11.25%). This is because the I/O is no longer the main performance factor compared with the computation.

5.3.3 Size: 16,000

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	5462.419	5482.5624	5822.543
2	5456.314	5459.0973	5519.292
3	5469.852	5558.52	5517.689
4	5473.408	5436.624	5576.047
5	5477.924	5465.9402	5492.715
Average.	5467.984	5480.549	5585.654
Overhead		0.2%	2.15%

Table 7. Gauss-Jordan experiment results for datasize: 16,000.

In the case of a problem size of 16,000, we observed the best performance of libELC, whose overheads are only 0.2% in the protocol mode, and 2.15% in the checkpoint mode.

5.3.4 Analysis

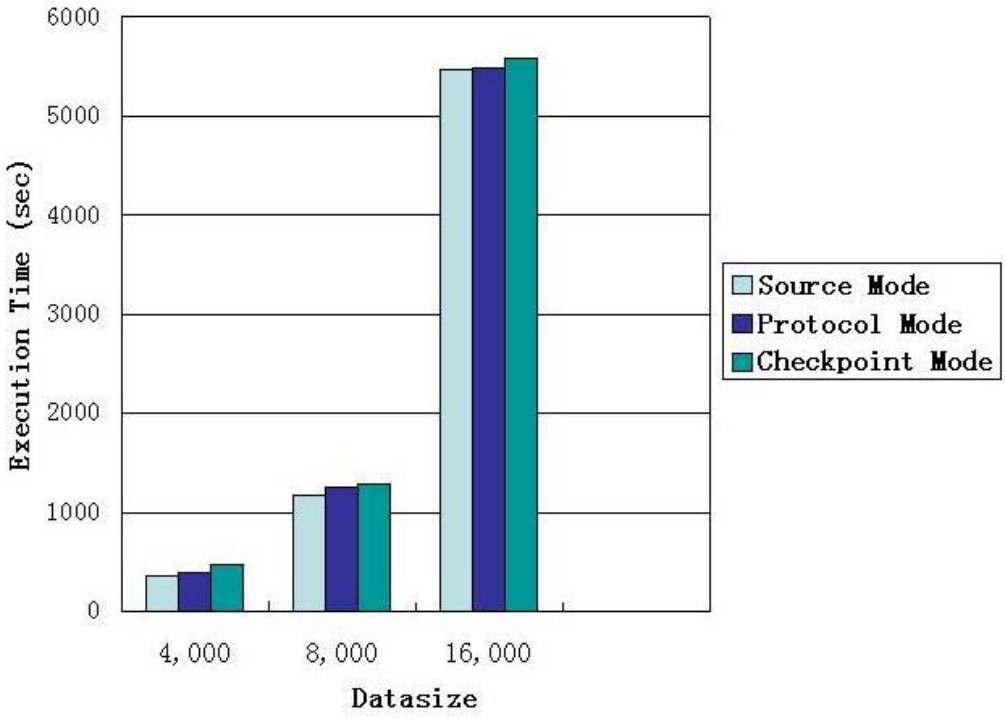

Figure 9. Experiments results of Gauss-Jordan method, in which the x-axis scale is the size of the linear system.

We ran the Gauss-Jordan Elimination program with three data sizes. The performance of libELC is shown in the following figure (See Figure 9). In this test, the program's computation volume is denoted by the expression $f(DataSize) = DataSize^2$ and the checkpoint data size is the same as the input data size: $g(DataSize) = DataSize$. Increasing the data volume did not change the data structure and the number of variables in the program, which determines *PO*. Also, more data incurs no extra coordination overhead. The impact of a larger data size is that it increases the program execution time T_{exe}, as well as the checkpoint file creation time $\frac{DataSize}{IO\_Speed}$. Applying to the performance model, if all other parameters are fixed except *DataSize*.

$$Overhead = \frac{N_{CKPT} * (\frac{DataSize}{IO\_Speed} + C_m * N_{MSG}) + C_p * (N_M + f * N_{VAR})}{\frac{DataSize^2}{\sum Speed_i}}$$

$$= \frac{N_{CKPT} * \sum Speed_i}{IO\_Speed * DataSize} + \frac{N_{CKPT} * C_m * N_{MSG} + C_p * (N_M + f * N_{VAR})}{\frac{DataSize^2}{\sum Speed_i}}$$

It is obvious from the model that an increase of *DataSize* will cause less overhead of libELC, which is exactly the desired result since fault tolerance is only of interest for large long-lived executions.

5.4 Test 2: 2-D block decomposition Laplace Solver

The Laplace Solver program we tested was first developed by Robb Newman, and converted to MPI by Xianneng Shen. This program uses a finite difference scheme to solve Laplace's equation for a square matrix distributed over a square process topology, in which each matrix element is updated based on the values of the four neighbouring matrix elements. This procedure is repeated until the average change in any matrix element is smaller than a specified value. Similar with the Gauss-Jordan method experiment, in this test the first checkpoint is taken when the matrix is initialized and the last one is made when the computation finishes. The other two checkpoints are trigger every 2,500 iterations (5,000 in total). We adopted two configurations for running this test: 4 and 16 processes with MPICH-1.2.6.

5.4.1 Number of Processes: 4; Matrix Size: 512*512

Machine used	Number of processes
csserver.ucd.ie	1
csultra01.ucd.ie	1
csultra02.ucd.ie	1
csa007b4pc5.ucd.ie	1

Table 8. 4 process configuration in Laplace Solver experiment.

In this scenario, we ran the test with 4 processes and a 512*512 matrix with 48 node edges (Table 9):

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	473.271	502.125	530.720
2	489.628	496.933	534.606
3	467.290	504.978	554.885
4	493.735	568.046	551.169
5	469.644	492.173	533.683
Average.	496.353	517.951	540.952
Overhead.		4.35%	8.98%

Table 9. Laplace Solver experiment results for 4 processes and matrix size 512*512.

5.4.2 Number of Processes: 16; Matrix Size: 512*512

Machines used	Number of Processes
csserver.ucd.ie	2
csultra01.ucd.ie	2
csultra02.ucd.ie	2
pg1cluster01.ucd.ie	2
pg1cluster02.ucd.ie	2
pg1cluster03.ucd.ie	2
pg1cluster04.ucd.ie	2
csa007b4pc5.ucd.ie	2

Table 10. 16 process configuration in Laplace Solver experiment.

In the first run, we used 16 processes to solve the same matrix as the 4 process configuration: size of 512*512 with 24 node edges:

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	603.259	600.584	604.555
2	600.544	602.439	618.515
3	594.183	640.861	640.113
4	579.079	615.563	610.396
5	584.251	608.003	680.321
Average.	592.263	613.490	630.780
Overhead.		3.58%	6.50%

Table 11. Laplace Solver experiment results for 16 processes and matrix size 512*512.

5.4.3 Number of Processes: 16; Matrix Size: 1024*1024

Then, we increase the matrix size to 1024*1024 (48 node edges) with the same 16 process configuration. The results are:

Runs	Source Mode (sec)	Protocol Mod (sec)	Checkpoint Mode (sec)
1	1654.477	1631.159	1560.790
2	1567.377	1695.699	1608.597
3	1687.606	1773.178	1762.854
4	1695.908	1592.223	1772.727
5	1764.140	1778.471	1860.225
Average.	1673.901	1694.146	1713.038
Overhead.		1.21%	2.43%

Table 12. Laplace Solver experiment results

for 16 processes and matrix size 1024*1024

From the above tables, we observe a significant increase of the execution time caused by the increase of number of processes and data size. However, the overhead of libELC drops steadily, 3.58%, 6.50% for the matrix size 512*512, and 1.21%, 2,43% for the 1024*1024 matrix.

5.4.3 Analysis

In this experiment we observe the best protocol-mode performacve of libELC: 3.05%. As discussed in Section 5.2 and 4.2.2.3, the complexity f of the data structure employed in an MPI program plays a significant role in determining libELC's protocol overhead PO: $PO = C_p * (N_M + f * N_{VAR})$. Among the five test programs, the data structure used in the Laplace Solver program is the simplest, which could be used to explain the outstanding performance in the protocol-mode runs (Figure 10).

Moreover, given the parameters that have the impact on CO $\frac{N_{CKPT} * (\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG})}{\frac{f(DataSize)}{\sum Speed_i}}$, it is noted that increasing the number of processes does not introduce extra checkpoint overheads (in the second run 16 processes with a 512*512 matrix) . It looks the results conflict with the performance model, however it is explainable.

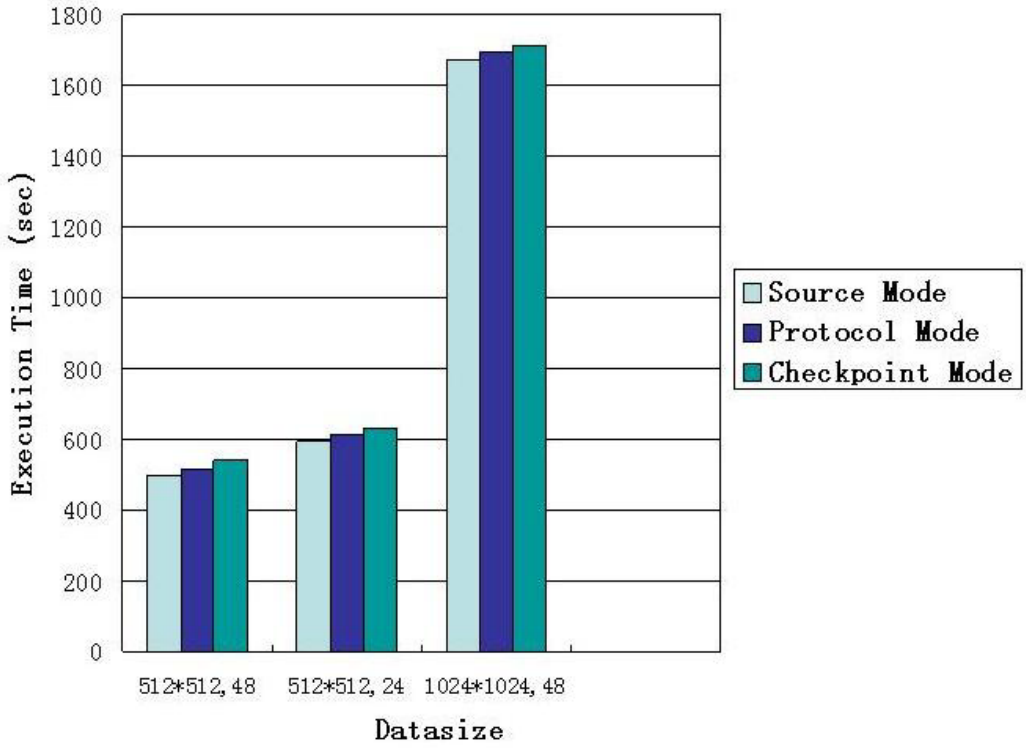

Figure 10. Experimental results from the Laplace Solver, in which tests are carried out with different sizes of linear equatations.

As mentioned above, this Laplace Solver program uses a square process topology to solve the equation for a square matrix. And for each matrix element, it is updated based on the values of the four neighbouring matrix elements. Thus, when employing more processes, the extra communication cost for exchanging data among distributed matrix blocks counteracts, and even overweighs the potential (compare the source-mode results between Table 10 and 11).

5.5 Test 3: Parallel NeuronSys - solve a system of ODE's modelling a network of neurons

Parallel Neuronsys is a neuron simulator program publically available at http://www.cs.usfca.edu/neurosys/. Generally, it is used to solve a system of Ordinary Differential Equation's (ODE) modelling a network of biologically realistic neurons on parallel computers. The current version uses a fourth order Runge-Kutta method to solve the equation. Neurons are evenly distributed over 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. We conducted experiments to model a network of 64 randomly interconnected neurons and a checkpoint is taken every 2,500 iterations. The equations modelling one neuron are based on a model presented in [75]. In our experiments, we ran the Parallel NeuroSys with MPICH-1.2.6.

5.5.1 4 Process Configuration

This test was launched with a four process configuration, as can be seen in Table 13.

Machine used	Number of Processes
csserver.ucd.ie	1
csultra01.ucd.ie	1
csultra02.ucd.ie	1
csultra03.ucd.ie	1

Table 13. 4 process configuration in Parallel NeuroSys experiment.

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	607.2415	771.7847	787.7825
2	602.9916	777.6131	796.7314
3	606.4636	770.2838	815.5689
4	604.6494	771.3599	791.9368
5	616.2382	768.0574	785.3921
Average	607.5169	771.8198	786.4258
Overhead		27.04%	29.44%

Table 14. Parallel NeuroSys experiment results for 4 processes configuration.

In the 4 process configuration, libELC protocol causes 27.04% overhead. This can be explained by the complex data structure the 64 randomly interconnected neuron network uses, which libELC needs to disassemble to locate the structure elements. And the checkpoint overhead is 29.44%.

5.5.2 8 Process Configuration

Machines used	Number of Processes
csserver.ucd.ie	1
csultra01.ucd.ie	1
csultra02.ucd.ie	1
pg1cluster01.ucd.ie	1
pg1cluster02.ucd.ie	1
pg1cluster03.ucd.ie	1
pg1cluster04.ucd.ie	1
csa007b4pc5.ucd.ie	1

Table 15. 8 process configuration in Parallel NeuroSys experiment.

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	212.2723	221.8225	231.2935
2	196.9984	212.1979	230.1724
3	192.6401	218.2799	232.4861
4	221.5995	222.0529	226.4064
5	219.1607	222.4031	225.9673
Average	208.5342	219.3512	229.2652
Overhead		5.18%	9.94%

Table 16. Parallel NeuroSys experiment results for 8 processes configuration.

We observed significant improved performance in the 8 process configuration: 5.18% overhead for libELC protocol and 9.94% overhead for checkpointing.

5.5.3 16 Process Configuration

Finally, we tested with 16 processes:

Machines used	Number of Processes
csserver.ucd.ie	2
csultra01.ucd.ie	2
csultra02.ucd.ie	2
csultra03.ucd.ie	2
pg1cluster01.ucd.ie	2
pg1cluster02.ucd.ie	2
pg1cluster03.ucd.ie	2
pg1cluster04.ucd.ie	2

Table 17. 16 process configuration in Parallel NeuroSys experiment.

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	73.0123	72.4537	77.7428
2	67.3325	65.7690	71.4936
3	70.6295	70.3439	73.6690
4	66.8718	69.9402	78.1830
5	71.3673	75.2425	73.9286
Average	69.8427	70.7498	75.0034
Overhead		1.30%	7.38%

Table 18. Parallel NeuroSys experiment results for 16 processes configuration.

Since the testing machines share the same file system in our test, the I/O operation became the main performance bottleneck when the number of processes was increased. However, as the previous experiment data showed, the overheads drop from 27.04% to 5.18%, to 1.30% for libELC protocol, and 29.44% to 9.94% to 7.38% for checkpointing, for 4, 8 and 16 process configuration respectively.

5.5.4 Analysis

In the tests of the Parallel NeuronSys environment (See Figure 11), we concentrated on the scalability of libELC. One immediate observation is that employing extra processes reduces the amount of time required to solve a particular problem, which should causes more checkpoint overhead. However, as the experiment results

demonstrate, the performance of libELC gets better with the increase of number of processes. To illustrate the results, we investigated the checkpoint files, in which we found that the reduced program execution significantly reduces the occurrence of in-transit and orphan messages (N_{MSG}). Fewer in-transit and orphan messages mean that upon checkpointing, libELC spends much less time on the message identification and logging. Recalling the performance model:

$$Overhead = \frac{N_{CKPT} * (\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG}) + C_p * (N_M + f * N_{VAR})}{\frac{f(DataSize)}{\sum Speed_i}}$$

Although the increase of computing capacity would augment the proportion of overhead, the message cost saving counteracts this effect and lowers the overall overhead. Moreover, in the experiments, we observe that if multiple processes share the same storage system, the I/O cost of creating the physical checkpoint file increases significantly as the system expands.

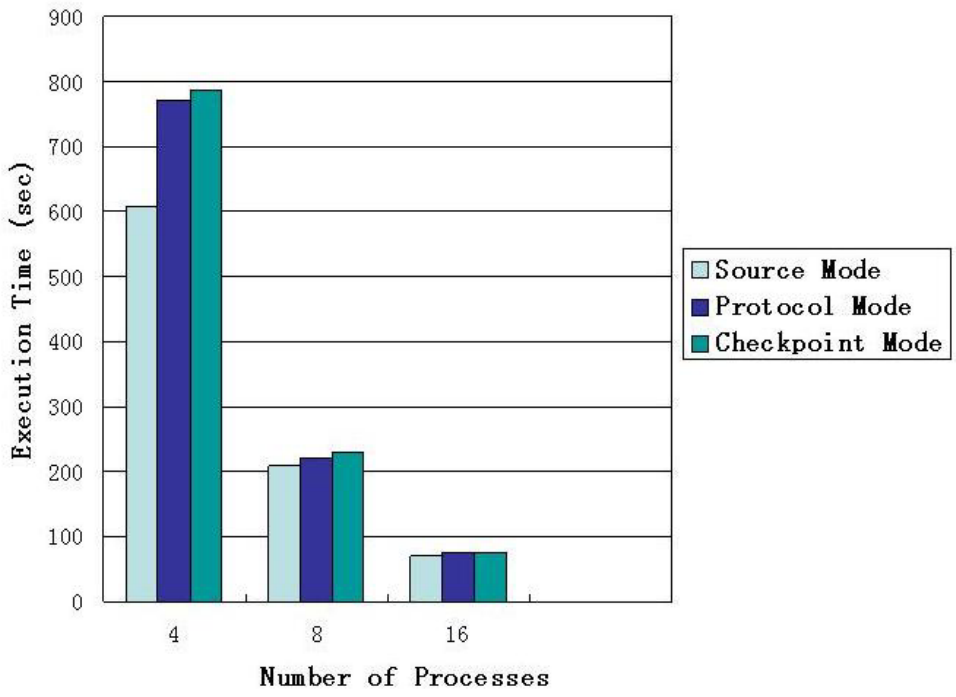

Figure 11. Experiment results of Parallel NeuronSys, in which tests are carried out with different numbers of processes.

5.6 Test 4: Monte-Carlo simulation of a system of hard disks

This 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 a 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. Due to limitations imposed by the program, all tests were performed with the same machine configuration with MPICH-1.2.6:

Machine used	Number of Processes
csserver.ucd.ie	1
csa007b4pc5.ucd.ie	1

Table 19. Process configuration in Monte-Carlo simulation.

To exercise the Time Interval method of triggering checkpoints, we deployed it in the Monte-Carlo simulation; the checkpoints were generated every 150 seconds. Also, we conducted three experiments by changing the simulation input parameters to vary the execution time.

5.6.1 Number of Disks: 16; Number of Sweeps: 10,000

In the first experiment, we ran a simulation of 16 disks, with 10,000 sweeps. The libELC protocol adds about 24.87% overhead, and checkpointing increases the overhead to 33.15%.

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	134.316	165.668	193.546
2	134.026	172.971	178.596
3	133.762	163.933	191.569
4	132.600	169.208	170.963
5	137.608	167.749	160.538
Average	134.462	167.905	179.042
Overhead		24.87%	33.15%

Table 20. Monte-Carlo simulation results for 16 disks and 10,000 sweeps.

5.6.2 Number of Disks: 32; Number of Sweeps: 10,000

In the following two tests, we increased the computation by increasing the simulation parameters. We observed a steady decrease of the overheads of both the protocol mode and the checkpoint mode. In the second test this dropped to 15.86 % (protocol mode) and 23.83% (checkpoint mode).

Runs	Source Mode (sec)	Protocol Mode (sec)	Checkpoint Mode (sec)
1	1007.543	1127.36	1111.139
2	1012.864	1121.27	1114.899
3	957.016	1112.925	1267.157
4	925.707	1115.312	1262.268
5	921.590	1113.149	1219.235
Average	964.944	1118.003	1194.939
Overhead		15.86%	23.83%

Table 21. Monte-Carlo simulation results for 32 disks and 10,000 sweeps.

5.6.3 Number of Disks: 32; Number of Sweeps: 20,000

The third experiment was run with the largest simulation size, 32 disks and 20,000 sweeps. However, the best performance obtained was 11.77% overhead for protocol mode and 14.32% for the checkpoint mode.

Runs	Source Mode	Protocol Mode	Checkpoint Mode
1	1808.593	2225.312	2503.654
2	2092.401	2303.537	2241.002
3	2284.970	2252.679	2225.032
4	1895.911	2239.588	2236.935
5	1989.446	2236.644	2307.413
Average	2014.262	2251.523	2302.837
Overhead		11.77%	14.32%

Table 22. Monte-Carlo simulation results for 32 disks and 20,000 sweeps.

5.6.4 Analysis

The Monte-Carlo Simulation test shows the performance of libELC with a time interval of 150 seconds. In these tests, three different volumes of simulation were performed, varying the program's execution time. In keeping with the experimental results reported earlier, as the data size increased, the overhead due to libELC reduced (See Figure 12).

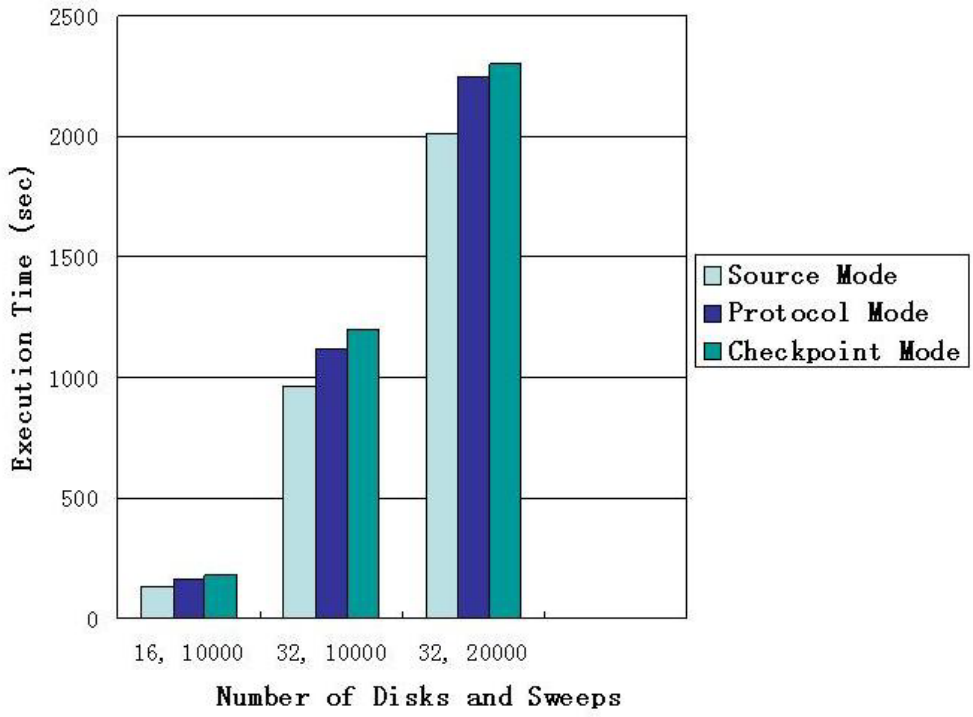

Figure 12. Experiments results of Monte-Carlo Simulation, in which the x-axis is the number of disks and sweeps in the simulation.

Recalling the performance model: given the other parameters fixed, the overhead of libELC will increase with the number of checkpoints. However, with the Time Interval Checkpoint mechanism, the checkpoint is triggered at regular intervals. So, the number of checkpoints created during the program execution is proportional to the program's execution time, in which case the parameter N_{CKPT} is denoted by: $N_{CKPT} = \frac{T_{exe}}{T_{interval}}$, in which $T_{interval}$ is the checkpoint interval set by the user. Then, the performance model becomes:

$$Overhead = \frac{\frac{f(DataSize)}{\sum Speed_i}}{T_{interval}} * (\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG}) + C_p * (N_M + f * N_{VAR})}{\frac{f(DataSize)}{\sum Speed_i}}$$

$$= \frac{(\frac{g(DataSize)}{IO\_Speed} + C_m * N_{MSG})}{T_{interval}} + \frac{C_p * (N_M + f * N_{VAR})}{\frac{f(DataSize)}{\sum Speed_i}}$$

As we can see, using the Time Interval Checkpoint mechanism, although the number of checkpoints is proportional of the program execution time, the overall overhead does not increase. Therefore, long running tasks benefit more from the checkpoint mode.

5.7 Test 5: Comparing Event Logging with Message Tagging

Given the existence of other coordinated checkpoint algorithms, it is necessary to compare Event Logging to its competitors. In this section we present the results of a comparision between Event Logging and Message Tagging.

Being coordination algorithms, Event Logging and Message Tagging aim to orchestrate multiple processes so as to create a valid recovery line. Neither of them deals with single process checkpoint and recovery. Thus in this test we concentrate on the communication overhead introduced by the two algorithms (no checkpoints taken).

As to the implementation, we chose the derived datatype approach (see Section. 3.2.2) for tagging the header information onto the message. It is noted that libELC (Event Logging) and libMTC (Message Tagging) are both constructed on top of the Chandy-Lamport algorithm. They share the codes for trigger local checkpoints, logging and replaying messages in our implementations. The differences rely in how a particular algorithm identifies the in-transit and orphan messages, which are summarized below:

Event Logging:

(1) *For each message sending/receiving operation, the process saves the message envelope in a structure of <tag, rank, communicatior>, and append the envelope onto the send/receive logs.*

(2) *Upon a process finishes its local checkpoint, it sends out the sending logs instead of marker messages to the corresponding processes.*

(3) *Upon receiving a send log, a process compares the send log with the local receive log to identify the in-transit message and orphan message envlopes (the process may trigger a new local checkpoint depending on whether it is the first send log it receives). Using the identified envelopes, the process can detect and save the trouble messages.*

Message Tagging:

(1) *For each message sending operation, the process copies the message content and header information into a buffer and defines a new derived datatype (orginal datatype plus header datatype). The temporary buffer and derived datatype are used instead of the original in the communication.*

(2) *For receiving a message, a process also needs to prepare the derived datatype and a temporary buffer. The tagged message is saved in the buffer, where the header is unpacked and checked to see wether the message is intransit or orphan.*

(3) *Note, the above two steps only help a process to identify the trouble messages. Coordination message is still necessary to notify the finish of a local checkpoint. In libMTC, a process p counts the numbers of the incoming/outgoing messages ($IN_{p,i}$, $OUT_{p,i}$) for the process i. $OUT_{p,i}$ will be sent to process i upon checkpointing as marker messages to let i know how many in-transit messages to be logged before closing the checkpoint ($OUT_{p,i} - IN_{i,p}$).*

It is observed from the above comparision tht the performance of Message Tagging is heavily affected by the message size; Event Logging, on the other hand, is not:

$$O_{MT} = \frac{C_{tag} * N_M * MessageSize}{\frac{f(DataSize)}{\sum Speed_i}}$$

$$O_{EL} = \frac{C_{\log} * N_M}{\frac{f(DataSize)}{\sum Speed_i}}$$

O_{MT}, O_{EL} denote the overheads of Message Tagging and Event Logging. C_{tag}, $C_{\log}$ respectively denote the cost for tagging messages tagging and logging message envelopes. N_M is the total number of messages and *MessageSize* is the average message size. $\frac{f(DataSize)}{\sum Speed_i}$ denotes the average execution time in source mode with outliers removed.

To compare the two methods, we ran a simple 1-D decomposition matrix multiplication program on the following machines with MPICH-1.2.6:

Machine used	Number of Processes
csserver.ucd.ie	2
csultra01.ucd.ie	1
csa007b4pc5.ucd.ie	1
csa007b3pc2.ucd.ie	1

Table 23. Process configuration in 1-D Decomposition matrix multiplication experiment.

Since we wanted to examine the communication overhead, we were concerned with the performance of the message passing between both homogeneous and heterogeneous processes. In particular we started the master process on a Linux 2.4 machine (csserver.ucd.ie), and launched the slaves on four different platforms: Linux 2.4 (csserver.ucd.ie), Solaris 5.8 (csultra01.ucd.ie), Linux 2.6 (csa007b4pc5.ucd.ie) and FreeBSD 5.2.1 (csa007b3pc2.ucd.ie).

In this program, the master process (rank 0) distributes the matrix to the slaves in a cyclic manner. These slave processes do the multiplication job and return the result to the master. The distribution unit is set to 4 columns so that the overall message number is $\frac{Column}{4}$ and the message size is $4 * Row$, in which *Column* and *Row* represent the number of columns and rows respectively. In this case, the performance model for message tagging becomes:

$$O_{MT} = \frac{C_{tag} * N_M * MessageSize}{\frac{f(DataSize)}{\sum Speed_i}}$$

$$= \frac{C_{tag} * \frac{Column}{4} * 4 * Row}{\frac{Column * Row}{\sum Speed_i}} = \frac{C_{tag} * Column * Row}{\frac{Column * Row}{\sum Speed_i}}$$

5.7.1 Matrix Size: 512*512, Message Size: 512 KB

The first run used a matrix of 512*512, in which the message size was 512 KB.

Runs	Source Mode (sec)	Event Logging (sec)	Message Tagging (sec)
1	32.423	32.756	33.784
2	32.838	32.212	34.349
3	32.564	31.853	34.347
4	32.582	32.439	33.260
5	32.341	32.439	35.059
Average	32.579	33.340	34.160
Overhead		2.33%	4.85%

Table 24. Matrix multplication experiment result for matrix size 512*512.

5.7.2 Matrix Size: 1024*1024, Message Size: 2 MB

We then increased the matrix size to 1024*1024, and the message size increased to 2 MB.

Runs	Source Mode (sec)	Event Logging (sec)	Message Tagging (sec)
1	123.243	125.380	131.566
2	124.056	126.373	127.262
3	118.744	125.438	130.939
4	118.289	122.868	132.458
5	122.494	124.113	129.518
Average	121.365	124.834	130.348
Overhead		2.85%	7.40%

Table 25. Matrix multplication experiment result for matrix size 1024*1024.

5.7.3 Matrix Size: 2048*2048, Message Size: 8 MB

Finally, two 2048*2048 matrix were multiplied. The message size was 8 MB.

Runs	Source Mode (sec)	Event Logging (sec)	Message Tagging (sec)
1	554.479	535.848	589.596
2	533.030	556.435	578.466
3	569.116	541.447	584.748
4	554.758	617.252	602.031
5	517.098	553.635	593.42
Average	545.696	560.923	589.652
Overhead		2.79%	8.05%

Table 26. Matrix multplication experiment result for matrix size 2048*2048.

5.7.4 Analysis

In this test, we ran a 1-D decomposition matrix multiplication program with three sizes: 512*512, 1024*1024, and 2048*2048 (See Figure 13).

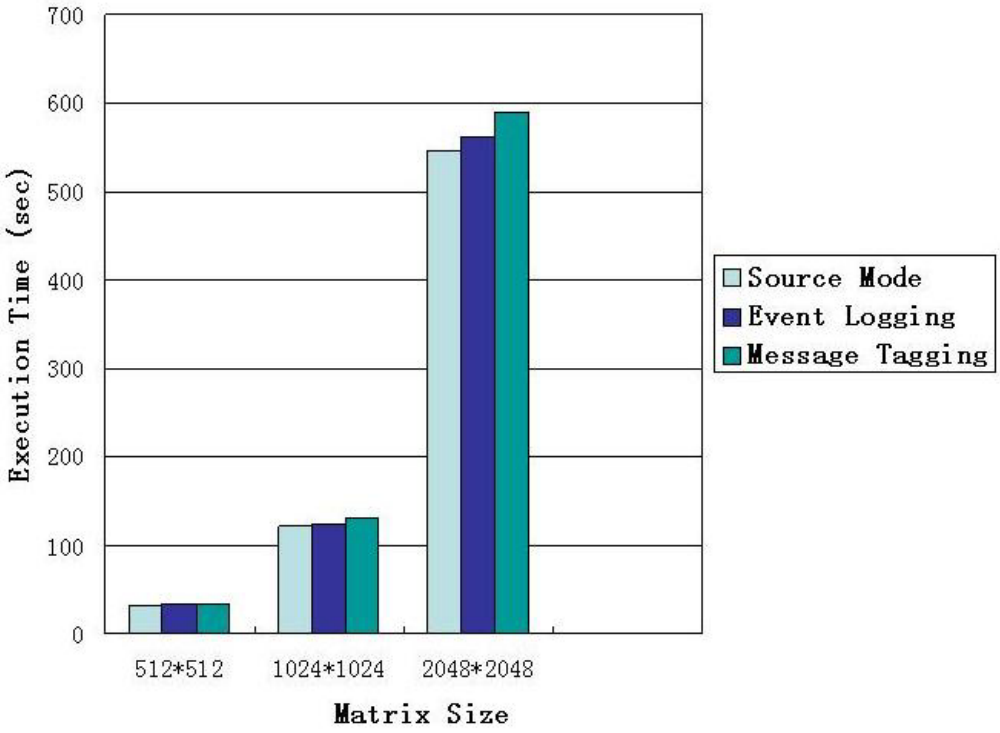

Figure 13. Comparison results of matrix multiplication, in which the message sizes in the three tests are 512KB, 2MB and 8MB.

In general, we observe more overheads introduced by Message Tagging than Event Logging. Moreover, in the derived datatype approach, processes need to build a new datatype and copy the header and the message into a temporary buffer for every outgoing message. Event Logging processes, on the other hand, only log the message envelopes. In this sense, Message Tagging is influenced by the message size, while Event Logging is not. This can be used to explain the above experiment results. When the message size increases with the matrix size, Message Tagging introduces more overhead but that of Event Logging remains approximately the same.

5.8 Optimal Checkpoint Interval

The reason for creating checkpoints for a long running program is to reduce the execution time lost due to failures. However in failure-free time, checkpointing prolongs the program's execution. We therefore need to find out the program's optimal checkpoint scheme, which minimizes the execution time, without weakening fault tolerance.

Take, for example, a program whose original execution time is 100 minutes. Creating a checkpoint for the program takes 4 minutes and the recovery costs 2 minutes. Moreover, assume a checkpoint is taken every 20 minutes throughout the program's life. Given the above conditions, the failure-free execution time of the program with checkpoints is:

$$100 + 4 * \frac{100}{20} = 120 \text{ (minutes);}$$

Moreover, suppose the program's MTBF (*Mean Time Between Failure*) is 50 minutes, so during its execution the possible number of failures is 100/50=2. Taking into account the recovery time, the expectation of the program execution time is $120 + 2 * 2 + \sum_{i=1}^{2} Lost_i$ ($0 < Lost_i \le 20$ minutes), in which $Lost_i$ is the execution time lost due to the i^{th} failure. The expectation of $\sum_{i=1}^{2} Lost_i$ is $2 * \frac{(1 + 2 + ...20)}{20} = 21$ (minutes). Then the expectation of the program execution time with checkpointing (*EPET*) is 145 minutes.

In this section, we present an approach trying to minimize *EPET*. From the above example, we can construct a generic expression to denote a program's *EPET*. Suppose the program's original execution time is T_O, the checkpoint cost is T_C, the recovery cost is T_R and the checkpoint interval is Δt. Also, suppose the program's average uptime is T_{UP} and the execution time lost due to the i^{th} failure's execution time lost is $Lost_i$. Then *EPET* is calculated by

$$EPET = T_O + \frac{T_O}{\Delta t} * T_C + \frac{T_O}{T_{UP}} * T_R + \sum Lost_i ;$$

The expectation of $\sum Lost_i$ is $\frac{T_O}{T_{UP}} * \frac{1 + \Delta t}{2}$. EPET then becomes

$$EPET = T_O + \frac{T_O}{T_{UP}} * T_R + \frac{T_O}{\Delta t} * T_C + \frac{T_O}{T_{UP}} * \frac{1 + \Delta t}{2} ;$$

Moreover, for a specific program the parameters T_O, T_C, T_R, T_{UP} are usually constants. So, when $\frac{T_O}{\Delta t} * T_C + \frac{T_O}{T_{UP}} * \frac{1+\Delta t}{2}$ is minimized, in which $\Delta t = \sqrt{2 * T_C * T_{UP}}$, *EPET* has the minimal value $T_O + \frac{T_O}{T_{UP}} * T_R + \frac{T_O}{2 * T_{UP}} + \sqrt{\frac{2 * {T_O}^2 * T_C}{T_{UP}}}$.

So, we conclude for a program with the Time Interval mechanism, the optimal checkpoint interval is $\sqrt{2 * T_C * T_{UP}}$.

5.9 Conclusion

In this chapter, we described several experiments conducted with libELC. The five different MPI programs were the Gauss-Jordan method for solving systems of linear equations; 2-D block decomposition Laplace Solver; Parallel NeuronSys and a Monte-Carlo simulation. In general, each program was run with either different input data sizes or different process scale. Moreover, the test of every data size and configuration was conducted with 3 modes: source, protocol and checkpoint modes. The protocol mode incorporates only the libELC protocol without triggering any checkpoints; and checkpoint mode creates four physical checkpoint files using libELC.

The experimental results demonstrate the portability of libELC in heterogeneous networks. Using a shared filesystem, machines with different architectures and data representation were able to generate uniform checkpoint files, which could be used to recover the MPI program's running state on other heterogeneous, non-compatible platforms. It is noted that we have performed severl recovery tests internally. However such testing results have not been included in this chapter. The formal experiments work is in progress and would be finished soon.

As to the performance, we observed that libELC's overhead is influenced by the data structure complexity, the data size and the process scale. In the test of the 2-D block decomposition Laplace Solver, we note that better performance was gained with simpler data structures in the MPI program use. As the data structure gets complex, it costs libELC more to locate the structure elements. Also, the more pointers the

program uses, the more time libELC spends on transforming the pointers' physical address to the logic representation.

Another performance factor is the data size. In the same program, for example, the Gauss-Jordan method for solving systems of linear equations), libELC introduces less overhead with large data sets than small ones. This becomes very significant when the machine is fully loaded, in which case the overhead of libELC becomes insignificant.

The third factor is the number of processes. Due to the significantly reduced number of in-transit and orphan messages, better performance results are obtained as the process scale increases. However, in our experiments, we also observe that if multiple processes share a single storage system (NFS), the concurrent I/O operations caused by creating the physical checkpoint files became a bottleneck for libELC.

Furthuremore, we tested libELC with the Time Interval mechanism. We observed that no significant performance descrease occured as the program's execution time increased. The advantage of the Time Interval mechanism is that it is transparent to the user. By comparision, the checkpoint function requires users to manage the checkpointing explicitly. However, the advantage of using the checkpoint function is that the user may manually control and select the most approapriate time for checkpointing.

We also presented the comparison between Event Logging and Message Tagging in this chapter. The experiments demonstrated that the performance of Message Tagging fluctuates with the message size. And more important, as discussed in Section. 3.2.2, the Message Tagging approach is not completely compliant with the MPI standard. By comparion, Event Logging introduces fewer overheads, and the performance of Event Logging is not affected by message size. And Event Logging is designed totally on top of MPI. From the point of view of implementation, another advantage of Event Logging over Message Tagging is that Event Logging enables fast recovery line commit. At the time message identification finishes, Event Logging has found out the envelopes for the in-transit messages, so a process can simply post receive requests to log these in-transit messages (note that FIFO communication is commonly 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. This is because the process has no knowledge of an in-transit message until it checks a message's header.

Chapter 6
Conclusion and Future Work

6.1 Summary

This thesis presents Event Logging, a high level coordination algorithm for the checkpoint/recovery of MPI programs in heterogeneous networks of computers. The main contribution of Event Logging is that it addresses the application-level non-FIFO challenge of the Chandy-Lamport algorithm, which is a key problem for a portable implementation of coordinated checkpointing in MPI programs. As a consequence, Event Logging is highly portable given a heterogeneous environment in

that it is totally built on top of the MPI standard. This makes Event Logging rely on no particular assumption of the underlying MPI implementation or the running platform. The portability significantly benefits Event Logging given the inherent heterogeneity of a network of computers.

From the point of the technique, Event Logging combines the merits of both coordinated checkpointing and message logging. However, it reduces most of the overheads of message logging since Event Logging records only the message envelope without the actual content. The message envelopes will be exchanged upon checkpointing for the message identification. The in-transit messages and orphan message envelopes will be identified and saved as part of the recovery line. The combination of coordinated checkpoint and message logging brings significant benefits including small failure-free overheads, fast recovery line commit, a simple recovery procedure and one-checkpoint rollback extent.

This thesis also presents the implementation of Event Logging: libELC. libELC is a portable checkpoint and recovery library for C/MPI programs. It employs Event Logging for distributed process coordination. As to the checkpoint/recovery for each individual heterogeneous process, libELC captures the snapshots at the application level. Unlike the traditional system-level or library-level checkpoint mechanisms, the application-level checkpoint examines the logic composition of a running process. This mainly consists of program variables, heap memory and execution flow. Thus, the state saving is done by recording the variable value, the heap content and program function calls. Upon recovery, the same sequence of functions is issued so the execution flow will be exactly reconstructed. Repetition of instructions executed before checkpoints is avoided by using *GOTO* statement. The value of a program variable is re-assigned following the variable's definition. The heap is reallocated and the saved content is restored in the new memory space. The advantage of such an application-level checkpoint/recovery approach is that it is totally system-independent. As long as a platform supports for ANSI C language, libELC is able to capture the snapshot for any C programs running on it.

Besides the uniprocess checkpoint/recovery facility, libELC integrates Event Logging by providing a set of wrapper functions for the MPI interface. These wrapper

functions hide the event logging operations completely from the point of view of the user. The user is given two options to trigger a global checkpoint in libELC, either by an explicit function call: *ELC_DoCKPT()* or by setting the Checkpoint Time Interval, which generates checkpoints in a preset period.

The experiments carried out to evaluate Event Logging are also presented in this thesis. Five programs were tested with two main MPI distributions: LAM/MPI-7.0.4 and MPICH-1.2.6. The experiment results demonstrate the efficiency of the Event Logging algorithm.

6.2 Future Work

Message Identification Optimization: The current version of libELC implements message identification by using the "plain" sequential search. However, as discussed in Section. 3.4.2, a fast search algorithm, like binary search, helps reduce the identification cost. Unfortunately, using such algorithm requires change to the data structure used to organize the event logs. Currently, libELC uses the link table to store the message envelope. In order to implement the fast search, changes will be necessary to use a sorted table to organize the envelopes.

Selective Checkpoint: One criticism of the coordinated checkpoint model is the lack of independence for each individual process to create local checkpoints. It mandates that all processes must participate in the checkpoint as well as the rollback. Given a small or medium network, the tradeoff is worthwhile. However, with an increase of the process number, as within a Computational Grid, such a penalty would be considerable. This is especially true of the involvement of all processes in recovery even if only one fails. To alleviate this constraint, the next step is to implement Selective Checkpoint [35, 36] to minimize the scale of checkpointing processes in libELC. Event Logging provides the ability to implement Selective Checkpoint since the inter-process message passing has already been logged upon checkpointing.

Minimize Checkpoint Datasize: Significant potential for optimization lies in minimizing the checkpoint datasize. Since the application-level approach takes the point of view of the program semantic, it is possible to use some compiler techniques [71, 72] to exclude the unnecessary data upon checkpointing. Given the dramatically

increasing problem size in the real world, such a benefit is scheduled in the future plan of libELC.

Local/Shared File System Support: Although the current libELC implementation allows the users to save the checkpoint files on the local disks, a shared file system is desired in most cases. The advantage of the shared file system is that it tolerates the faults such as disk corruption or machine physical damage, which make the local file system inaccessible upon recovery. However, the main drawback for using the shared file system is that the file system itself becomes a single point of failure. To avoid this problem, we consider adding supports to combine the merits of the local file system and the shared storage in the next version of libELC.

Bibography

[1] **A. Lastovetsky. "*Parallel Computing on Heterogeneous Networks*". J***ohn Wiley & Sons*, **2003.**

[2] **Alexey L. Lastovetsky, Ravi Reddy, "HMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers",** *In Proceedings of IPDPS*, **2003.**

[3] **A. Y. Zomaya and H. B. Diab, "Dependable Computing Systems: Paradigms, Performance Issues, and Applications",** *Wiley Series on Parallel and Distributed Computing, Wiley-Interscience*, **2005.**

[4] **W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high performance, portable implementation of the MPI message passing interface standard",** *Parallel Computing,* **Volume 22(6), pp.789-828, 1996.**

[5] **G. Burns, R. Daoud and J. Vaigl, "LAM: An open cluster environment for MPI",** *Proceeding of Supercomputing Symposium*, **pp.379--386, 1994, Toronto, Canada.**

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

[7] **R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems",** *ACM Transactions of Computer System*, **Volume 1(3), pp.222-238, ACM Press, 1983.**

[8] **L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem",** *ACM Transactions of Programming Language System*, **Volume 4(3), pp.382-401, ACM Press, 1982.**

[9] **R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall and M. W. Sukalski, "Architecture of LA-MPI, A Network-Fault-Tolerant MPI.",** *IPDPS*, **2004.**

[10] **B. Johnson, "An Introduction to the Design and Analysis of Fault-Tolerant Systems", In Pradhan, D.K (ed.),** *Fault-Tolerant Computer System Design,* **pp.1-87. Upper Saddle River, NJ: Prentice Hall, 1995.**

[11] **R. Guerraoui and A. Schiper, "Software-Based Replication for Fault Tolerance",** *Journal of Computer*, **Volume 30(4), pp.68-74, IEEE Computer Society Press, 1997.**

[12] **R. Batchu, A. Skjellum, Z. Cui, M. Beddhu, J. P. Neelamegam,Y. Dandass and M. Apte, "MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing",** *1st International Symposium on Cluster Computing and the Grid*, **May, 2001.**

[13] **G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, J. Dongarra, "Process Fault-Tolerance: Semantics, Design and Applications for High Performance Computing",** *International Journal for High Performance Applications and Supercomputing*, **April, 2004.**

[14] **G. Fagg, J. Dongarra, "Building and using a Fault Tolerant MPI implementation",** ***International Journal of High Performance Applications and Supercomputing*****, 2004.**

[15] **G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI",** *Proceedings of the International Parallel Processing Symposium*, **pp.526-531, Apr. 1996.**

[16] **K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems",** *ACM Transactions on Computing*

Systems, **Volume 3(1), pp.63-75, Aug. 1985.**

[17] Y. Chen, K. Li and J. S. Plank, "CLIP: A Checkpointing Tool for Message-passing Parallel Programs", *Proceedings of SC97: High Performance Networking Computing*, **Nov. 1997.**

[18] S. Louca, N. Neophytou, A. Lachanas and P. Evripidou, "Portable Fault Tolerance Scheme for MPI", *Parallel Processing Letters*, **Volume 10(4), pp.371-382, 2000.**

[19] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier and F. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging", *In proceedings of The IEEE/ACM SC2003 Conference*, **Phoenix USA, Nov. 2003.**

[20] N. Woo, S. Choi, H. Jung, J. Moon, H. Y. Yeom, T. Park and H. Park, "MPICH-GF: Providing Fault Tolerance on Grid Environments", ***The*** *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), the poster and research demo session*, **May. 2003.**

[21] N. T. Karonis, B. Toonen and Ian Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface", *Journal of Parallel and Distributed Computing*, **Volume 63(5), pp.551-563, May. 2003.**

[22] G. Bronevetsky, D. Marques, K. Pingali and P. Stodghill, "C^3: A System for Automating Application-level Checkpointing of MPI Programs", *International Workshop on Languages and Compilers for Parallel Computing*, **Oct. 2003.**

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, **Volume 21(7), pp.588-565, Jul. 1978.**

[24] Y. Tamir and C. H. Séquin, "Error recovery in multicomputers using global checkpoints", *In Proceedings of the International Conference on Parallel Processing*, **pp. 32-41, Aug. 1984.**

[25] L. Valiant, "A bridging model for parallel computation", *Communications of the ACM*, **Volume 33(8), pp.103-111, 1990.**

[26] Z. Tong, R. Y. Kain and W. T. Tsai, "Rollback-recovery in distributed systems using loosely synchronized clocks", *In IEEE Transactions on Parallel and Distributed Systems*, **Volume 3(2), pp.246-251, Mar. 1992.**

[27] F. Cristian and F. Jahanian, "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computation", *10th Symposium on Reliable Distributed Systems*, **Sep. 1991.**

[28] P. Ramanathan and K. G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System", *IEEE*

Transactions on Software Engineering, **Volume 19(6), pp.571-583, Jun. 1993.**

[29] N. Neves and W. K. Fuchs, "Using Time to Improve the Performance of Coordinated Checkpointing", *IEEE International Computer Performance and Dependability Symposium*, **Sep. 1996.**

[30] N. Neves and W. K. Fuchs, "Coordinated Checkpointing without Direct Coordination", *Proceedings of IEEE International Computer Performance and Dependability Symposium*, **pp.23-31, Sep. 1998.**

[31] G. P. Kavanaugh and W. H. Sanders, "Performance Analysis of Two Time-based Coordinated Checkpointing Protocols", *Pacific Rim International Symposium on Fault-Tolerant Systems*, **Dec. 1997.**

[32] R. Koo, S. Toueg, "Checkpointing and rollback recovery for distributed systems", *IEEE Transaction of Software Engeneering Special Edition,* **Volume 13, pp. 23-31, 1987.**

[33] K. Li, J. F. Naughton and S. Plank, "Checkpointing multicomputer applications", *In Proceedings of IEEE Conference On Reliable Distributed System,* **pp.2-11, 1991.**

[34] S. Venkatesan, "Message optimal incremental snapshots", *In Proceedings of IEEE 9th International Conference of Distributed Computer System*, **pp 53-60, 1991.**

[35] P. J. Leu and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems", *In Proceedings of International Conference on Data Engineering*, **pp.154-163, 1988.**

[36] J. L. Kim and T. Park, "An efficient protocol for checkpointing recovery in distributed systems", *IEEE Transactions of Parallel Distributed System*, **Volume 4, pp.955-960, 1993.**

[37] B. Randell, "System structure for software fault-tolerance", *IEEE Transactions on Software Engineering Special Edition*, **Volume 1(2), pp.220-232, Jun. 1975.**

[38] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic and Causal", *Proceedings of the 15th International Conference on Distributed Computing Systems*, **pp.229-236, 1995.**

[39] R. Strom and S. Yemini, "Optimistic recovery in distributed systems." ***ACM Transactions on Computer Systems,*** **Volume 3(3), pp. 204-226, Aug. 1985.**

[40] R.E. Strom, D.F. Bacon, and S.A. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems," *Proceedings of 18th International Symposium of Fault-Tolerant Computing*, **pp. 44-49, 1988.**

[41] **B. Bieker, G. Deoninck, E. Maehle and J. Vounckx, "Reconfiguration and checkpointing in massively parallel systems",** *In Proceedings of the 1st European Dependant Computing Conference*, **EDCC-1, pp. 353-370, Oct. 1994.**

[42] **T. T-Y. Juang and S. Venkatesan, "Crash recovery with little overhead",** *In Proceedings of the International Conference on Distributed Computing Systems*, **pp. 454-461, May 1991.**

[43] **K. Bhatia, K. Marzullo and L. Alvisi, "The relative overhead of piggybacking in causal message logging protocols",** *In Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, **pp. 348-353, 1998.**

[44] **B. Bhargava and S. R. Lian. "Independent checkpointing and concurrent Rllback for recovery - An optimistic approach",** *In Proceedings of the Symposium on Reliable Distributed Systems*, **pp. 3-12, 1988.**

[45] **A. Acharya and B. R. Badrinath, "Recording distributed snapshots based on causal order of message delivery",** *In Information Processing Letters*, **Volume 44(6), Dec. 1992.**

[46] **E. N. Elnozahy, "Manetho: Fault tolerance in distributed systems using rollback recovery and process replication",** *Ph.D. Thesis, Rice University*, **Oct. 1993. Also available as** *Technical Report 93-212, Department of Computer Science, Rice Universit***y.**

[47] **L. Alvisi, "Understanding the message logging paradigm for masking process crashes",** *Ph.D. Thesis, Department of Computer Science, Cornell University*, **Jan. 1996. Also available as** *Technical Report TR-96-1577***.**

[48] **D. B. Johnson, "Distributed system fault tolerance using message logging and checkpointing",** *Ph.D. Thesis, Rice University, Departmen of Computer Science*, **1989.**

[49] **G. Muller, M. Hue and N. Peyrouz, "Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment",** *In Lecture Notes in Compute Science: Dependable Computing,* **EDDC-1, pp.491-508, 1994.**

[50] **J. S. Plank, "Efficient checkpointing on MIMD architectures",** *Ph.D. Thesis, Princeton University, Department of Computer Science*, **1993.**

[51] **L. M. Silva, "Checkpointing mechanisms for scientific parallel applications",** *Ph.D. Thesis, University of Coimbra, Department of Computer Science*, **1997.**

[52] **D. Briatico, A. Ciuffoletti and L. Simoncini, "A distributed domino-effect free recovery algorithm",** *In Proceedings of the IEEE International Symposium on Reliability, Distributed Software, and Databases***,**

pp. 207-215, Dec. 1984.

[53] **D. L. Russell, "State restoration in systems of communicating processes",** *In IEEE Transactions on Software Engineering Special Edition*, **Volume 6(2), 183-194, Mar. 1980.**

[54] **L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain and A. D. Mel, "An analysis of communication-induced checkpointing",** *In Digest of Papers, FTCS-29, The Twenty Nineth Annual International Symposium on Fault Ttolerant Computing*, **pp.242-249, Madison, Wisconsin, 1999.**

[55] **Elnozahy and W. Zwaenepoel, "On the use and implementing of message logging",** *In Digest of Papers, FTCS-24, The Twenty Fourth International Symposium on Fault-Tolerant Computing*, **pp.298-307, 1994.**

[56] **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.**

[57] **J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walker, "MPI: The Complete Reference***", The MIT Press*, **1996.**

[58] **L. Silva, J. Silva, "Global checkpointing for distributed programs",** *In Proceedings of. IEEE 11th Symposium On Reliable Distributed System*, **pp.155-162, 1992.**

[59] **E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing",** *In Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, **pp. 39-47, Oct. 1992.**

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

[61] **M. J. Litzkow and M. Solomon, "Supporting Checkpointing and Process Migration Outside the UNIX Kernel",** *USENIX Conference Proceedings*, **pp.283-290, Jan. 1992, San Francisco, CA.**

[62] **J. S. Plank, M. Beck, G. Kingsley and K. Li, "Libckpt: Transparent Checkpointing under Unix",** *USENIX Winter 1995 Technical Conference*, **pp.213-224, Jan. 1995.**

[63] **H. Zhong and J. Nieh, "CRAK: Linux Checkpoint / Restart As a Kernel Module",** *Technical Report CUCS-014-01, Department of Computer Science, Columbia University*, **Nov. 2002.**

[64] **E. Pinheiro, EPCKPT,** *http://www.research.rutgers.edu/~edpin/epckpt/*

[65] **O. O. Sudakov, E. S. Meshcheryakov, CHPOX,**

http://www.cluster.kiev.ua/tasks/chpx_eng.html

[66] **M. Litzkow, T. Tannenbaum, J. Basney and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed System",** *University of Wisconsin-Madison Computer Sciences Technical Report #1346*, **April 1997.**

[67] **D. Gibson, Esky,** *http://cap.anu.edu.au/cap/projects/esky/index.html*

[68] **B. Ramkumar and V. Strumpen, "Portable Checkpointing for Heterogeneous Architectures",** *In 27th International Symposium on Fault-Tolerant Computing - Digest of Papers*, **Seattle, WA, pages 58-67, June 1997.**

[69] **A. Ferrari, "Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems",** *Technical Report: CS-96-15, Department of Computer Science, University of Virginia*, **1996.**

[70] **V. C. Zandy, B. P. Miller and M. Livny, "Process Hijacking",** *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, **pp.32, IEEE Computer Society, 1999.**

[71] **J. S. Plank, M. Beck, and G. Kingsley, "Compiler-assisted memory exclusion for fast checkpointing",** *IEEE Technical Committee on Operating Systems and Application Environments*, **Volume 7(4), pp.10-14, Winter 1995.**

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

[73] **K. Venkatesh, T. Radhakrishnan, H. F. Li, "Optimal checkpointing and local recording for domino free rollback recovery",** *Inf. Process. Lett.* **Volume 25, pp.295-303, 1987.**

[74] **E. N. Elnozahy, L. Alvisi, Y. M. Wang and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems",** *ACM Computer Survey*, **Volume 34(3), pp.375-408, ACM Press, 2002.**

[75] **J White, et al,** *Journal of Computational Neuroscience*, **Volume 5, pp.5- 16, 1998.**

[76] **N. Woo, H. Jung, D. Shin, H. Han, H. Y. Yeom, "Performance Evaluation of Consistent Recovery Protocols using MPICH GF",** *to appear*.

[77] **Y. M. Wang, E. Chung, Y. Huang and E. N. Elnozahy, "Integrating checkpointing with transaction processing",** In Digest of Papers, FTCS-27, The Twenty Seventh Annual International Symposium on Fault-Tolerant Computing, **pp.304-308, 1997**

Appendix A.

Example of libELC

```
/********************** FILE: mm.c ****************************************
* DESCRIPTION:
*   In this template code, the master task distributes a matrix multiply
*   operation to numtasks-1 worker tasks.
*   NOTE1:  C and Fortran versions of this code differ because of the way
*     arrays are stored/passed.  C arrays are row-major order but Fortran
*     arrays are column-major order.
* AUTHOR for MPL version: Ros Leibensperger / Blaise Barney
* LAST MPL REVISED: 09/14/93 for latest API changes.  Blaise Barney
* CONVERTED TO MPI: 11/12/94 by Xianneng Shen
*****************************************************************************/

#include <stdio.h>
#include <mpi.h>
```

```
#include <libELC.h>

#define SIZE    1024
#define NRA SIZE                        /* number of rows in matrix A */
#define NCA SIZE                        /* number of columns in matrix A */
#define NCB SIZE                        /* number of columns in matrix B */
#define MASTER 0                        /* taskid of first task */
#define FROM_MASTER 1                   /* setting a message type */
#define FROM_WORKER 2                   /* setting a message type */

main(int argc, char **argv)
{
/**************** Program Variable Definition ********************/
        int numtasks,                   /* number of tasks in partition */
            taskid,                     /* a task identifier */
            numworkers,                 /* number of worker tasks */
            source,                     /* task id of message source */
            dest,                       /* task id of message destination */
            nbytes,                     /* number of bytes in message */
            mtype,                      /* message type */
            intsize,                    /* size of an integer in bytes */
            dbsize,                     /* size of a double float in bytes */
            rows,                      /* rows of matrix A sent to each worker */
            averow,
            extra,
           offset,                      /* determine rows sent to each worker */
            i, j, k,                    /* misc */
            count;

        double *a,                      /* matrix A to be multiplied */
                *b,                     /* matrix B to be multiplied */
                 *c;                    /* result matrix C */
        MPI_Status status;

/**************  Records Execution Flow in the Flow Table *************/
        OnCallEnter(0,0);       // this is the main() function;

/***************** Records Variables in Shadow Stack **************/
        OnVarDef(&numtasks,sizeof(int));
        OnVarDef(&taskid,sizeof(int));
        OnVarDef(&numworkers,sizeof(int));
        OnVarDef(&source,sizeof(int));
        OnVarDef(&dest,sizeof(int));
        OnVarDef(&nbytes,sizeof(int));
        OnVarDef(&mbyte,sizeof(int));
        OnVarDef(&insize,sizeof(int));
        OnVarDef(&dbsize,sizeof(int));
        OnVarDef(&rows,sizeof(int));
        OnVarDef(&averow,sizeof(int));
        OnVarDef(&extra,sizeof(int));
        OnVarDef(&offset,sizeof(int));
        OnVarDef(&i,sizeof(int));
        OnVarDef(&j,sizeof(int));
        OnVarDef(&k,sizeof(int));
        OnVarDef(&count,sizeof(int));

        OnPtrDef(&a,1);
```

```
        OnPtrDef(&b,1);
        OnPtrDef(&c,1);

        OnVarDef(&status,sizeof(MPI_Status));

/************** Execution Jump in the Recovery*************/
        if (STATE_FLAG==ELC_RECOVERY)
        {
          ELC_MPI_Init(&argc, &argv);        // re-init MPI runtime environment
         switch (g_flowTail->LABEL)
         {
              case (-1):          goto K1;
              case (-2):          goto K2;
              case (-3):          goto K3;
              case (-4):          goto K4;
              case (-5):          goto K5;
              case (-6):          goto K6;
              case (-7):          goto K7;
              case (-8):          goto K8;
              case (-9):          goto K9;
              case (-10):         goto K10;
              case (-11):         goto K11;
              case (-12):         goto K12;
              case (-13):         goto K13;
              case (-14):         goto K14;
              case (-15):         goto K15;
         }
        }

/****************** Execution Statement ***************************/
        intsize = sizeof(int);
        dbsize = sizeof(double);

        ELC_MPI_Init(&argc, &argv); // wrapper MPI_Init()

        // Record the Heap Allocation
        a=ELC_malloc(NRA*NCA*sizeof(double));
        b=ELC_malloc(NCA*NCB*sizeof(double));
        c=ELC_malloc(NRA*NCB*sizeof(double));

        MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        numworkers = numtasks-1;

/****************** Master Task  ************************************/
        if (taskid == MASTER) {
         for (i=0; i<NRA; i++)
           for (j=0; j<NCA; j++)
            *(a+i*SIZE+j)= i+j;
         for (i=0; i<NCA; i++)
           for (j=0; j<NCB; j++)
             *(b+i*SIZE+j)= i*j;

         /* send matrix data to the worker tasks */
         averow = NRA/numworkers;
         extra = NRA%numworkers;
         offset = 0;
```

```
          mtype = FROM_MASTER;
          for (dest=1; dest<=numworkers; dest++) {
            rows = (dest <= extra) ? averow+1 : averow;
K1:         ELC_MPI_Send(&offset, 1, MPI_INT, dest,
                         mtype, MPI_COMM_WORLD,-1,0);
K2:         ELC_MPI_Send(&rows, 1, MPI_INT, dest,
                           mtype, MPI_COMM_WORLD,-2,0);
            count = rows*NCA;

K3:         ELC_MPI_Send(a+offset*SIZE+0, count, MPI_DOUBLE, dest,
                         mtype, MPI_COMM_WORLD,-3,0);
            count = NCA*NCB;
K4:         ELC_MPI_Send(b, count, MPI_DOUBLE, dest,
                         mtype, MPI_COMM_WORLD,-4,0);

            offset = offset + rows;
          }

K5:       ELC_DoCKPT(-5,0);

          /* wait for results from all worker tasks */
          mtype = FROM_WORKER;
          for (i=1; i<=numworkers; i++) {
            source = i;
K6:         ELC_MPI_Recv(&offset, 1, MPI_INT, source,
                         mtype, MPI_COMM_WORLD, &status,-6,0);
K7:         ELC_MPI_Recv(&rows, 1, MPI_INT, source,
                         mtype, MPI_COMM_WORLD, &status,-7,0);
            count = rows*NCB;
K8:         ELC_MPI_Recv(c+offset*SIZE+0, count, MPI_DOUBLE, source,
                         mtype, MPI_COMM_WORLD, &status,-8,0);
          }
        }  /* end of master section */

/****************** Worker Task ******************************/
        if (taskid > MASTER) {
          mtype = FROM_MASTER;
          source = MASTER;
K9:       ELC_MPI_Recv(&offset, 1, MPI_INT, source,
                         mtype, MPI_COMM_WORLD, &status,-9,0);
K10:      ELC_MPI_Recv(&rows, 1, MPI_INT, source,
                         mtype, MPI_COMM_WORLD, &status,-10,0);
          count = rows*NCA;
K11:      ELC_MPI_Recv(a, count, MPI_DOUBLE, source,
                        mtype, MPI_COMM_WORLD, &status,-11,0);
          count = NCA*NCB;
K12:      ELC_MPI_Recv(b, count, MPI_DOUBLE, source,
                          mtype, MPI_COMM_WORLD, &status,-12,0);

          for (k=0; k<NCB; k++)
            for (i=0; i<rows; i++) {
              *(c+i*SIZE+k) = 0.0;
              for (j=0; j<NCA; j++)
                *(c+i*SIZE+k) = *(c+i*SIZE+k) +
                                *(a+i*SIZE+j) *
                                (*(b+j*SIZE+k));
```

```
            }

            mtype = FROM_WORKER;

K13:        ELC_MPI_Send(&offset, 1, MPI_INT, MASTER,
                         mtype, MPI_COMM_WORLD,-13,0);
K14:        ELC_MPI_Send(&rows, 1, MPI_INT, MASTER,
                         mtype, MPI_COMM_WORLD,-14,0);
K15:        ELC_MPI_Send(c, rows*NCB, MPI_DOUBLE, MASTER,
                         mtype, MPI_COMM_WORLD,-15,0);
          }  /* end of worker */

          ELC_MPI_Finalize();
          OnCallReturn();          // the end of main() function
} /* of main */
```

Appendix B.

Source Codes of *ELC_MPI Send()* and *ELC_MPI_Recv()*

```
#include <stdio.h>
#include "mpi.h"
#include "event.h"

int ELC_MPI_Send(void *buf,int count,MPI_Datatype datatype,int target,
                 int tag,MPI_Comm comm,int LABEL,int FID)
{
  int ierr;
  LOG *t_logTemp;

  // if CKPT is ongoing, probe CKPT request from other processes
  if (g_CKPT_FLAG==YES)     Probe_Request();

  // locate the event log for the target process
  t_logTemp=Seek_Log(target);
```

```
  //create the send event log (ELC_SEND)
  log_envelope(t_logTemp,tag,comm,ELC_SEND);

  ierr=MPI_Send(buf,count,datatype,target,tag,comm);

  return ierr;
}

int ELC_MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,
                         int tag,MPI_Comm comm,MPI_Status *pStatus,int LABEL,int FID)
{
  int ierr;
  LOG *t_logTemp;

  // ---------------------------------- RECOVER ---------------------------------------------
  if (g_RECOVER_FLAG==YES)
  {
   if (source==MPI_ANY_SOURCE)
   {      // for WILDCARD Communication
     t_logTemp=g_logHead->next;
     while (t_logTemp!=NULL)
     {
         if (t_logTemp->IntransitCounter>0)
           // if find in in-transit message logs, return;
           if (Search_Intransit(t_logTemp,tag,comm,buf))          return;

         if (t_logTemp->OrphanCounter>0)
          // if find orphan message, discard the repeated orphan message
           if (Search_Orphan(t_logTemp,source,tag,comm))       break;

         t_logTemp=t_logTemp->next;
      }
    }
    else {          // for non WILDCARD Communication
      t_logTemp=Seek_Log(source);

      // if there are in-tranist message logs
      if (t_logTemp->IntransitCounter>0)
         // if find in in-transit message logs, return;
         if (Search_Intranit(t_logTemp,tag,comm,buf))   return;
           // if find orphan message, discard the repeated orphan message
           if (t_logTemp->OrphanCounter>0)
             Search_Orphan(t_logTemp,tag,comm)         break;

         // check the finish of recovery
         if (t_logTemp->IntransitCounter==0 &&
           t_logTemp->OrphanCounter==0)
         {
           if (++g_RECOVER_COUNT==g_Process_NUM-1)
            g_RECOVER_FLAG=NO;
         }
     }

// ---------------------------------- Receive ---------------------------------------------
  ierr=MPI_Recv(buf,count,datatype,source,tag,comm,pStatus);
  if (source==MPI_ANY_SOURCE || tag==MPI_ANY_TAG)
  {
```

```
    source=pStatus->MPI_SOURCE;
    tag=pStatus->MPI_TAG;
  }
  // create the receive event log (ELC_RECV)
  t_logTemp=Seek_Log(source);
  log_envelope(t_logTemp,tag,comm,ELC_RECV);

// ---------------------------------- CKPT ----------------------------------------------------
  if (g_CKPT_FLAG==YES)
  {
    Probe_Request();

    switch (t_logTemp->flag)
    {
      // if all IN-TRANSIT messages have been logged, break
      case (LOG_DONE):        break;
      // if has received the CKPT request from THIS source process
      case (ENV_YES): {
          // log all in-transit messages
          log_intransit_message(buf,count,datatype,source,tag,comm,t_logTemp);
          break;
      }
      // if haven't gotten the CKPT request from this source
      case (ENV_NO): {
          // log this message in case of an in-transit message
          log_message(buf,count,datatype,source,tag,comm,t_logTemp);
      }
    }
  }

  return ierr;
}
```

Appendix C.

APIs for Uniprocess Checkpoint

/************************** Checkpoint API ****************************/

void ELC_DoCKPT(int LABEL,int FID);

Parameters:

LABEL: the label of calling statement.

FID: the evaluation sequence of an inline call

Description:

This routine is called by user explicitly to trigger a new global checkpoint.

Example:

C7: ELC_DoCKPT(7,0);

/************************** Program Variable API ****************************/

void OnVarDef(void *pAddr,int size);

Parameters:

pAddr: the starting address of the variable, e.g. &variable_name.

size: the size of the variable.

Description:

This routine is called immediately after a variable's definition statement to record the starting address and length of the variable.

Example:

```
int a;
OnVarDef(&i,sizeof(int));

double b[5];
OnVarDef(b,sizeof(double)*5);
```

void OnPtrDef(void **pAddr,int count);

Parameters:

pAddr: the address of the pointer, e.g. &pointer_name.
count: the number of pointers.

Description:

This routine is called immediately after a pointer's definition statement to record the address of the pointer.

Example:

```
int *a;
OnPtrDef(&a,1);

double *b[5];
OnPtrDef(b,5);
```

void OnStructureDef(StructureDesc descTemp);

Parameters:

descTemp: a descriptor of the structure composition.

Description:

This routine is called immediately after a structure's definition statement to record the structure's composition, which will be used by OnStrDef() to locate the inline pointers.

Example:

```
struct Node {
   int ID;
   struct Node *next;
   struct Property prop;
}

StructureDesc descTemp;

descTemp.name="struct Node";
strcpy(descTemp.name,"struct Node");
descTemp.ptrCount=1;
descTemp.ptrOffset=(void *)malloc(descTemp.ptrCount*sizeof(void *));
descTemp.ptrOffset[0]=sizeof(int)-1;
descTemp.strCount=1;
descTemp.strOffset=(void *)malloc(descTemp.strCount*sizeof(void *));
descTemp.strOffset[0]=sizeof(int)+sizeof(struct Node *)-1;
descTemp.strName=(char *)malloc(descTemp.strCount*sizeof(char *)*
                          MAX_STRUCT_NAME_LENGTH);
strcpy(descTemp.strName,"struct Property");

OnStructureDef(descTemp);
```

```
typedef struct StructureDesc {
   char name[MAX_STRUCT_NAME_LENGTH];
   int  ptrCount;
   int  *ptrDisp;
```

```
   int  strCount;
   int  *strDisp;
   char *strName;
}
```

Items:

name:	the structure name.
ptrCount:	the number of inline pointers.
ptrDisp:	an array contains the offsets of each inline pointer from the starting address of the strucutre.
strCount:	the number of inline structure items.
strDisp:	an array contains the offsets of each inline structure item from the starting address of the structure.
strName:	an array contains the names of each inline structure item.

void OnStrDef(void *pAddr,int size,char *pStrName);

Parameters:

pAddr:	the starting address of the structure variable.
size:	the size of the structure variable.
pStrName:	the structure name.

Description:

This routine is called immediately after a structure variable's definition to resolve the inline pointers.

Example:

```
struct NODE my_node;
OnStrDef(&my_node,sizeof(struct NODE),"struct NODE");

struct CARD his_node[5];
OnStrDef(his_node,sizeof(struct NODE)*5,"struct NODE");
```

/******** Program Execution Flow API **********/**

void OnCallEnter(int LABEL,int FID)

Parameters:

LABEL:	the label of calling statement.
FID:	the evaluation sequence of an inline call

Description:

This routine is called at the entry of a function to record the program execution flow.

Example:

```
void function(..., int LABEL, int FID)
{
  // Variable Definition
  ....

  // Record Execution Flow
  OnCallEnter(LABEL,FID);

  // Source Code
  ....

  // Remove Execution Flow
  OnCallReturn();
  return;
```

```
}
```

void OnCallReturn()

Parameters:

N/A

Description:

This routine is called before every RETURN statement in a function to tell libELC the execution flow has returned to the calling function.

Example:

```
void function()
{
  ....

  if (err) {
    OnCallReturn();
    return;
  }

  ....

  OnCallReturn();
  return;
}
```

/****************************** **Heap Memory API** ******************************/

void *ELC_malloc(size_t size)

Parameters:

size: the size of the allocated heap memory.

Description:

A wrapper function for malloc(). ELC_malloc() records the address and size of the allocated heap memory.

Example:

```
int *p=ELC_malloc(5*sizeof(int));
```

void *ELC_calloc(size_t nmemb,size_t size)

Parameters:

nmemb: the number of allocated heap memories.

size: the size of each allocated heap memory.

Description:

A wrapper function for calloc(). ELC_calloc() records the address and size of each allocated heap memory.

void *ELC_realloc(void *ptr, size_t size);

Parameters:

ptr the address of the heap memory to be re-allocated.

size the new size of the heap memory pointed by ptr.

Description:

A wrapper function for realloc(). ELC_realloc() also alters its record of the heap memory pointed by ptr.

Example:

```
p=realloc(p,10*sizeof(int));
```

void ELC_free(void *ptr);

Parameters:

ptr: the address to the heap memory to be released.

Description:

A wrapper function for ELC_free(). ELC_free() deletes the record of the heap memory pointed by ptr.

Example:

```
ELC_free(p);
```

/****************************** **File I/O API** ******************************/

FILE *ELC_fopen(const char *path, const char *mode)

Parameters:

path: the file path.

mode: file access mode.

Description:

A wrapper function for fopen(). ELC_fopen() records the file path and access mode to reconstruct the file descriptor during recovery.

Example:

```
fp=ELC_fopen("path/to/file","r+");
```

void OnFileDef(FILE **pAddr)

Parameters:

pAddr: the address of the file descriptor.

Description:

This routine is called immediately after the definition statement of a file descriptor to record its address.

Example:

```
FILE *my_fp;
OnFileDef(&my_fp);
```

int ELC_fclose (FILE *fp)

Parameters:

fp: the file descriptor to be closed.

Description:

A wrapper function for fclose(). ELC_fclose() deletes the record of the file descriptor pointed by fp.

Example:

```
err=ELC_fclose(my_fp);
```

int ELC_fcloseall()

Parameters:

N/A

Description:

A wrapper function for fcloseall(). ELC_fcloseall() deletes all file descriptor records.

Example:

```
err=ELC_fcloseall();
```