Checkpoint/Restart Implementation for OpenSHMEM

A Thesis presented

by

Delafrouz Mirfendereski

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2019
Stony Brook University

The Graduate School

Delafrouz Mirfendereski

We, the thesis committee for the above candidate for the Master of Science degree, hereby recommend acceptance of this thesis

Barbara Chapman
Professor of Applied Mathematics and Statistics, and of Computer Science

Rezaul Chowdhury
Associate Professor of Department of Computer Science

Michael Ferdman
Associate Professor of Department of Computer Science

This thesis is accepted by the Graduate School

Dean of the Graduate School
Numerous companies and research labs use parallel computing clusters in order to exploit their exceptional compute power to design products, obtain exceptional accuracy in simulations, process very large amounts of data, and achieve new research results. Therefore, the study and usage of parallel programming models and parallelization of programs and algorithms is of high relevance in many areas. Today, multiple parallel programming models exist, and many Application Programming Interfaces (APIs), libraries and languages are designed based on them. They include the shared memory model, message passing, and the Partitioned Global Address Space (PGAS) model.

OpenSHMEM is a PGAS specification providing the necessary API for communication between processes running in parallel, as well as synchroniza-
tion and atomic operations. The communication in OpenSHMEM is one-sided and done through putting or getting data to or from the shared memory portions. Every Processing Element (PE) has both private and shared memory parts, the latter being accessible from other PEs as well.

With clusters growing in number of nodes and cores, failures happen more frequently: as often as once every few hours. The types of failures vary in a range of simple bit flips to compute node failures. Besides, parallelizing programs and algorithms causes many complications that can easily lead to errors or crashes due to wrong usage of systems resources. Therefore, it is important to study the causes of such failures, their cost and frequency of occurrence, how to detect them and inform relevant system parts about the fault, and if or how to prevent, recover from or repair the failure. There are some techniques regarding resilience in serial systems and programs, but they cannot be directly applied to parallel ones due to higher complexity in larger and parallel systems.

There are many efforts to make parallel programming models more resilient. Some of the APIs or libraries are initially designed or developed (partially) resilient, such as Global Address Space Programming Interface (GASPI) or Global View Resilience (GVR); but some are not inherently fault tolerant and need additional tools or libraries for that matter, such as OpenSHMEM or Message Passing Interface (MPI). Due to the variety of fault types, there are multiple methods for resilience: some focus on node or process failures like User-Level Failure Mitigation (ULFM) library for MPI, and many provide Checkpoint/Restart (CPR) mechanisms so that a program’s progress is not lost after failure as the Fenix library does for MPI programs.

We attempt to make OpenSHMEM more resilient by studying types and costs of failures, the previously implemented and tested methods, and the applicability or priority of such practices in parallel programming models. Since CPR is seen as the primary strategy to prevent major data or progress loss in case of a failure, we are currently implementing the basic support for it in OpenSHMEM. In this model, the user (programmer) decides when and what
parts of data to checkpoint. We intend to test our implemented CPR using applications written in OpenSHMEM. Furthermore, we are making the communication and synchronization functions in OpenSHMEM ready to adapt to fundamental resilience techniques in the future. We also investigate possible contributions of compilers in making fault tolerance more optimal, e.g. suggesting the insertion of checkpoints in certain places of programs or reducing CPRs overhead by checkpointing only the changed parts of data. Our work is in collaboration with Rutgers University and the University of Tennessee Knoxville, and together we aim to make implementations of MPI and OpenSHMEM more fault tolerant and provide the foundations of resilience available in platforms that can be used by other parallel programming libraries as well.
Dedication

To my parents
for their love and sacrifice,
and to my siblings
for all the guidance and fun.
Contents

List of Figures x

List of Tables xii

List of Abbreviations xiii

Acknowledgements xv

1 Introduction 1

1.1 Motivation 1

1.2 Contribution 2

1.3 Thesis Structure 3

2 Parallel Programming Models 4

2.1 Some Parallel Programming Models 5

2.1.1 Message Passing 5

2.1.2 Partitioned Global Address Space (PGAS) 6

3 OpenSHMEM 8

3.1 Overview 8
5.2 Checkpoint/Restart Model ........................................... 38
    5.2.1 Overview of the Model ...................................... 38
    5.2.2 Different Types and Roles of Processes .................... 39
    5.2.3 Two Modes of Storing Checkpoints ......................... 41
    5.2.4 Provided Functions .......................................... 42

5.3 Implementation .................................................... 45
    5.3.1 Labels, Lists and Queries .................................. 45
    5.3.2 Carriers ..................................................... 47
    5.3.3 Checkpoint Table and Shadow Memory ....................... 48
    5.3.4 Queues ..................................................... 49

5.4 Usage and Example ................................................ 52

6 Results ........................................................................ 62
    6.1 Test Framework ................................................... 62
    6.2 Results ............................................................ 62

7 Future Work .................................................................. 67
    7.1 Improving Checkpoint/Restart Extension ....................... 67
    7.2 Compiler Contributions .......................................... 68
    7.3 More Resilient API ................................................. 69

8 Conclusion .................................................................. 70
List of Figures

2.1 Different parallel programming models. As illustrated, PGAS model stands between shared memory and message passing, with similarities to both of them [16] ............................................ 7

3.1 OpenSHMEM memory model [3] ................................. 11

4.1 A chart of different resilience methods, based on a more detailed work in [18]. Here we only discuss two relevant methods: redundancy and recovery. More details on other methods exist in [18]. ................................................................. 35

4.2 The ring topology used in heartbeat failure detection mechanism in[7]. Every entity sends periodic heartbeats to its observer and monitors its predecessor. ................................. 36

4.3 An instance of CDs’ hierarchy. Errors are handled either within the same level or sent to upper levels for management. ....... 36

5.1 The different types of PEs in both of the checkpointing modes and their relations. A PE’s type can change when a failure is detected. Recovering a dead PE often needs system support and is not included in this work. ................................. 57
5.2 The different roles of PEs in both of the checkpointing modes. The transition between different roles is described here. All PEs maintain their roles if no failure happens.

5.3 Active PEs are the same in both modes. They keep one local copy of checkpoints of their own data, called shadow memory. They fill their requests for reservation and checkpoint in carriers and send them to storage PEs. Active PEs are either of type original or resurrected. In this picture, there are $k$ active PEs. Since reservation and checkpoint queues are part of symmetric memory, they also exist on active PEs but are not used.

5.4 Spare PEs in many-copy mode all have storage role. Queues are symmetric variables and exist on every PE but they are only used if they belong to a storage one. Queues, checkpoint table and shadow memory, and carriers are explained in detail in Section 5.3.

5.5 Spare PEs in two-copy mode are either storage or dormant PEs. More on queues, checkpoint table and shadow memory, and carriers is studied in detail in Section 5.3.

6.1 Different frequency intervals used for checkpointing in both models.

6.2 Overhead of one or two rollbacks compared with a case with no rollbacks.
List of Tables

6.1 Comparison of matrix multiplication program with no checkpoints, and checkpoints inserted every 400 or 1000 iterations, for both many-copy and two-copy models. . . . . . . . . . . 64

6.2 Comparison of matrix multiplication program with no checkpoints, and checkpoints inserted every 400 or 1000 iterations, with or without rollback, for both many-copy and two-copy models. The number of active PEs is all cases is 8. . . . . . . . . . . 65
Acronyms

AMP Atomic Memory Operation. 10, 15

API Application Programming Interface. iii, iv, 3, 5–7, 12, 13, 28, 31, 32, 37, 70

CD Containment Domain. x, 24, 25, 36

CPR Checkpoint/Restart. iii–v, viii, ix, 2, 3, 20, 22–26, 29, 32, 37, 38, 42, 43, 52, 55, 56, 67, 68, 70

CRB Consensus Recovery Block. 24

FEC forward error correction. 23

FT Fault Tolerance. 24

FTI Fault Tolerance Interface. 24

GASPI Global Address Space Programming Interface. iv, viii, 28

GFLOPS Giga Floating-Point Operations per Second. 4

GVR Global View Resilience. iv, viii, 27, 37

HPC High Performance Computing. viii, 20–22, 27, 37

ILP Instruction Level Parallelism. 4
MPI Message Passing Interface. iv, v, viii, 5, 6, 11, 29, 37, 70

MSPE Main Spare PE. 40, 44, 52

PE Processing Element. iv, x–xii, 8–15, 32–34, 38–52, 55–62, 64, 65, 68

PGAS Partitioned Global Address Space. iii, vii, x, 5–8, 11, 28, 70

RB Recovery Block. 21, 24, 25

RDMA Remote Direct Memory Access. 28

RMA Remote Memory Access. 5, 6, 10–12

SCR Scalable Checkpoint/Restart. 23, 24

SPMD Single Program Multiple Data. 6, 9

TFLOPS Tera Floating-Point Operations per Seconds. 4

TMR Triple Modular Redundancy. 20

ULFM User-Level Failure Mitigation. iv, 29
Acknowledgements

I would like to thank Prof. Chapman and Tony Curtis for their help and guidance throughout the course of this project and my education.
Chapter 1

Introduction

1.1 Motivation

With the increase in complexity of systems and applications, as well as emergence of new and more complicated programming paradigms, especially parallel programming models, the increase in human and system errors is inevitable. In fact, it is predicted that the overhead and extra cost caused by failures, errors, and pauses in execution, will delay reaching the goal of exascale performance. Therefore, recently many scientists have suggested and studied the urgency of addressing failures in high performance computing, as well as providing research plans on this matter and making systems and applications more resilient.

Many fault tolerance methods have been developed and tested on serial programming environments and systems. This job is more difficult to be applied to parallel programming models, due to the typical complications of parallel programs, the complexity of systems where parallel application execute on, and the inherent non-determinism of such applications due to message transfers and remote data writes. A good resilience technique for parallel models must not only be able to handle the complications of parallel programs, but also leverage it to increase the performance of the technique.
Moreover, with the additional complexity and number of components in today’s systems and clusters, it is needed to develop proper failure detection techniques that are able to discover a failure or an error, and report the problem to the proper components of the system. On the other hand, different levels and layers of applications and systems need to be capable of communication with each other regarding these errors, and compromising on the best course of action.

Due to the variety of potential errors and failures in parallel systems and applications, and the difficulty in resolving them, it is unlikely that a single work be able to provide all the solutions. Currently some libraries exist which provide solutions for a category of failures, or methods needed to detect and recover from them. Therefore, it is necessary to leverage different libraries and enable them to work properly and in harmony with each other.

Checkpoint/Restart (CPR) is a very popular method of resilience in high performance computing: its cost and overhead is lower than other techniques, it is easy to use for programmers, and prevents the loss of data and program’s progress as well as providing a recovery mechanism. In this work we present a new checkpoint/restart extension for OpenSHMEM, a parallel programming library.

1.2 Contribution

This work is an effort towards making OpenSHMEM more resilient against failures and errors. We have designed and implemented a Checkpoint/Restart extension for OpenSHMEM which is easy to use, since it is designed partially similar to popular and efficient works for other libraries. This extension also tries to improve the previous CPR library implemented for OpenSHMEM by decreasing the memory usage and providing more flexibility for users. The model, its implementation, and performance are provided in this thesis.
1.3 Thesis Structure

This thesis is presented in eight chapters. In the first chapter, we have introduced our work, motivations, and contributions. In the second chapter, we would review the importance and development of parallel programming needs and models. Some important categorizations of parallel programming models are presented and three related classes of them are explained.

The third chapter includes the necessary knowledge of OpenSHMEM for the purpose of this thesis. The memory and communication models of OpenSHMEM are explained, and then we present a summary of OpenSHMEM’s Application Programming Interface (API).

In the fourth chapter, we review relevant aspects of fault tolerance. The typical failures of different systems and faults in programming are shown, and then main concepts in resilience are introduced. Then we go through some well-known methods of fault tolerance, and review some implementations and resilient libraries with similarities to either OpenSHMEM or our work.

We present our work to date to make OpenSHMEM resilient in the fifth chapter. We explain the plan for checkpoint and recovery in OpenSHMEM, along with details about how it was implemented, and the two models designed for this job.

The result of using our implemented CPR for OpenSHMEM on applications is illustrated in chapter six.

We suggest possible research continuing this thesis’ work in chapter seven. Multiple ways for improving or extending the current CPR are considered in this chapter. Moreover, we introduce how compilers can contribute to better performance of this implemented library. We also show the ways OpenSHMEM’s API can be more fault tolerant. Finally, the conclusion of this thesis is gathered in chapter eight.
Chapter 2

Parallel Programming Models

For years, increasing the clock frequency of CPUs was the principal strategy to increase the speed of computation, bringing Giga Floating-Point Operations per Second (GFLOPS) to desktop computers and Tera Floating-Point Operations per Seconds (TFLOPS) to data centers. Starting from 2003, due to physical constraints and high energy consumption, that increase in performance has been mainly achieved through making programs run in parallel on either multicore or many-thread hardware [29].

Meanwhile, numerous companies and research labs use parallel computing clusters in order to exploit their exceptional compute power to design products, obtain accuracy in simulations, process very large amounts of data, and achieve new research results. Therefore, the study and usage of parallel programming models and parallelization of programs and algorithms is of high relevance in many areas.

One way to achieve parallelism is through Instruction Level Parallelism (ILP), which is the components of a computer working simultaneously to perform logical or memory access operations for example [14]. ILP is nowadays available on many microprocessors. Other methods of parallelization other than Instruction Level Parallelism (ILP) or parallel compilers, is through parallel programming, while the latter usually leads to better and more flexible
performance [17]. In the next section, we would investigate some parallel programming models.

\section*{2.1 Some Parallel Programming Models}

Today, multiple parallel programming models exist, and many APIs, libraries and languages are designed based on them. Various categorizations of parallel architectures or programming models based on different criteria are suggested, but most parallel models fall into two broad classes: shared memory or distributed memory. In shared memory systems, all parallel entities of the program access the same memory address space. However, in distributed memory model, each processor has and uses its own memory. The most popular API to be used on shared memory systems is OpenMP, and MPI is the one for distributed memory systems [17].

To achieve higher performance, a variety of hybrid systems are designed to advantage from both shared and distributed architectures, and currently systems are moving towards more complex (or heterogeneous) architectures and models [17, 16]. The hybrid models are very diverse, most of which are out of this dissertation’s scope. Partitioned Global Address Space (PGAS) is an example of such models, with similarities to both shared memory and message passing models. Due to their relevance to this work, we further explore MPI and PGAS models.

\subsection*{2.1.1 Message Passing}

The most famous interface for message passing model is MPI. Data transfer in MPI happens from the memory space of one process to another process’s through a collaborative operation. Other than this point-to-point type of communication, MPI provides collective communication and synchronization, parallel I/O, Remote Memory Access (RMA) operation, and dynamic process
The main type of point-to-point communication in MPI is two-sided, meaning both parties should participate in the call. Two-sided communication is mostly a blocking operation, meaning the sender waits until the message is received by the receiver. To avoid a deadlock when sending and receiving data at the same time, `sendrecv` which is a simultaneous send and receive operation, can be used. However, a more generalized solution would be to use non-blocking send and receive functions [25].

Since version 2.0, one-sided communication or RMA is also supported by MPI [23]. These functions only need the participation of one party. Broadcasts and reductions are examples of collective communications in charge of sending data to or collecting it from a group of processes.

### 2.1.2 Partitioned Global Address Space (PGAS)

Before the emergence of Partitioned Global Address Space programming model, parallel program fell into two main categories of shared memory model (e.g. OpenMP) or message passing (e.g. MPI). PGAS model stands between the two: its shared memory part that can be accessed by all the parallel entities like shared memory model, and its communication between processes with a cost that resembles message passing. The similarities of these models is shown in Figure 2.1.

Programming languages or APIs following PGAS model mostly follow the four characteristics below:

1. Designed for parallel programming: Parallel execution models of PGAS models follow Single Program Multiple Data (SPMD), Asynchronous PGAS, and implicit parallelism. In the first, a single program executes on different data. In the second, a parallel entity starts the program and then spawns others for different tasks. The third model exists in languages that apply parallelism to parts of the program, such as loops.
Figure 2.1: Different parallel programming models. As illustrated, PGAS model stands between shared memory and message passing, with similarities to both of them [16]

2. Specified partitioning of global address space: The placement of parallel entities and the cost of accessing memory in other entities must be defined by the APIs or languages of PGAS model.

3. Data distribution over different partitions: Data distribution over the parallel entities should be defined. Some examples of such distributions are block or cyclic distributions.

4. Data access model: It should be described by the API or language what data can be distributed and remotely accessible, and how that access can be requested or happen.
Chapter 3

OpenSHMEM

Before OpenSHMEM came into existence, several implementations of SHMEM were developed with differences in API syntax, behaviors and performance. This made the portability of applications written in SHMEM problematic. OpenSHMEM started as an effort to bring together different implementations of SHMEM with one standard specification and a reference implementation. OpenSHMEM is a communications library based on PGAS programming model. It helps the application programmer to start up a number of Processing Elements, or PEs, in parallel, let them do computations on different data, communicate and synchronize with each other whenever needed, and then finalizing the parallel processes and return to serial programming. In this chapter, we would further introduce this specification and investigate its features and performance [10].

3.1 Overview

As explained earlier, OpenSHMEM is a parallel programming specification and library based on PGAS model. This means that in an OpenSHMEM program, every PE has access to both private and shared memory locations. Through the shared memory address space, PEs can communicate with each other and
exchange pieces of information. However, this communication is one-sided and happens with only one of the parties participating in it and the other one unaware of the data transfer. This means a PE can put data into another PE’s shared memory portion, or get some information from it, while the remote PE continues working its computation. That leads to the overlap of communication and computations which helps OpenSHMEM programs to scale and perform very well [26, 15].

OpenSHMEM follows a Single Program Multiple Data (SPMD) programming style. This means that the same program is ran by every PE but on possibly different data. This data can reside in either the remotely accessible or private part of a PE’s memory. PEs in OpenSHMEM are numbered from 0 to $n - 1$ where $n$ is the number of PEs. Programmers can use PE numbers to control which parts of program to execute on what PEs, e.g. PE 0 can do some computation on the data that PE 1 has put in memory of PE 0, while PE 0 does no communication.

OpenSHMEM API allows the user (application programmer) to start up and end the parallel part of the program, query runtime variables such as the number of PEs and PE numbers, create and manage shared data objects, let some or all of PEs communicate and synchronize with each other, while each PE would be able to perform basic atomic operations\(^1\) and create mutually exclusive regions.

As of version 1.4, OpenSHMEM also provides essentials for thread-safe communications. Due to the increasing popularity and usage of heterogeneous or hybrid parallel programming models, support for thread-safe communications was added to OpenSHMEM’s API and it can now be used together with other threading libraries such as OpenMP or POSIX. There are four different threading modes available based on the number of threads per

\(^1\)An operation done by a parallel entity on a piece of shared memory is atomic if it is seen as an instantaneous change by other entities. This means no parallel thread or process can see the change in data when it is half-done, and meddle in or change the atomic operation. Atomic operations provide mutually exclusive changes to shared data.
PE and the type of their access to OpenSHMEM’s interface: SHMEM_THREAD_SINGLE, SHMEM_THREAD_FUNNELED, SHMEM_THREAD_SERIALIZED, and SHMEM_THREAD_MULTIPLE. All Atomic Memory Operations (AMPs), Remote Memory Access (RMA) and memory ordering routines function upon communication contexts. These contexts are user defined and responsible for ordering and completion of mentioned operations, and this is how they assist in computation and computation overlap. When PEs are multi-threaded, contexts also manage the communications of threads within a PE.

3.2 OpenSHMEM’s Memory Model

Every Processing Element (PE) in OpenSHMEM has two parts of memory: one private to the PE itself, and one remotely accessible by other PEs. It is through the remotely accessible portion that communications in OpenSHMEM take place. This shared, or as called in OpenSHMEM, symmetric memory has two parts itself: one that includes all the global and static variables, and one that is called symmetric heap. The PEs can create data objects on the symmetric heap through a collective function call of shmem_malloc(size), where size is the amount of memory in bytes that will be allocated to that object on each PE. Collective calls in OpenSHMEM are functions that need to be called by all PEs simultaneously.

Atomic operations in OpenSHMEM also take place on symmetric data objects. The atomics guarantee no other atomic operation inside OpenSHMEM can access the object at the same time, but it can be changed by other communication functions in or operations outside of OpenSHMEM. Also many of the collective communication functions in OpenSHMEM work only with or on symmetric data. These functions are reviewed later in this chapter.
OpenSHMEM’s Communication Model

OpenSHMEM follows PGAS communication and memory model. Having a globally accessible address space, it is easy for libraries in PGAS style to implement one-sided communication: the parallel entities of the program only need to put data to or get it from one or PEs. One-sided communications happen without the acknowledgment of remote (or destination) PE (or other kinds of parallel entities). This matter speeds up the execution since at least one of the parties in the communication can still continue its work.

Traditional message passing models are developed usually with focus on two-sided communications, although MPI introduced one-sided ones in MPI 2. The current MPI’s one-sided communications are done through data windows, an object that exposes a part of a process’s memory to other selected processes for the purpose of RMA operations [35]. One-sided communication in OpenSHMEM can be expected to perform better than MPI’s, due to the overhead of windows creation in MPI and OpenSHMEM’s inherent one-sided communications. Some research have proven this point [33].
3.4 OpenSHMEM’s API

OpenSHMEM’s API is designed to support the needs of parallel programmers. It includes routines for library setup and queries, RMA and collective communications, synchronization and ordering, basic atomic operations, symmetric memory management and query, communication management through communication contexts, and mutual exclusion.

1. Library setup and queries: enable the user to initialize the library either single or multi-threaded, and query the number of active PEs and the number associated with each PE.

2. RMA and collective communications: provide the needed communications between PEs. It includes two main categories:

   - RMA: RMAs are the equivalent of point-to-point communications only one-sided. They are performed through different types of put and get function calls.

   - Collective communications: as shown by the word collective, these communications happen with the participation of all PEs in the active set. They either help broadcast information, collect results from specific symmetric data objects, or calculate a reduced result from them.

3. Synchronization and ordering: handle three major types of ordering and synchronization,

   - Fence: ensures the changes made to a symmetric object (through put, atomics, or memory store operations) happen in the order of program’s instructions.

   - Quiet: ensures completion of RMA or memory store operations on a remote symmetric data object.

   - Barrier: helps all PEs reach the same point in the program together, and finish remote and local work prior to the barrier.
4. Atomic operations: support atomic increments, swaps, add and bitwise operations on a PE’s own or a remote symmetric data object.

5. Symmetric memory management and query: PEs can create a symmetric data object through the collective allocation calls, re-allocate memory to it or delete it. Also some queries are available to check if an address on another PE is remotely accessible.

6. Communication management: through contexts, the user can manage the computation and communication overlap in a single or multi-threaded PE.

7. Mutual exclusion: provide locks for mutually exclusive access to a region of symmetric memory. There are routines to help acquire a lock, test if it is free, and release (or clear) it.

For the purpose of this dissertation, we study some parts of OpenSHMEM’s API in more detail. The functions studied below follow the C11 standard which is the standard we used for our work as well. Other standards and many more functions not mentioned here can be found in OpenSHMEM’s complete API documentation[3].

3.4.1 Library Setup and Queries

This part of the API assists in starting up the library and finalizing it, as well as providing information on processes and their symmetric memory.

3.4.1.1 Initialization and Finalization

Any part of the program between `shmem_init()` and `shmem_finalize()` can leverage the OpenSHMEM’s provided facilities. As of OpenSHMEM 1.4, by calling `shmem_init_thread(int requested, int *provided)` the library can be also initialized with threading support. Any usage of OpenSHMEM functions
before initialization or after finalization can result in unwanted or unknown behavior.

### 3.4.1.2 Process Queries

OpenSHMEM provides information on the processes, using `shmem_my_pe()` and `shmem_n_pes()`. The first returns the number of the process calling the function. The returned result can be used to divide the work, have different program flows for different PEs, and help with point-to-point communication and atomics. The latter return the total number of PEs running the program.

### 3.4.1.3 Memory Query

We can determine if a PE or an address on a PE’s memory is remotely accessible or not. The function `shmem_pe_accessible(int pe)` answers the first and `shmem_addr_accessible(const void *addr, int pe)` returns information on the second query.

### 3.4.2 Communication

OpenSHMEM provides one-sided point-to-point communication through various puts and gets functions for sending and receiving data respectively. Function `shmem_put(TYPE *dest, const TYPE *source, size_t nelems, int pe)` copies `nelems` blocks of data of type `TYPE` from `source` at PE calling the function to process `pe` at address `dest`. Similarly for receiving data we use `shmem_get(TYPE *dest, const TYPE *source, size_t nelems, int pe). Other types of puts and gets exist to help with non-blocking communication, transferring only one element, or strided data.

Collective communications consist of broadcasts, all-to-all’s, reductions and collects. More information on these functions can be found in [3].
3.4.3 Synchronization and Memory Ordering

This set of OpenSHMEM operations can synchronize some or all of processes or provide the ordering of memory operations. They will help the user to control the flow and order of program’s instructions. Collective synchronization functions barrier and barrier-all help some or all of the processes meet at a point in the program.

Point-to-point synchronization functions are wait and test, for example

```
shmem_wait_until(TYPE *ivar, int cmp, TYPE cmp_value)
```

pauses the calling PE’s execution until the values in `ivar` and `cmp_value` meet the comparison criteria in `cmp`. The value of `cmp` should be set to one of the OpenSHMEM’s defined comparison constants, i.e. `SHMEM_CMP_EQ`, `SHMEM_CMP_NE`, `SHMEM_CMP_GT`, `SHMEM_CMP_GE`, `SHMEM_CMP_LT`, and `SHMEM_CMP_LE`.

The ensure the order of different memory operation such as puts and AMOs, fence and quiet functions can be used.

3.4.4 Atomics

Atomic Memory Operation (AMP)s handle reading, and in cases updating the symmetric memory variables. The atomicity of such operations guarantees that the data saved in a variable will not be changed during an AMP by other operations, and therefore the atomic read and/or write to the variable will have the exact result as user wants.

The atomic functions in OpenSHMEM are either fetching or non-fetching. The first category return the original value of the variable before the Atomic Memory Operation (AMP) makes changes to it. The second type, however, only update the value atomically.

Examples of the fetching type are `shmem_atomic_fetch(const TYPE *source, int pe)` which returns the value stored in `source` at remote process `pe`, while `shmem_atomic_fetch_inc(TYPE *dest, int pe)` increments the value in `source` by one and returns the amount before increasing. Operations that can be combined with fetch other than increment are
add, and, or, xor. Fetching functions also include swapping ones.

The second category provide atomic set, inc, add, and, or, and xor. For example, `shmem_atomic_set(TYPE *dest, TYPE value, int pe)` sets the amount of `dest` at process `source` to `value`.
Chapter 4

Fault Tolerance

Scientists are trying to achieve more computing speed and power to move from the petascale\(^1\) computing to exascale\(^2\). A requirement for that matter, is larger systems consisting of many processing and memory components as well as interconnections. But as a system grows in size and complexity, the rate of errors and failures happening in it will increase due to the following reasons\(^{45}\):

- Hardware’s complexity would cause more faults. Moreover, as the number of components in a system increases, there will be a higher chance that a component fails in such system.

- A more complicated hardware needs more complicated software, and a larger system requires many failure and energy management programs. These would add in complexity and therefore errors.

- Programmers are developing larger or more complex applications. They utilize modules, libraries, and different languages in a single application.

---

\(^1\)A petascale system is one capable of reaching petaflops performance, i.e. \(10^{15}\) floating point operations per second.

\(^2\)An exascale system can achieve exaflops performance, meaning \(10^{18}\) floating point operations per second.
On the other hand, algorithms are becoming more complex to deliver better performance or lower energy consumption. Therefore applications are becoming more prone to errors.

The rest of this chapter reviews important concepts regarding errors and failures, and techniques to decrease or mitigate them in systems, and some related work to make different parallel programming models or libraries more resilient.

4.1 Faults, Errors and Failures

Faults, errors, and failures are concepts that are interchangeably used most of the time. But according to [45]:

- Fault: is the cause of errors. Faults can be active or inactive depending on whether they cause errors or not. They usually exist on one component of the system, and they can be hard, i.e. a system fault, or soft.

- Error: is a state of system which does not work properly and may start a failure. They can be detected, or stay silent/latent.

- Failure: brings the system to unwanted or incorrect service. Failures may be perceived differently or identically by the users. They can also be detected or not.

There exists many mechanisms to detect, reduce, predict, recover from, or repair faults, errors or failures. The rest of this section addresses some of these techniques and concepts in detail.
4.2 Fault Tolerance Concepts

To make a system resilient, some actions on hardware, system, or application level must be taken. The hardware level solutions will lead to the least changes in applications but is costly due to the changes made to hardware. Application level option requires developers to handle resilience and needs many changes in applications. It sometimes needs the system or hardware support and cannot tolerate errors alone. System level methods require both of hardware and software to manage faults and errors. This option, similar to hardware one, needs little or no change to applications.

According to [45], solutions applied for resilience are of the following categories: prevention, prediction, tolerance, detection, containment, recovery, diagnosis, and repair. The aim in prevention methods is to reduce the number of errors occurrences, and repair ones try to repair or replace the failed components. The difference between detection and diagnosis is that in the first errors are found and not their causes, but in second we try to find the root of an error which may help us reduce, predict or prevent errors in future. Tolerating errors will help us prevent failures and containing them tries to minimize the range of their effects on the system. Most of the prediction methods need a log of history and causes of errors happened on the system as well as an algorithm to leverage that information and predict errors [18].

Recovery methods fall into two categories: Backward or forward recoveries. In both, the effort is to escape an erroneous or failed state and return to a safe one.

- Forward Recovery: It brings the system to a state to continue the work safely and error free. This method cannot be implemented without multiple copies of relevant components, such as multiple modules running the same part of an application. Forward recovery is usually used when providing a service will little or no delay or pause is more important than recovery or the whole system working.
• Backward Recovery: Backward recoveries move the state of system to an older safe one and restart from then, and therefore need one or multiple checkpoints of previous states to be able to return to them. Another name for this method is Checkpoint/Restart (CPR). CPR is among the most used techniques in HPC, since it prevents long-running applications from restarting anew when a failure happens and only needs little time to get the application running error-free. CPR is further studied in Section 4.3.3.

Aside from recovery method explained above and further studied in Section 4.3.3, other main systems of resilience fall into one of the following categories[18]:

• Migration: This is a prevention method. When a failure is predicted to happen, the possible part of application that will be affected by it will be migrated to another node or process. The prediction relies on prediction algorithms which use the proper log files from the system regarding its error rates and scalability. It is hard to provide precise predictions since some failures are not reported in the mentioned logs, and even if they are, prediction may still be not accurate enough.

• Redundancy: This is to provide multiple instances of hardware or process or other components, in case one or some of them fail. The main approaches are as follows:
  
  – Process pair: The model consists of a set of active and passive (backup) processes on different processors. When an active process fails, a backup fills its place.

  – Triple Modular Redundancy (TMR): Three modules run the same program, then their results are ran through a voting system to produce the final result. This method is not immune to two modules failing, or faces challenges with non-deterministic programs.
– Other methods like N-version Programming[11], Recovery Block (RB), or N Self-Checking Programming provide more than three modules or copies of the same program, and they either use a single voting system to check all the outputs or use a self-checking program per module.

• Failure semantics: A set of anticipated failures are provided within the fault tolerant system and in case a failure is detected, the proper and previously provided recovery action will take place. This system’s accuracy depends on the ability of the designer to provide a proper set of failures and proper actions associated with each of them.

• Failure masking: It ensures that the system still provides for user needs even in case of a failure. Failure masking, similar to forward recovery, is appropriate for systems where on-time service to user is more important than recoveries and detections.

4.3 Fault Tolerance Techniques in HPC

In this chapter we expand some famous or related implementations of methods mentioned above for HPC. These systems may use one or more of the studied techniques.

4.3.1 Fault Detection

A popular failure detection method at process or compute node level uses frequent heartbeat messages. An entity, which can be a process, a compute node, or daemons associated with groups of the previous two, form a ring topology depicted in Figure 4.2. Any entity sends frequent heartbeat messages to its successor, and receives them from its predecessor. If one component does not receive a heartbeat from the one it is monitoring until a deadline,
the component is believed dead and the information is propagated to all the appropriate components. This method can tolerate a maximum of \( \log n - 1 \) overlapping failures, which happen before the knowledge of previous one is received by all alive components. The appropriate propagation method is explained next.

### 4.3.2 Failure Knowledge Propagation

Propagation algorithms usually use broadcasts and efficient broadcasts are implemented on binary trees. The broadcast algorithm mentioned in [2] is implemented on binomial broadcast trees. Since the algorithm works for \( 2^k \) nodes, for a system with \( n \) components, two broadcasts are called, one for entities \([1..2^k]\) and one for \([2^k + 1..n]\), where \( k = \lfloor \log n \rfloor \).

The broadcast only finishes when at most \( k - 1 \) entities fail, which explains the criteria in Section 4.3.1. The worst-case time complexity of the algorithm happens in case of over-lapping failures, and is \( O((\log n)^3) \). The best time complexity is \( O(\log n) \).

### 4.3.3 Checkpoint/Restart (CPR)

CPR methods designed for HPC differ in various aspects: the locale of checkpoint storage, redundancy, checkpoint and recovery methods.

#### 4.3.3.1 Global and Local CPR

Many CPR implementations use global centralized checkpointing, meaning all the checkpoints will be stored on a centralized location on disk. This storage system introduces much I\( \backslash \)O overhead and delays in checkpoints, such that in some models it consumes up to 50\% of execution time, even in near ideal conditions (with near perfect checkpoint frequency and parallel I\( \backslash \)O)[36].

To reduce the overhead caused by global checkpoints, diskless checkpoints
were studied [38]. Diskless checkpoints use on-memory checkpoints and/or encoding the checkpoints and storing them on the processes themselves. The encoding methods consist of parity codes, erasure codes\(^3\), and Reed-Solomon codes[30]. Although they are faster than global checkpoints, diskless checkpoints cannot tolerate all types of failures, a total or multi-node crash is an example of them.

To leverage the resilience of global checkpoints and performance of diskless checkpoints, two-level or multi-level CPR systems have been introduced. Two-level systems provide storage either on memory and disk such as [48], or either on local and remote disks as mentioned in [37]. The idea behind two-level checkpoints is that in case of process failures, its checkpoints can be retrieved from any place except its own memory, while multiple process, node or whole-system crashes require more stably-stored checkpoints. In this method’s terminology, less stable checkpoints are called 1-checkpoint and can tolerate less failures, while N-checkpoints are more resilient and stable [49, 48]. Usually, the two different levels of checkpoints are done with different frequencies to lower the overhead of writing on disks.

By generalizing the idea mentioned above, checkpoints can be stored on/from multiple levels. A Markov model has been presented for multi-level CPR in [24] to calculate system efficiency. Failures and checkpoints in multi-level checkpointing are assumed in \(L\) different levels, with checkpoints of level 1 to be least expensive and resilient, while \(L\)-level checkpoints to be most costly and failure-tolerant. A checkpoint of level \(k\) can resolve and handle all the failures from levels 1 to \(k\) [34]. Scalable Checkpoint/Restart (SCR) is an implementation of such method and provides caching checkpoints in the local storage as well as intermittent flushing of cached checkpoints to the parallel file system.

\(^3\)Erasure codes are a method of forward error correction (FEC), which is form of forward error recovery. Erasure codes insert some encode parts in a \(k\)-word data and lengthen it to \(n\) word. Encoding and insertion methods are chosen such that by having any \(k\) parts out of \(n\), the original \(k\)-worded data can be rebuilt.
Fault Tolerance Interface (FTI) is also a famous three-level CPR system: it uses an intermediate level between local and global levels. Similar to SCR, the first and last levels of storage in FTI are local compute node storage and parallel file system, but the second is chosen to be either Partner method or erasure codes. In Partner technique, a second node holds the checkpoints of its partner. Therefore, while the first level tolerates soft or process failures, the second level can tolerate multiple node failures in case a node’s partner does not fail. For erasure codes technique, Reed-Solomon or XOR methods can be used to retrieve the original data. Erasure codes are also vulnerable to multiple node failures. FTI leverages FT-dedicated threads to take care of checkpointing, which reduces the CPR’s overhead. The CPR provided by FTI works between two functions calls of `FTI_Init()` and `FTI_Finalize()` [4].

Another example of multi-level checkpoints are Containment Domain (CD)s. They are fully inspected in Section 4.3.3.2.

### 4.3.3.2 Hierarchical Checkpoints

Hierarchical Checkpoint/Restart techniques are a combination of multi-level CPR as well as containment methods. In this system, errors can be detected and tolerated at different levels. Famous systems implementing this method are Recovery Block (RB)s [40] and Containment Domain (CD)s [13, 46].

Recovery Blocks provide different alternatives in case of failures happening. Upon arrival at an RB, the state of system is checkpointed and the main program (or alternate here) is executed. The result of main alternate is tested by an evaluation (i.e., failure detection) system, and if an error is found the recovery mode starts and last checkpoint is revived. Then the second alternate is ran and evaluated, until program exists successfully and discard the checkpoint, or all the alternates are executed or a timeout is reached and program ends with failure. The different alternates help contain soft errors at application level and not cause hard errors at base levels.

Consensus Recovery Block (CRB)s are an attempt to combine N-version
programming together with RBs [42]. In N-version programming, N independent versions of the same program are executed and tested against a single voting system to choose the correct output [18]. The common assumption made for N-version technique is that versions are independent and therefore no common fault can exist among them. The limitations of both N-version programming and RBs are high resource usage, as well as challenges in handling non-deterministic applications.

Containment Domain (CD)s can be seen as small independent boxes of resilience at different levels. They consist of four parts as shown in Figure 4.3:

- Preserve: This is the place to hold the local checkpoints.
- Domain Body: It holds the main application ran at this level.
- Detect: Containing detection algorithm of this level, this component is in charge of finding and reporting errors.
- Recovery: Each CD is provided with its own recovery method which may be or not be able to handle the discovered error at this level. If it can, it will reload the checkpoint in the CD’s own Preserve. Otherwise, it asks for recovery either from its siblings or its parent. An error such as multiple node failures can be tossed up until reaching the root of hierarchy, causing the root-level recovery which is equivalent of a global recovery.

Containment Domain (CD)s are popular due to their incredible flexibility at both detection and recovery algorithms, as well as containing errors in lowest levels possible. Their full semantics is available at [47].

4.3.3.3 Full and Incremental Checkpoint/R:estart

In full checkpoints, every time all of the data or memory is checkpointed, while only a small part of it may have changed since the last checkpoint.
This unwanted storage of data increases the I/O usage and/or time or other resources the CPR method uses.

In oppose to full checkpoints, incremental checkpoints only store the parts of data or memory that has changed since the last checkpoint. In systems that checkpoints are done by hardware, incremental checkpointing usually need a software support, such as [1].

To determine what part of data has changed since the last checkpoint, usually a hashing function is used [1, 50], but another algorithm is presented in [39]. In this method, the difference of current data and the last checkpoint is calculated by a bitwise exclusive or function. If the changes of data is small w.r.t the size of data, the difference will have many 0 bits and can be easily compressed and sent for storage.

Although incremental CPR decreases the data transfer and therefore checkpointing time, sometimes it leads to slower rollback recoveries [50]. Since rollbacks happen less frequently than checkpoints, using full and incremental checkpoints together seems like a good solution to leverage both of their advantages, as explained in Section 4.4.3.3.

4.3.3.4 Coordinate or Uncoordinated Rollback Recovery

Coordinated rollbacks result in recovery of all processes running the program, even if most of them have survived and not been affected by the failure. Any rollback results in repeating some parts of the application (whatever has been executed since the last checkpoint), which results in slower CPR in case of coordinated rollbacks.

In uncoordinated recoveries, only failed processes and the ones affected by them (through communication for example) will rollback to the previous checkpoint. A necessity for uncoordinated rollbacks are message logging, since it is through them that processes affected by a failure are known. Although recovery of only a group of processes saves time, but message logging introduces
much overhead during checkpointing [9].

4.4 Related Work

In this section we review some of the fault tolerance techniques applied to HPC. These have affected our work or may influence it in the future.

4.4.1 Global View Resilience (GVR)

Global View Resilience (GVR) is a resilient parallel programming model using arrays distributed across nodes[12]. It keeps multiple versions of data structures and therefore enables the application to survive latent errors. Silent or latent errors are ones that are hard to detect and cause miscalculations and wrong results that can stay unnoticed for a long time[31]. GVR utilized two main part, one in charge of communication, synchronization, and data versioning, the other handling error detection and failure recovery.

1. GVR Global-Array Interface: This interface manages the communication, synchronization, accumulate operations, and data access and versioning functions. User can allocate memory and create arrays, put to or get from the arrays, perform atomics, synchronize by fences and waits, and accumulate results. Versioning functions also provide labeling, and moving between different versions.

2. GVR Error-Handling Interface: GVR uses Open Resilience (OR) interface to detect and manage failures. User can create error descriptors which will be used to choose the proper error handler when an error occurs. GVR allows the user to create error handlers for a category of errors. These handlers are called by the GVR Error-Handling Interface in case an error is detected. These handlers can allow rollbacks to any of the stored versions of data. Detection is also done through the system or other possible application-based methods.
Multi-version checkpoints are most useful for recovering from silent errors[31]. Detection of such errors can take long enough to corrupt one or more versions of checkpoints, and therefore two or more versions of checkpoints are suggested for different systems.

4.4.2 Global Address Space Programming Interface (GASPI)

Global Address Space Programming Interface (GASPI), is a PGAS API [27]. Similar to OpenSHMEM, GASPI provides one-sided communication. However, unlike OpenSHMEM which uses symmetric distributed memory to provide such communication, GASPI uses configurable Remote Direct Memory Access (RDMA) pinned memory segments. A segment is a contiguous block of virtual memory which can exist on different levels of memory hierarchy or on different devices. Therefore, memory addresses in GASPI consist of process rank, segment identifier, and offset.

GASPI provides one-sided, passive and collective communications, as well as atomics, memory management functions and many more facilities. GASPI utilized queues for requests of asynchronous dataflow and provides the ability to monitor the completion of requests remotely[44].

GASPI’s API provides timeouts for blocking functions, to allow the user to choose the maximum time a function blocks. When the special value -1, or GASPI_BLOCK, is passed to a blocking function, it can block indefinitely. Value 0, or GASPI_TEST lets the procedure to partially finish: in this case, the process does not wait to receive data from other processes. If timeout is set to any amount more than zero, it shows the maximum time in milliseconds the function can wait [22].
4.4.3 MPI

Lots of studies have been done to make MPI more resilient or provide fault tolerant extensions for it. These works cover a wide range of fault tolerant techniques and possible errors in MPI. In this section we review some of works that have influenced this study.

4.4.3.1 User-Level Failure Mitigation (ULFM)

User-Level Failure Mitigation (ULFM) is a library built for MPI to make it more resilient. Normally when a failure happens in an MPI application, it exist the whole program, or causes deadlocks in case of blocking communication or synchronizations. Some error codes are defined in ULFM to help find and detect causes of errors and prevent deadlocks.

Another importance matter when a rank fails in an MPI program, is to shrink the communicators containing it and make new communicators [6]. An evaluation of ULFM is presented in [5].

4.4.3.2 Fenix

Fenix is an application-based CPR library built for MPI and upon ULFM. It leverages diskless checkpoints and coordinated rollbacks, and besides delivering mechanisms for checkpointing and restart, it provides some further support specially needed for MPI programs such as fixing communicators containing failed processes, and re-spawning new processes after failure[19].

The main functionalities of Fenix are initialization and finalization, memory allocation, checkpointing and restoration[21]:

- Initialization: All the ranks in a communicator call the initialization function collectively. They ask for a number of spare ranks, and a new communicator containing all ranks minus the asked spare ones is re-
turned. Calling ranks also specify if new spare ranks should be spawned in case all of them were previously revived to replace failed ranks.

- Memory allocation: The user can create and label a set of data using `Fenix_Data_group_create` function in Fenix. This function receives the id and communicator used for this data group as well as the number of versions that should be kept from the data in the data group. After creating a data group, data members can be added to it by `Fenix_Data_member_create` which received the data and member group id’s, as well as type and size of data which will be checkpointed by this member in future.

- Checkpointing: Each defined data member can then store versions of its data through `Fenix_Data_member_store` function call. It needs the group and member id’s together with a subset of data to checkpoint. Other versions of storing function, such as non-blocking ones, are also available.

- Restoration: Data is recovered with `Fenix_Data_member_restore` providing the group and member id’s with the timestamp specifying which version of data to recover, and returns a pointer to the recovered data. Restorations can also be done by asking restoration from a specific rank’s data. Recovery is called by all the ranks in the communicator.

- Finalization: This function cleans all stated used by Fenix. Fenix’ function calls will not work after this call.

Other routines are provided to query rank roles, redundancy policies, etc. Ranks can have one of three different roles, `FENIX_ROLE_INITIAL_RANK` which is a rank that started running at the beginning of program, `FENIX_ROLE_RECOVERED_RANK` that has replaced a failed rank, and `FENIX_ROLE_SURVIVOR_RANK` which has survived a rank. The difference between the first and third can help the programmer to perform the initialization of ranks at the appropriate ones. Some work has been done to leverage Fenix for checkpointing
Stencil-based\textsuperscript{4} applications but perform local recoveries\cite{20}. In this method, the recovery only takes place in re-spawned ranks (the ones that will replace the failed ones) as well as the ranks that communicated with failed ones. Local recovery improves the overhead caused by global recoveries.

### 4.4.3.3 Hybrid Checkpointing

A study combines both incremental and full-checkpoints to achieve better performance\cite{50}. Incremental checkpoints (checkpointing the portions of data that have changed since the last checkpoint) reduce the amount of time, memory, and I/O usage needed for checkpointing. However, when recovery is needed, incremental checkpoints must be merged with full-checkpoints to provide the latest state of system. This merging increases the time needed for recovery. But since checkpointing happens more often than recovery, reducing the time needed for it will compensate for slower recoveries, and with a good ratio of this two methods, system can perform well. The suggestion in this work is to fully checkpoint once out of ten times of checkpointing.

### 4.4.4 OpenSHMEM

The only two known work of resilience in OpenSHMEM before this work are studied in this section.

#### 4.4.4.1 Error Handlers for OpenSHMEM

Error handlers are a suggested extension to OpenSHMEM’s API, to provide detection and handling of user-specified error, propagate the knowledge of error, and to move on from errors to resume the application\cite{8}. The following is the main suggested extension:

\textsuperscript{4}Stencil-based applications consist of multiple iterations in which computation and sometimes communication takes place.
typedef void ( * shmem_errhandler_cb_fn ) ( int errcode , void * user_params ) ;

void shmem_errhandler_set ( int errcode ,
shmem_errhandler_cb_fn errh , void *user_params ) ;

void shmem_errhandler_get ( int errcode ,
shmem_errhandler_cb_fn errh , void *user_params ) ;

Through the above API, the user can define a desired error handling function for an error, and set error handlers based on that function and error. Then they can request the proper handler by having an error’s information.

Some pre-defined error handlers exist in the suggested API, including shmem_errhandler_gexit, shmem_errhandler_break, and shmem_errhandler_gbreak. The first terminates the application, the second ends the blocking of an operation, and the third ends all blocking functions.

4.4.4.2 Checkpoint/Restart (CPR) for OpenSHMEM

A CPR library in [28] implements a method to store checkpoints of all the symmetric memory in OpenSHMEM. It provides two different approach for checkpointing:

1. Single Backup: In this method, each PE stores a copy of its symmetric memory in the next PE. If there are \( N \) PEs, the next PE to \( p_i \) is \( p_i (i+1) \% N \). In this case if a PE fails, the checkpoint of its previous PE is lost and when restarting the PEs, one of them cannot reload its last state.

2. Dual Backup: There are two copies of PEs’ checkpoints in this mode: one in the PE itself and one in the next PE. This way, if a PE fails, the backup of its previous PE is not lost forever.

Checkpointing in this library is done collectively, meaning all the PEs perform it at the same time. User cannot choose which data to checkpoint and
the whole symmetric heap and global variables are copied in every checkpoint, therefore user has to put every variable they need to be checkpointed on the symmetric memory. Checkpointing is done through `shmem_checkpoint_all()` which handles both error detection and checkpointing. If a failure has happened since the last checkpoint, the function returns with failure status and no checkpointing is done. Otherwise the symmetric memory will be copied into appropriate places according to the mode.

Function `shmem_restart_pes()` is used for a coordinated rollback. It reloads the latest checkpoints into the symmetric memory and replace dead PEs with previously sleeping ones, turning them into working PEs. To distinguish original PEs (which have started from the beginning of program) with substitute ones, `shmem_ft_algo_init()` is used. This can be useful when an initialization is done on PEs, since substitute ones do not have to go through it again. Function `shmem_query_fault()` also provides the status of all PEs with reasons of failures for failed ones.

This work provides a valuable contribution to resilience for OpenSHMEM. However, it contains some flaws which can be resolved in future works: this method does not checkpoint private data, it checkpoints the whole symmetric memory which can cause unwanted memory usage and overhead, as well as low flexibility for users.

### 4.5 Possible Errors and Failures in OpenSHMEM

The main types of failures that may happen in an OpenSHMEM program are [8]:

1. Network Failures: Network problems and failures can lead to delays in sending and receiving messages, or making some processes non-responsive to a group of other processes.
2. Resource Exhaustion: The example of such failures are PEs asking for symmetric memory more than it is possible.

3. Crash Failures: When a group of PEs stop working or being responsive, they are crashed. Crashes can happen due to hardware problems (e.g. power outage), network failures, deadlocks, or even simple bit flips (setting a wrong signal.)

4. Corruption Failures: Some processes may start acting wrong or disruptive. It is usually difficult to track or detect this group of failures unless the failure follows some conditions such as dataset corruption.
Figure 4.1: A chart of different resilience methods, based on a more detailed work in [18]. Here we only discuss two relevant methods: redundancy and recovery. More details on other methods exist in [18].
Figure 4.2: The ring topology used in heartbeat failure detection mechanism in [7]. Every entity sends periodic heartbeats to its observer and monitors its predecessor.

Figure 4.3: An instance of CDs’ hierarchy. Errors are handled either within the same level or sent to upper levels for management.
Chapter 5

Resilience for OpenSHMEM

5.1 Checkpoint/Restart Extension for OpenSHMEM

As mentioned earlier, a checkpoint restart is one of the most widely used resilient methods in HPC. Some libraries provide checkpoint/restart functionalities within their API such as GVR, and some have extensions providing it for them, such as Fenix for MPI. The only attempt for resilience in OpenSHMEM before this work is also a checkpointing extension which we have reviewed in the previous section.

Due to its popularity, ease of use and lower overhead than some other resilience techniques, we have chosen CPR as the best start for making OpenSHMEM fault tolerant. We have implemented a CPR extension for OpenSHMEM which has similarities to some reviewed works such as Fenix or the previous CPR for OpenSHMEM [28]. This chapter includes the details of our model and its implementation.
5.2 Checkpoint/Restart Model

The implemented model is an effort to provide a CPR platform that is flexible and can answer different needs of users. The user can decide to store whichever piece of memory they require and checkpoint the data with a desired frequency. They are also able to choose the number of copies of the checkpoints to be stored.

Some processes are selected at the beginning of the CPR initialization as backup to replace non-responsive processes in case of errors or failures. The user can also specify the number of backup processes since they may be more familiar with the possibility of failures and their frequency on the system they are using, as well as the system resources that can be spared, or the urgency of having no stops or long delays in application’s execution. The different components of this model are explained further.

5.2.1 Overview of the Model

The processes are divided into different categories at the start of CPR part to take care of different roles in Checkpoint/Restart process. Some, called the active ones, run the main program and some, called storages, maintain checkpoints and handle the rollback in case a PE dies. All PEs are labeled accordingly and there are some lists to keep track of PEs’ types or roles.

To provide the appropriate amount of memory needed to keep checkpoints, processes need to announce the amount of memory they need for storage before they are allowed to start checkpointing their data. This process is called reservation. Storage PEs manage the reservation requests and allocate memory as needed.

After reservation, active PEs can checkpoint different variables and arrays on their memory with their required frequency. All the reservation and checkpoint request and data are transferred between active and storage PEs through carriers and queues. Further details of these structures are explained
in the coming sections.

During rollbacks, one storage PE will load the dead PE’s data it has kept into its memory and change its role from storage to active. They replace the dead PEs in this manner. The rest of this chapter addresses this model, its details and implementation.

5.2.2 Different Types and Roles of Processes

Before the checkpointing portion of a program starts, all PEs act the same, but after that PEs are divided into different categories and handle different procedures in the program. The categorization is done based on the checkpointing mode user chooses, as explained in Section 5.2.3, and divides PEs into different types with different roles.

5.2.2.1 Types of PEs

A PE can have different types during the program and act according to it. After initialization of the checkpoint/restart part, every PE starts with one type, and maintains it while the program runs error-free. Some PEs’ types will change when a failure happens. However, at any certain point in the program a PE has exactly one of the following five types:

- Original PE: When the checkpointing starts, an original PE is the one that starts running the main program. It can ask for checkpoints of its memory portions.

- Dead PE: A dead PE is one that has failed. It might have any other type prior to the failure. Dead PEs take no part in the main program or checkpointing. In the absence of a way to recover dead PEs if possible, they will stay dead until the end of the program. Such recovery mechanism would need the support of system or lower level process management libraries.
• Spare PE: The user asks for a certain number of spare PEs at the initialization of the program’s checkpointing part. These spare PEs take no part in running the main program, but are used for keeping copies of checkpoints or simply waiting for a failure to replace a dead PE. Based on the storage mode explained next, spare PEs can play different roles in checkpointing.

• Main Spare PE: A special kind of spare PEs are Main Spare PEs, or MSPEs. This type is only used in Two-Copy Mode explained later. They are in charge of storing the checkpoints, replacing a dead PE in case of failure, and making new copies of checkpoints if necessary. Their role is explained in details further in this section.

• Resurrected PE: Resurrected PEs are previously spare PEs or MSPEs that have replaced dead ones after a failure. After a PE changes to resurrected type, it acts similar to original PEs: they run the program and ask for their data being checkpointed.

The relation of different PE types through different stages of a program is shown in Figure 5.1.

5.2.2.2 Roles of PEs

In addition to their types, PEs can play different roles in the checkpoint/restart process. A PE, if not dead, can have one of the following roles:

• Active PE: All the PEs running the main program are called active PEs. These can be either the original or resurrected PEs.

• Storage PE: One copy of all the checkpointed data is kept in every storage PE. Depending on the checkpointing mode, either spare PEs or MSPEs are chosen as storage ones. When a failure happens, one or more of the storage PEs will replace one or more dead ones.
Dormant PE: Dormant PEs only exist in the Two-Copy Mode. They are a subset of spare PEs and are neither active nor storage. When a failure happens, one of the dormant PEs will turn into a storage one.

The different roles of PEs are depicted in Figure 5.2

Different types and roles of PEs enable the checkpointing system to properly manage PEs and provide flexibility according to the mode user chooses.

5.2.3 Two Modes of Storing Checkpoints

In order to make this platform flexible for different user needs, we have provided two checkpointing modes. They are mainly different in the number of copies of checkpoints stored on remote PEs. It should be noted that every active PE keeps one copy of its own checkpointed data in its memory. This will help to achieve faster global rollbacks, since there is no need to fetch the last checkpoint from remote memory for a PE that has survived failure. Figure 5.2.3 shows active PEs and their components in both of these modes. The other difference between these two modes is what happens after a failure and how the rollback is handled, which will be explained later.

Other than one local copy of checkpoints, there can be two or more remote copies of data on remote PEs. Based on the probability and frequency of failures, number of processes per node and in total, amount of memory to spare and the speed of rollback process, the user can decide the best option of the two for their application. The differences of these two modes in different scenarios are studied in Chapter 6.

5.2.3.1 Many-copy Mode

In this mode, every spare PE is a storage PE, meaning one copy of all the checkpointed data will be kept in every single spare PE. Since user chooses the number of spare PEs at the initialization of checkpoint region, this mode
can provide as many copies as user needs and system allows. Also, since every spare PE already has all the necessary data for recovery in them, in case more than one PE fail, they can replace the failed PEs almost immediately. The complete anatomy of this model is shown in Figure 5.2.3.1.

5.2.3.2 Two-copy Mode

We also provide another mode of checkpoint/restart which keeps only two remote copies of checkpointed data. Two of the spare PEs (of their number is more than or equal to two) will be chosen at the initialization time to continue as storage PEs, and the rest of them will stay dormant until a failure happens.

All the active PEs post their requests of checkpointing to the two storage PEs. If a failure happens in one of the active ones, one storage PE will replace the dead PE but the second one will choose a dormant PE (if any) and turns it into a storage one by copying all of the checkpointed data into its memory. This way it is ensured that at all times, at least one copy of the data is available. Figure 5.2.3.2 illustrates the components of this mode.

5.2.4 Provided Functions

Any checkpoint/restart provides two main functions: the part storing checkpoints of data or the state of system, and the part that recovers that checkpoint when an error is detected. As reviewed in Chapter 4, different methods exist to provide the checkpoint mechanism: they differ in the locale of storage, number of versions and copies kept, and the level providing it (from system to application). Rollbacks, or restarts, also vary in being global or partial. Similar to some existing methods in HPC, we have provided an application-based Checkpoint/Restart with in-memory storage in local and remote processes and global rollback. Our implementation consists of five main functionalities explained below.
5.2.4.1 Initialization

At any point after the initialization of OpenSHMEM library and before its finalization, the user can start the Checkpoint/Restart process. It is provided through \texttt{shmem\_cpr\_init()} function call. Through this the user declares the number of spare PEs they want and the appropriate checkpointing mode. The default checkpointing mode is \texttt{CPR\_NO\_CHECKPOINT} providing no checkpointing functionalities.

In this function the needed memories for checkpoint tables and shadow memories are allocated, reservation and checkpointing queues are created, and variable, PE types and roles are loaded with appropriate amounts. After this function call the user can request for reservations and checkpointing. The initialization function is a collective one and should be called by all PEs.

5.2.4.2 Reservation

In order to allocate the needed memory, the user has to declare the amount of memory they need before starting to store the content. This is done by \texttt{shmem\_cpr\_reserve()} function call. The user has to request for reservation and call this function for every variable they want to checkpoint in future. As arguments to this function, user should announce which PE is asking for reservation, provide a unique integer id associated with the variable they want to checkpoint, the address of that variable and its size.

This function can be called by at least one active PE and all the storage ones. The active ones send their request to reservation queues on all the storage PEs. Then storage PEs will read the information from their queues and allocate the appropriate amount of memory in their checkpointing table for storing the data. The requesting PE also allocates a place on its memory for this data, since all active PEs keep one copy of their own checkpoints locally. The local place of storage in an active PE’s own memory is called shadow memory.
5.2.4.3 Checkpointing

This function’s performance is very similar to the reservation function. Through the `shmem_cpr_checkpoint()` function, user sends the information of a previously reserved variable along with the data stored in it. The arguments are the number of PE asking for checkpoint, together with id, address and size of data.

Similar to reservation, this function should be called by all the storage PEs and at least one active PE. An active PE will send the information and data to the storage PEs’ checkpoint queues. Then it would update the corresponding place on its shadow memory with the new data. When storage PEs receive the information of a checkpoint request, they look for the location on their memory corresponding to that id and PE number, and then update the data.

5.2.4.4 Rollback

Rollback function or `shmem_cpr_rollback()` is called when a failure is detected which had lead to disfunction of one or more PEs. As the rollback in our model is global, this function is a collective one and must be called from all PEs. The information it receives is the number of PE calling it and the number and list of dead PEs.

We first should check the last role of the dead PE. If it was a dormant PE, no action and rollback is needed. If the role is storage, depending on the mode one of the following can happen: if the application is in many-copy mode, no action is needed. However, in two-copy mode, if one of the storage PEs fails, the other one should copy all of its stored checkpoints to a dormant PE, if any, and make another storage.

In case an active PE dies, a storage one replaces it. In order to change a PE from storage to active, we first change its type from MSPE or spare to resurrected, and its role from storage to active. Then we create the shadow memory inside it and fill it with the checkpoints of the dead PE. At this
point all active PEs will reload the last checkpoint from their shadow memory into the corresponding variables. We finally update the list of PEs and their numbers on all the PEs. When the mode is set to two-copy checkpoint, we will also pick a dormant PE (if any) and turn it into storage.

5.2.4.5 Finalization

Before finalizing the OpenSHMEM, the checkpoint/restart library should be finalized through `shmem_cpr_finalize()`. This will clear all the types and roles and checkpoint associated variables, free the allocated memory and prepare PEs for continuing the normal flow of the program. After this point, calling any checkpoint/restart function, except a new initialization, will result in errors or unknown behavior.

5.3 Implementation

The CPR system in this work has been implemented using various labels and data structures, OpenSHMEM functions, and methods. This section reviews these components in detail.

5.3.1 Labels, Lists and Queries

Each PE keeps two labels regarding their type and role. To keep track of other PEs and their roles, PEs also keep an array of other PEs’ types and roles. Three variables `cpr_num_active_pes`, `cpr_num_storage_pes`, and `cpr_num_spare_pes` hold the number of active, storage, and spare PEs respectively. It should be noted that in case of failures, the numbers above might decrease during a program’s execution.

It should be noted that if a PE $p_i$ fails and $p_j$ replaces it, all the communications, AMOs, and other functions that target $p_i$ should address $p_j$ instead. To facilitate such changes in PE numbers for users, and since they
cannot know which PE has replaced a dead one, we have provided a function
to return the original PE numbers. Similar to OpenSHMEM’s query func-
tion `shmem_my_pe()` which return a PE’s own number, `shmem_cpr_replaced_ pe_num (int pe_num)` returns the number of PE that has replaced $PE_{pe.num}$. For example, until PE $p_i$ works flawlessly, `shmem_cpr_replaced_pe_num(i)` will return $i$, but if $p_i$ fails and $p_j$ replaces it, the function above will return $j$. Therefore, all communication functions using PE number $i$ as an input should use `shmem_cpr_replaced_pe_num(i)` instead, e.g. an `shmem_put` will change to format `shmem_put(TYPE *dest, const TYPE *source, size_t nelems, shmem_cpr_replaced_pe_num(int pe))`.

On the other hand it is sometimes necessary to know which crashed PE a resurrected one has replaced. For example, when $p_j$ replaces $p_i$ af-
ner it crashes, $p_j$ is now in charge of all the checkpoints and reservations
made by $p_i$, therefore in reservation and checkpoint function calls it should
still use the number associated with $p_i$. `shmem_cpr_pe_num(int pe_num)` returns the number of a PE that used to live and work as $PE_{pe.num}$ does now.
It is up to user to determine when `shmem_cpr_replaced_pe_num(int pe),
shmem_cpr_pe_num(int pe_num)`, or the actual PE number should be used.

PEs keep an array of storage PEs to access them when needed. The check-
pointing mode is stored in `cpr_checkpointing_mode` and can hold one of the values `CPR_NO_CHECKPOINT`, `CPR_MANY_COPY_CHECKPOINT`, and `CPR_TWO_COPY_CHECKPOINT`. The `CPR_NO_CHECKPOINT` mode is usually assigned when the CPR cannot be initialized for any reason.

The types of PEs are `CPR_ORIGINAL_PE`, `CPR_SPARE_PE`, `CPR_MSPE`, `CPR_RESURRECTED_PE`, and `CPR_DEAD_PE`. The roles provided are also `CPR_ACTIVE_ROLE`, `CPR_STORAGE_ROLE`, and `CPR_DORMANT_ROLE`. The role and type of any PE can be queried using `shmem_cpr_get_pe_role (int pe_num)` and `shmem_cpr_get_pe_type (int pe_num)` respectively.

An array containing numbers of all storage PEs is returned by `shmem_cpr_get_storage_pes()`. User can also query the number of different kinds of PEs
by using `shmem_cpr_get_num_active_pes()`, `shmem_cpr_get_num_storage_pes()`, and `shmem_cpr_get_num_spare_pes()`, for active, storage, and spare PEs respectively. In two-copy mode, the total number of PEs running the program is calculated by the following formula:

\[
N_t = N_{st} + N_{sp} + N_a
\]

Where \(N_{st}\) is the number of storage PEs and \(N_{sp}\) is the number of spare PEs. For many-copy mode, the formula turns into:

\[
N_t = N_{sp} + N_a
\]

and \(N_{sp} = N_{st}\) in many-copy mode.

### 5.3.2 Carriers

Carriers hold and convey the reservation or checkpointing requests and data. Based on the type of the request they carry, carriers have two different kinds:

1. Reservation Carriers: They are in charge of keeping information about reservation requests. The information consists of the PE making the request, the size of memory needed to checkpoint the related data, address to data, an integer id associated with it, and an indicator to whether the data is on symmetric memory or not. Reservation carriers are filled by active PEs during reservation function call and are put to reservation queues on storage PEs.

2. Checkpoint Carriers: Checkpoint carriers are very similar to reservation ones except they carry a copy of the data that will be stored. They are created and filled in checkpoint functions by active PEs and sent to checkpoint queues on storage PEs. Checkpoint carriers are made to transfer a maximum amount of data at most. If the size of requested data
for checkpoint exceeds this limit, active PEs will send multiple carriers. The variable offset in checkpoint carriers is used for this matter, e.g. if the maximum capacity of a checkpoint carrier to transfer data is $N_{max}$, to checkpoint a data of size $3N_{max} + 5$ we have to send 4 carriers. The offsets of these 4 carriers will be $N_{max}$, $2N_{max}$, $3N_{max}$, and $4N_{max}$ respectively.

Carriers are shown in Figure 5.2.3.

### 5.3.3 Checkpoint Table and Shadow Memory

Checkpoint table and shadow memory are the places where we store copies of checkpoints. The difference is that shadow memory exists on an active PE and keeps the copy of local data, while checkpoint tables are on storage PEs and store the copy of all active PEs’ checkpoints.

Shadow memories start with a size of 16, meaning they can hold the information of 16 different variables and their checkpoints. However, if during the reservation process, an active PE asks for more than 16 variables to reserve, the shadow memory’s size will double. The doubling will repeat each time the number of variables exceeds the size of the shadow memory. As previously explained in 5.3.2, a data may need multiple carriers to transfer it, which is why each element of the shadow memory is an array of checkpoint carriers. The size of each array is determined during reservation using the following formula:

$$\text{shadow\_memory[i].length} = \left\lceil \frac{N}{N_{max}} \right\rceil$$

where an active PE asks for storage of $N$ bytes for variable $v_i$ while each carrier conveys $N_{max}$ bytes at most.

Checkpoint tables have a row to keep checkpoints of every active PE. Therefore, if there are $N_a$ active PEs, a checkpoint table on a storage PE will have $N_a$ rows: on the $i^{th}$ row there will be a copy of $p_i$’s shadow memory.
Similar to shadow memories, the size of each row starts at 16 and doubles if needed. The \( j^{th} \) element of \( i^{th} \) represents and stores the checkpoints of variable \( v_j \) in PE \( pe_i \). Therefore:

\[
\text{checkpoint\_table}[i][j].length = \text{shadow\_memory}[i][j].length
\]

Since every symmetric variable has a copy on all PEs, putting the checkpoint table and shadow memory on symmetric memory will lead to unwanted memory allocation and overhead. Therefore checkpoint table and shadow memory are both placed on the private parts of memories. But communication in OpenSHMEM happens through symmetric memory and for transferring data and checkpoints, we need to either send them to a part of symmetric memory, or read them from symmetric memory. Since we want to enable checkpointing private variables, and checkpoint table and shadow memory are also on private memory, the data transfer is done through a medium that exists on symmetric memory. Queues, as studied in next part, are used for this purpose.

### 5.3.4 Queues

Queues are part of symmetric memory used for transferring the information and data for reservation or checkpointing. While a checkpoint table or shadow memory can keep tens of thousands of checkpoints, queues only keep a few hundred at a time and therefore are a better choice to be placed on symmetric memory than they are.

There are two types of queues: one used for reservation and made up of reservation carriers, the other containing checkpoint carriers to hold checkpoints' information and data. Since symmetric variables exist on every PE, each PE regardless of their type has one reservation and one checkpoint queue but we will only use the ones on storage PEs.
5.3.4.1 Reservation Queues

As previously mentioned, an active PE sends its request of reserving a place for a variable to storage PEs. In fact, they create one reserve carrier, fill it with appropriate information, and put it to every storage PE’s reservation queue using OpenSHMEM’s `shmem_putmem()` function call. To avoid races when putting carriers on remote queues, every active PE will fetch and increment the tail of a storage PE’s queue atomically using `shmem_atomic_fetch_inc()`, and then put the carrier at the tail of queue. This way we make sure that when different active PEs put their carriers in the same queue, their data transfer will not interfere with each other.

Every time a reservation function is called, the calling active PEs start putting carriers to queues almost at the same time as storage PEs start reading from them. To ensure storage PEs will not find the queue empty and return from the function before active PEs have even started, each queue has a signal associated with it. When a queue is empty, its owner will reset the signal to zero. When an active PE wants to send data to a queue, if its signal is zero, the PE will set it to one. Therefore, when a storage PE enters the reservation function, it will wait until its reservation queue’s signal is set to one. This is done through `shmem_wait_until()` function.

The method mentioned above only ensures that every time a storage PE enters the reservation function, it will read at least one carrier from its queue. However, it is not sufficient to force the PE to read all carriers in its queue. That is, if in one function call \( k > 1 \) carriers are put to a reservation queue, its owner may not read all of them: a storage PE will probably return from the function reading any number of carriers from one to \( k \). This happens because there is no way to predict when or how much of data is sent or received using current OpenSHMEM’s put and get functions.

Whenever a storage PE calls the reservation function, it can check carriers left unread from the previous function calls. Therefore, the user can call reservation only by storage PEs to make sure all the reservation requests up
to that point in program will be taken care of. However, to make sure no reservation queue holds unread reservation carriers after the last reservation function call, when a storage PE enters a checkpoint function, it will first check its reservation queue for any unread requests, handle them if any, and then continues to reading its checkpoint queue. This method ensures the reservation is done completely even if the user does not re-call reservation at any point in their program.

Yet there is another problem in reading the carriers from queues. A storage PE waits until its queue’s signal is set, meaning at least one new carrier has arrived, then it will start reading from the queue. Between the time the signal is set by a remote active PE, and when PE starts to read the queue, the data may not have arrived yet, which causes the storage PE to read old or wrong data from queue.

To resolve the problem mentioned above, each element in the queue has a signal associated with it. When an active PE puts a carrier in the $i^{th}$ index of the queue, it will call a `shmem_fence()` to assure the transfer is done, and then sets the $i^{th}$ signal to one, using `shmem_atomic_set()` at the target storage PE. On the other side, when the storage PE wants to read a carrier at the head of its queue, it waits until the signal associated with that element turns to one using `shmem_wait_until()`. Since the active PE had used a fence between sending the carrier and setting the signal, when the storage PE is finished waiting for the signal, the data has definitely already arrived at the head of the queue. At this point, it is safe to read the carrier and handle the reservation request in it. The storage PE then immediately resets the signal of that element in queue, to prepare it for the next time a carrier is received at that place.

5.3.4.2 Checkpoint Queues

A checkpoint queue is very similar to the reservation one, except it is made of checkpoint carriers. It is filled when checkpoint function is called, through
shmem_putmem() and its tail is increased with shmem_atomic_fetch_inc() by active PEs. It also has a signal which shows whether it is empty, and a storage PE will wait until said signal is set to one before reading its queue. Also, to ensure the arrival of carriers at any point of the queue, each element has a signal associated with it, which is implemented and used similar to those of reservation queues.

Same as reservation queues, we cannot make sure if by one function call all the checkpoints are read from a checkpoint queue. Which is why when a rollback is called, storage PEs will first check their checkpoint queues for any unread data to make sure their version of any checkpoint is the latest. Again, similar to reservation process, the user can also call the checkpoint function at any point of the application to ensure all checkpoints are up-to-date, up to that point in the program.

Checkpoint queues also play another role: when in two-copy mode we need to create another storage PE, a currently storage PE should pick a spare one and turn it to an MSPE (hence, a storage one). The checkpoint table is created on the target PE and then the other storage will start sending checkpoint carriers into the new storage PE’s queues. The rest is similar to a normal checkpointing procedure.

### 5.4 Usage and Example

Below, we will show an example code using the CPR system provided:

```c
int iter, progress; // To keep track of program's progress

int main(int argc, char const *argv[]) {
    int npes, spes, me, num_iter, frequency;
    int success_init, array_size;
    unsigned long* a;
    shmem_init();
```
me = shmem_my_pe();
npes = shmem_n_pes();
spes = 4;
array_size = 100;
frequency = 100;
num_iter = 10000;

a = (int *) shmem_malloc( 100 * sizeof(unsigned long));
shmem_barrier_all();

success_init = shmem_cpr_init(me, npes, spes, CPR_MANY_COPY_CHECKPOINT);

if ( success_init )
{
    // initialization of variables
    initialize (a);

    // Reservation requests
    shmem_cpr_reserve(0, &iter, 1 * sizeof(int) ,
                     shmem_cpr_pe_num(me));
    shmem_cpr_reserve(1, a, array_size * sizeof(unsigned long) ,
                     shmem_cpr_pe_num(me));
    shmem_cpr_reserve(2, &progress, 1 * sizeof(int) ,
                     shmem_cpr_pe_num(me));
    shmem_barrier_all();

    // Optional: making sure all reservation requests are read
    if ( cpr_pe_role == CPR_STORAGE_ROLE )
        shmem_cpr_reserve(0, NULL, 0, shmem_cpr_pe_num(me));

    progress = 1;
    shmem_cpr_checkpoint(2, &progress, 1 * sizeof(int) ,
                          shmem_cpr_pe_num(me));

    for ( (*iter) = 0; (*iter) < num_iter; ++(*iter) )
    {
        if ( cpr_pe_role == CPR_ACTIVE_ROLE )
            // Update a variable
    }
// Do calculation and communication

if ( (*iter) % frequency == 0 )
{
    shmem_cpr_checkpoint(0, &iter, 1 * sizeof(int) ,
    shmem_cpr_pe_num(me));
    shmem_cpr_checkpoint(1, a, array_size * sizeof(unsigned long) , shmem_cpr_pe_num(me));
    shmem_barrier_all();

    // Optional: making sure all checkpoint requests are read
    if ( cpr_pe_role == CPR_STORAGE_ROLE )
        shmem_cpr_checkpoint(0, NULL, 0 ,
    shmem_cpr_pe_num(me));
}

if ( failure_happened ){
    shmem_cpr_rollback(dead_pe, shmem_cpr_pe_num(me));
    shmem_barrier_all();
}

// Second part of calculation

progress = 2;
shmem_cpr_checkpoint(2, &progress, 1 * sizeof(int) ,
    shmem_cpr_pe_num(me));
if ( cpr_pe_role == CPR_ACTIVE_ROLE )
    // Do calculation and communication

    shmem_cpr_checkpoint(1, a, array_size * sizeof(unsigned long) , shmem_cpr_pe_num(me));
    shmem_barrier_all();

    // Optional: making sure all checkpoint requests are read
    if ( cpr_pe_role == CPR_STORAGE_ROLE )
        shmem_cpr_checkpoint(0, NULL, 0 , shmem_cpr_pe_num(me));
}
if ( failure_happened ){
    shmem_cpr.rollback(dead_pe, shmem_cpr_pe_num(me));
    shmem_barrier_all();
}

    shmem_cpr_finalize();
}

else
{
    // Normal program without checkpointing
}

shmem_finalize();
return 0;

In line 20, the CPR library is initiated. It asks of \texttt{spes} spare PEs out of \texttt{npes} ones, and chooses a checkpointing mode, here \texttt{CPR\_MANY\_COPY\_CHECKPOINT}. If the request to initialization can be processed it return success otherwise failure, e.g. when \texttt{spes} is greater than or equal to \texttt{npes}, or an undefined mode is asked, initialization fails and mode is set to \texttt{CPR\_NO\_CHECKPOINT}.

In lines 28-30, the user asks for reservation of different variables. While user can ask for reservation and checkpoints of both private and symmetric data, since access to restored symmetric variables is easier, all the variables here are chosen to be symmetric. Variables \texttt{iter} and \texttt{progress} can be checkpointed to retrieve the progress of application after a failure. For example, in this code if a failure happens at iteration \texttt{iter = 5*frequency+10}, after restoration the loop continues at \texttt{iter = 5*frequency} if \texttt{frequency < 10}.

User can checkpoint their data after their calculations and at the desired frequency. To ensure the inspection of all reservation or checkpoint requests, the user can call those functions on PEs with \texttt{CPR\_STORAGE\_ROLE}. This is of course optional and we have explained how the library makes sure all the
requests are read.

This CPR implementation currently does not provide any failure detection methods. If failures on some PEs is detected in any way, the rollback function can be called. This function must be called collectively similar to initialization and finalization, while reservation and checkpointing ones can be called on a subset of PEs. The user only needs to make sure at each call, all storage PEs must be involved. At the end of checkpointing region, the library can be finalized.

As mentioned, \texttt{shmem\_cpr\_pe\_num(int pe)} will return the PE number used throughout checkpointing, e.g. if PE 3 fails and 11 replaces it, \texttt{shmem\_cpr\_pe\_num(11)} returns 3. Since \texttt{p11} is handling the previous checkpoints of \texttt{p3} after a restart, then \texttt{p3}'s number should be used when \texttt{p11} calls the reservation, checkpoints, and rollback.

In another note, in the above example, assume that \texttt{p2} decides to send some data to its next PE which is \texttt{p3}. Since \texttt{p3} has failed, the data should be sent to \texttt{p11}, therefore \texttt{pe2} can use \texttt{shmem\_cpr\_replaced\_pe\_num((me+1)\%npes)} as the number of target PE. If user is sure that \texttt{p2} has not failed, they can use \texttt{shmem\_cpr\_pe\_num(me)}, \texttt{shmem\_cpr\_replaced\_pe\_num(me)} and \texttt{me} interchangeably.
Figure 5.1: The different types of PEs in both of the checkpointing modes and their relations. A PE’s type can change when a failure is detected. Recovering a dead PE often needs system support and is not included in this work.
Figure 5.2: The different roles of PEs in both of the checkpointing modes. The transition between different roles is described here. All PEs maintain their roles if no failure happens.
Active PEs are the same in both modes. They keep one local copy of checkpoints of their own data, called shadow memory. They fill their requests for reservation and checkpoint in carriers and send them to storage PEs. Active PEs are either of type original or resurrected. In this picture, there are $k$ active PEs. Since reservation and checkpoint queues are part of symmetric memory, they also exist on active PEs but are not used.
Figure 5.4: Spare PEs in many-copy mode all have storage role. Queues are symmetric variables and exist on every PE but they are only used if they belong to a storage one. Queues, checkpoint table and shadow memory, and carriers are explained in detail in Section 5.3.
Figure 5.5: Spare PEs in two-copy mode are either storage or dormant PEs. More on queues, checkpoint table and shadow memory, and carriers is studied in detail in Section 5.3.
Chapter 6

Results

6.1 Test Framework

The tests in this section were performed on Seawulf cluster, on compute nodes with two Intel Xeon E5-2690v3 CPUs each. More information and description of the cluster can be found in [43]. The programs have used OpenMPI 4.0.1’s implementation of OpenSHMEM and gcc 9.1.0 compiler. All the programs follow c11 standard.

6.2 Results

For testing the models, we have used a matrix multiplication program using OpenSHMEM. The matrices A and B are $N \times N$ and each PE keeps a portion of both of them with size $N \times N/n_p$, where $n_p$ is the number of PEs. To calculate $A \times B$ each PE calculates the multiplication of portions they have, and they shift their portion of matrix B to another PE. Therefore, after transferring a portion of matrix B for $n_p$ times, a PE has $1/n_p$ of the final result. The partial results can be gathered by one PE to be returned as the final result. Below the algorithm is shown:
```plaintext
me = shmem_my_pe()
npes = shmem_n_pes()
N_s = N / npes

Initialize(A_s, B_s, C_s, B_next)
// A_s: N_s * N matrix
// B_s: N * N_s matrix
// C_s: A_s * B_s
// B_next = B_s stored in the next PE

for i = 0 to npes
  matrix_mult(A_s, B_s, C_s)
  // get the B_s from next PE
  shmem_get(B_s, B_next, N * N_s, (me+1)%npes)
  swap(B_s, B_next)

if me == 0 // Collecting all the results in C
  C[0..N_s] = C_s
  for i = 1 to npes
    shmem_get(C[i * N_s .. (i+1) * N_s], C_s, N * N_s, i)
```

As this algorithm only has a loop with \( n_p \) iterations, and we needed a higher number of iterations to insert checkpoints with different frequencies, in three tests we changed the number of iterations to 50k, 100k, and 200k.

While the location of checkpoints, redundancy and many other technical details play an important role in a checkpointing mechanism’s performance, here we also have studied the importance of checkpoint insertion intervals here. Calculating the optimal checkpointing interval in global checkpoint systems has been discussed in [36, 45]. Here We have tried to study the effect of different frequencies of checkpoint insertions in our application-based checkpointing system.

Aside from the effect of checkpoint frequency, we have tried to compare our two models with a the above matrix multiplication program that does not use checkpointing. We have tested each model in three cases of no rollbacks.
happening, and one or two rollbacks happening. Therefore, we will be able to compare the effect of checkpoints and rollbacks in the two models.

In all of the tests below, we have used 4 spare PEs for both models. The active PEs were either 4 or 8 ones. Due to similarity, we only report the results of programs running for 200k iterations. In the above matrix multiplication, we have inserted checkpoints for C_s array during intervals of calculation. The execution times reported below, are calculated from the point of initialization of checkpoint mechanism, until its finalization. All timings reported here are the average of multiple runs. The matrix size for these test was set to $64 \times 64$, meaning each checkpoint is requested for 4 to 8 kB of data.

We can see in Table 6.2 that inserting checkpoints has little overhead in time: in case of 8 PEs, inserting 500 and 200 checkpoints increases execution time by 40.91% and 37.58% respectively, using many-copy model. These overheads reduce to 38.26% and 37.27% for two-copy mode. As expected, since in two-copy mode fewer communications take place, it performs better than many-copy mode. Also by comparison of cases with different number of checkpoints, it can be concluded that each checkpoint operation with data of size $4\text{k}^-8\text{k}$ takes around $0.1^-0.3$ milliseconds.

<table>
<thead>
<tr>
<th>Number of PEs</th>
<th>No Checkpoint</th>
<th>Many-Copy f=0.0025</th>
<th>Many-Copy f=0.001</th>
<th>Two-Copy f=0.0025</th>
<th>Two-Copy f=0.001</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12.33625</td>
<td>11.8125</td>
<td>11.6975</td>
<td>11.725</td>
<td>11.63</td>
</tr>
<tr>
<td>8</td>
<td>3.3</td>
<td>4.65</td>
<td>4.54</td>
<td>4.4525</td>
<td>4.39</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of matrix multiplication program with no checkpoints, and checkpoints inserted every 400 or 1000 iterations, for both many-copy and two-copy models.

As can be inferred from Figure 6.1, number (or interval) of checkpoints affect the execution time. It can also be seen that the two-copy model scales better as number of checkpoints go higher, since each checkpointing operation takes less time in this method.
Figure 6.1: Different frequency intervals used for checkpointing in both models.

The overhead caused by rollbacks in many-copy mode in depicted in Figure 6.2.

In the final results, we present the comparison between the two models, many-copy and two-copy, when no rollback happened and when one rollback happened, at different frequencies of checkpointing.

<table>
<thead>
<tr>
<th>Frequency of Checkpointing</th>
<th>No Checkpoint</th>
<th>Many-Copy 0 Rollback</th>
<th>Many-Copy 1 Rollback</th>
<th>Two-Copy 0 Rollback</th>
<th>Two-Copy 1 Rollback</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0025</td>
<td>3.3</td>
<td>4.65</td>
<td>4.8425</td>
<td>4.4525</td>
<td>4.48</td>
</tr>
<tr>
<td>0.001</td>
<td>3.3</td>
<td>4.54</td>
<td>4.6325</td>
<td>4.39</td>
<td>4.377</td>
</tr>
</tbody>
</table>

Table 6.2: Comparison of matrix multiplication program with no checkpoints, and checkpoints inserted every 400 or 1000 iterations, with or without rollback, for both many-copy and two-copy models. The number of active PEs is all cases is 8.
Figure 6.2: Overhead of one or two rollbacks compared with a case with no rollbacks.
Chapter 7

Future Work

This work is a starting point for fault tolerance in OpenSHMEM and can be extended or improved in various ways. Other methods and facilities can also bring more resilience to this library. In this chapter we propose some future possibilities and directions to make OpenSHMEM resilient.

7.1 Improving Checkpoint/Restart Extension

The CPR extension in this study provides partial full-checkpoints that are kept in memory. The rollbacks are coordinated and global, meaning every process recovers the last checkpoint and restarts from that point.

To improve this CPR extension, we can store some or all of the checkpoints on more permanent memory, e.g. on files. A good approach can be to write the checkpoints to files once in a while to balance the I/O overhead and accelerate the process, as in many two-level CPRs.

As mentioned in [50], a combination of incremental and full-checkpoints can also improve the speed and performance. In many incremental checkpoints, hashing functions are used to check if parts of data have changed since the last checkpoint. Again, full and incremental checkpoints can be done with
different frequencies to balance the overhead of one at the time of checkpoint and the other at the time of rollback.

Uncoordinated restarts can also be considered as a potential feature of the CPR extension. Not all the processes might need to rollback to a previous state depending on the type of failure and the range of its effect. However, messages and data transfers between processes in parallel programs play an important role in uncoordinated rollbacks, since if a data is sent between processes in different states or at a wrong time, it might cause errors in the final results. This is why uncoordinated rollback is often accompanied with a message logging system, which on its own decreases the performance and causes network or I/O overheads.

The work presented in Chapter 5 also consumes a lot of memory, especially in many-copy model. Instead of storing all of checkpoints in one PE, we can make in more distributed between spare PEs.

7.2 Compiler Contributions

Compilers can assist resilient methods achieve better performance or increase their ease of use. A study has been done to use compiler directives to leverage different CPR implementations and libraries in [32]. This system provides a unified form of using checkpoints and is easier to use for programmers.

Another contribution of compilers can be done by analyzing programs in advance and providing suggestions such as the best resilience method for an application (in case multiple methods are supported by the system). Compilers can also track some of the memory changes at compile time and advise the user to insert checkpoints in certain places in an application.

Another system called ComPiler for Portable Checkpointing (CPPC) provides a portable checkpointing tool for parallel programs [41]. CPPC consists of two main part, a compiler, and a checkpointing library. The compiler assists with checkpoint insertions, and in order to do that, analyzes the code to find
regions whose executions take a long time.

7.3 More Resilient API

OpenSHMEM’s API is not currently fault tolerant. Some efforts and suggestions are made towards creating a more resilient library through changes in its API. One instance is to make OpenSHMEM’s functions return a status containing the information on success or failure or that procedure. Providing this faces many challenges, e.g. it is difficult to check if a one-sided communication has finished successfully. Another method is to put timeouts on blocking functions. In some cases this reduces the run time, deadlocks, and can help identify the dead processes.
Chapter 8

Conclusion

In this thesis we have briefly introduced some parallel programming models such as message passing and PGAS. Our work has been done for OpenSHMEM library which is a PGAS API and implementation. We also have named different capabilities of OpenSHMEM and shown its performance and advantages towards MPI.

Faults and how to tolerate them, as well as methods to recover from them, are described in Chapter 4. The most popular fault tolerance and recovery method, Checkpoint/ Restart (CPR) is introduced and different methods to implement it are studied.

Because of its importance, ease of use, and flexibility, we have implemented a CPR mechanism for OpenSHMEM. This system is an application-based, in-memory checkpoint preserve method, with multiple copies of data due to the user’s request. Checkpoints can be applied to any data and with whatever frequency the user requires.

In chapters 6 and 7 we have presented the results of running this system and presented ideas to improve it.
Bibliography


