A CHECKPOINTING RESTART APPROACH FOR OPENSHMEM FAULT TOLERANCE

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Pengfei Hao

May 2016

A CHECKPOINTING RESTART APPROACH FOR OPENSHMEM FAULT TOLERANCE

Pengfei Hao

APPROVED:

Barbara Chapman
Dept. of Computer Science

Pavel Shamis
ARM Holdings

Edgar Gabriel
Dept. of Computer Science

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to thank my advisor Dr. Barbara Chapman, this thesis cannot be finished without her support and patient guidance.

I would like to thank Tony Curtis for the guidance provided on the OpenSHMEM project, he is the people lead me to OpenSHMEM project. Also thank Tony for the great system administration work to provide stable machine environment which supported the work in this thesis.

I would like to thank Pavel Shamis for his guidance on the fault tolerance topic, he contributed a lot of good ideas and advices to the prototype and implementation. Also thank him for joining my thesis commitee.

I would like to thank Dr. Edgar Gabriel for joining my thesis commitee. I took several classes including Parallel Computing of Dr. Gabriel. I benefit a lot from his clearly expression and presentation on HPC and Computer Architecture.

I would like to thank Oscar Hernandez and Manjunath Gorentla Venkata for their help during my intenship time in Oak Ridge National Laboratory.

I would like to thank Swaroop Pophale and Aaron Welch for their help on the original fault tolerance paper.

I would like to thank Steve Poole for his close attension of my fault tolerance work. I would like to thank Dounia Khaldi, Deepak Eachempati, Siddhartha Jana and Shiyao Ge for their help and advices on this thesis.

I would like to thank the resources and supports provided by United States Department of Defense and Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

A CHECKPOINTING RESTART APPROACH FOR OPENSHMEM FAULT TOLERANCE

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Pengfei Hao

May 2016

Abstract

The Partitioned Global Address Space (PGAS) has emerged recently for parallel programming at large scale. The PGAS ecosystem contains libraries, and languages (often implemented atop those libraries). One such library is OpenSHMEM, which offers an intuitive and easy-to-use API. OpenSHMEM's main feature is one-sided communication: in which communication and computation can be overlapped easily.

Performing computational science at large scale requires a resilient computing environment. Current computer systems, although generally reliable, do suffer from occasional faults. As the size of leadership high performance computing systems trends towards Exascale, the presence of faults will lead to system failures that cause fatal software failures. To mitigate against this problem requires software resilience, or "fault tolerance". One common approach is to checkpoint and restart from a known good state when an error is detected.

A long-running (e.g., weeks or months) program without fault tolerance will suffer from failure-restart cycles, which introduces unacceptably lengthy, uncertain execution times, and hugely increased resource usage.

In this thesis work, we explore a fault tolerance scheme based on check-point and restart that is specialized for the needs of PGAS programming models, using OpenSHMEM as a concrete implementation.

Using a 1-D Jacobi code, we show that this kind of approach is scalable and can save considerable resource usage. Ideas for more general solutions and other approaches are presented as future work.

Contents

List of Figures

Chapter 1

Introduction

As OpenSHMEM library evolves and accommodates features for petascale and exascale applications, it is imperative that more attention is given to fault tolerance and recovery. With long-running applications, it is prohibitive to re-run the applications (or execute identical instance in parallel) considering the combination of time requirements along with the increasing power costs.

1.1 Motivation

Latest high-performance computing (HPC) systems consist of an increasing number of computing resources, not only CPU cores but also hybrid architectures with GPUs and FPGAs. A large spectrum of HPC programs are targeted towards high volume data processing with run times extending to multiple days. Such applications require highly reliable systems with low failure rates. Achieving continuous

running time without failures is nearly impossible on such large-scale systems due to the combination of failure possibilities across computing resources. Unfortunately, the *Mean Time To Failure* (MTTF) of individual components of the HPC system is not expected to increase. We base this on the fact that the complexity, density, and sensitivity of individual components has increased over time. The hardware vendors increase the density of silicon chips and the number of transistors, while constantly trying to decrease power consumption of the components. This leads to a lower MTTF of system components. According to [14], the reliability of components in HPC systems has not improved in the past ten years. Since the reliability of individual components is not expected to increase while the number of components grow, the overall MTTF of the whole system is projected to decrease.

1.2 Contribution

The OpenSHMEM specification[19] currently lacks fault mitigation features, making OpenSHMEM programs vulnerable to system failures. In this thesis we present a working model of fault tolerance for OpenSHMEM[33]. This model is based on User Level Fault Mitigation (ULFM) scheme where an explicit API has to be introduced by an application developer to enable checkpointing. This provides flexibility to the application developer to modulate the frequency and placement within the application where the checkpoint may be introduced.

1.3 Thesis Organization

The thesis is organized as follows: Chapter 2 gives a brief background on the fault tolerance techniques used in HPC and the existing check-pointing approaches. Chapter 3 explains memory model and the failure types our work model will deal with, Chapter 4 describes the implemenation of fault tolerance on OpenSHMEM. Chapter 5 uses benchmark testings to illustrates the performance of the implementation. Chapter 6 gives the conclusion and discussion of potential future research directions.

Chapter 2

PGAS and OpenSHMEM

2.1 Partitioned Global Address Space (PGAS)

PGAS is a parallel programming mode. It assumes different processes hold piece of the memory in local which also accessible by other processes. By performing computation and data processing locally and exchange data between processes based on affinity information, PGAS model can provide performance benefits on multi-machine and multi-core systems. PGAS has gained popularity in recent years. There are different languages and libraries belongs to PGAS world.

Two main features separate PGAS from other programming models, process unit and one-sided communication.

- **Process unit**

  PGAS programs has logical computing resource and memory space bundle unit.

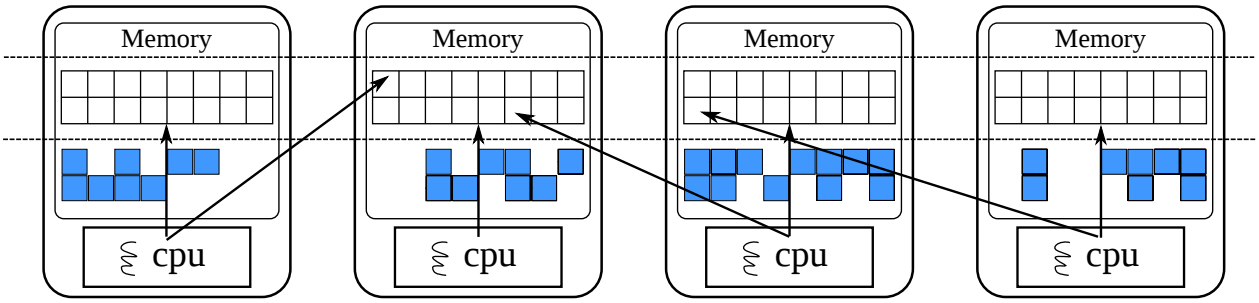


Figure 2.1: PGAS Model

Multiple process units exist during runtime, and memory located on different process unit can be remotely accessed. Fig.2.1 demonstrates the process unit and access model. Based on the actual libraries and languages being used, the name of process unit can be process, thread, image, processing element(PE) or context. The map target of process unit to system conceptions can very from process, thread or any hardware/software defined runtime context. Pieces of memory are bound to the process unit. They can be contiguous or discrete. In some languages or libraries, there might exist multiple memory bundles while others only allow one big chunk of memory. The memory bound to the process unit mainly depend on the affinity to the computing resource location, for example, process unit on NUMA architecture usually has on card bundle memory.

- **One-sided communication**

  The most classical HPC communication on Message Passin Interface(MPI) is pair of send() and receive(). In the code perspective, user call send() on the sending side and call receive() on the receiving side on the same context

point in different processes. This is a traditional two-sided communication, which means both sides of the communication need to perform actions for data/message exchanging. This impairs the CPU performance by interrupting current computing task and cause context switch in CPU.

Representatives of one-sided communication are put() and get() function series. In put() or get(), only one side of the communication need to perform the actual data transfer, the other side do not explicitly call any functions to dealing with incoming/outgoing logic. This feature is mainly implemented with Remote Direct Memory Access (RDMA), which is the data copy through network while not involving the operation system.

One-sided communication has the benefit of low latency high throughput. But an obvious drawback of one-sided communication is that it lacks notification which indicates the finish status of the data transfer. Some put/get+notify procedures was implemented to solve this problem. Another ways to solve this problem is to use global or group barrier, while this is a useful solution, but apparently barrier is the heaviest synchronization strategy and prone to the dangerous of deadlock. Tools[44] are created to avoid deadlock and strike better balance between performance and synchronization.

Below we give a brief list to introduce some languages and libraries of PGAS. The target library OpenSHMEM which this thesis work focus on will be omitted and illustrated in detail in 2.2.

- **Unified Parallel C (UPC)**[20]

UPC is an extension of C language. The user is presented with a single shared, partitioned address space. The processing unit is mapped to thread in UPC. The memory space is a logically contigous array across processing unit while the memory sizes binded to process units are identical. The communication layer of UPC is Global Address Space Networking (GASNet). Many different PGAS implementations implemented on top of GASNet, for example, the OpenSHMEM UH reference implementation.

- **Chapel**[15]

  Chapel is an programming language developed by Cray. In traditional Single Program Multiple Data (SPMD) models, all processes start the context with the same program and acquire the identity information by function like get_rank() or me(). Chapel is not a inherent a SPMD model, the default running context enters processing unit number as 0. User can use flexible defined scope, a set of processing units, on which they can define their data and computing. Chapel has keywords like forall and Python like list generators which significantly extends its usability and ability of expression.

- **CoArray Fortran (CAF)**[26, 30]

  CAF is an language extension added to Fortran 90/95, and has been a part of Fortran standard since Fortran 2008. The processing unit of CAF is called image. User can allocate symmetric variables among images. Each image hold one copy of the variable in local while other images can access this variable by image index array assignment operation.

- **Global Arrays**[42]

  Global Arrays is a PGAS library developed by Pacific Northwest National Laboratory. User can allocate a memory with specific dimension and map the array to processing units. User can perform get/put on the array for specific dimension with given indexes. The mapped memory on different processing units can be different. User do not need to know the location of array elements.

- **Titanium**[2]

  Titanium is a Java dialect developed at UC Berkeley. Titanium conforms SPMD model. In Titanium all objects allocated by a given process will always reside entirely in its own partition of the memory space. Titanium use global pointers to access remote shared memory. It allows user to copy a entire object from one process to another. Titanium also provide domain concept and foreach keyword like Chapel to perform feasible multiple processing units iteratively access.

2.2 OpenSHMEM

OpenSHMEM is a Partitioned Global Address Space (PGAS) library interface specification. OpenSHMEM aims to provide a standard Application Programming Interface (API) for SHMEM libraries to aid portability and facilitate uniform predictable results of OpenSHMEM programs by explicitly stating the behavior and semantics of the OpenSHMEM library calls. Through the different versions, OpenSHMEM will continue to address the requirements of the PGAS community.

OpenSHMEM processing unit is called Processing Element(PE). OpenSHMEM uses remotely shared memory regions called symmetric memory to communicate between PEs. Besides symmetric memory, PE can also access private memory region which part is not remotely accessible by other PEs. As an PGAS library, OpenSHMEM also use one-sided communication. In truth, OpenSHMEM is the first library which proposed the conception of one-sided communication. By using one-sided communication, communications between OpenSHMEM can hide latencies which makes OpenSHMEM ideal for unstructured, small/medium size data communication patterns. OpenSHMEM follow the Single Program Multiple Data (SPMD). OpenSHMEM provides interfaces for initialization, communication and synchronization across PEs. The OpenSHMEM specification defines library calls, constants, variables, and language bindings for C and Fortran. The C++ interface is currently the same as that for C. Unlike UPC, CAF, Titanium, and Chapel, which are all PGAS languages, OpenSHMEM relies on the user to use the library calls to implement the correct semantics of its programming model.

2.2.1 Memory Model

An OpenSHMEM program consists of data objects that are private to each PE and data objects that are remotely accessible by all PEs. Private data objects are stored in the local memory of each PE and can only be accessed by the PE itself; these data objects cannot be accessed by other PEs via OpenSHMEM routines. Private data objects follow the memory model of C or Fortran. Remotely accessible objects, however, can be accessed by remote PEs using OpenSHMEM routines. Remotely

accessible data objects are called Symmetric Data Objects. Each symmetric data object has a corresponding object with the same name, type, and size on all PEs where that object is accessible via the OpenSHMEM API. Symmetric data objects accessed via typed OpenSHMEM interfaces are required to be natural aligned based on their type requirements and underlying architecture. In OpenSHMEM the following kinds of data objects are symmetric:

- Fortran data objects in common blocks or with the SAVE attribute. These data objects must not be defined in a dynamic shared object (DSO).
- Global and static C and C++ variables. These data objects must not be defined in a DSO.
- Fortran arrays allocated with *shpalloc*
- C and C++ data allocated by *shmalloc*

OpenSHMEM dynamic memory allocation routines (*shpalloc* and *shmem_malloc*) allow collective allocation of Symmetric Data Objects on a special memory region called the Symmetric Heap. The Symmetric Heap is created during the execution of a program at a memory location determined by the implementation. The Symmetric Heap may reside in different memory regions on different PEs. Fig2.2 shows how OpenSHMEM implements a PGAS model using remotely accessible symmetric objects and private data objects when executing an OpenSHMEM program. Symmetric data objects are stored on the symmetric heap or in the global/static memory section of each PE.

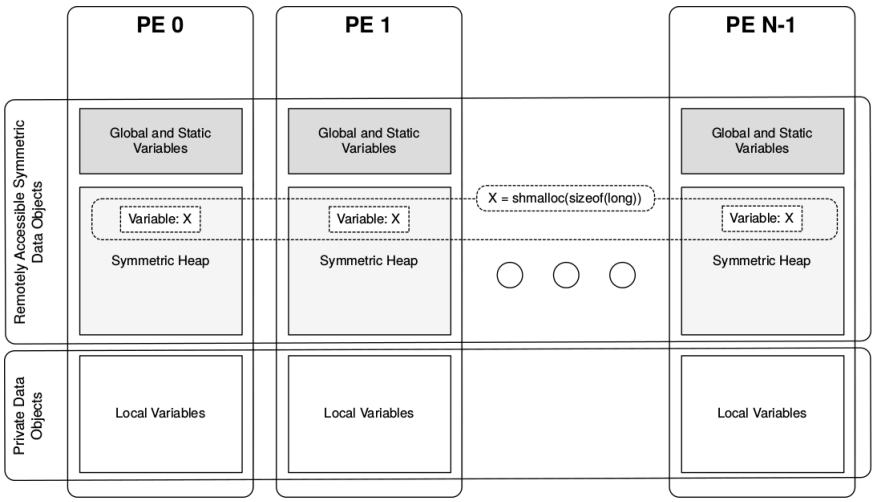


Figure 2.2: OpenSHMEM Memory Model

In OpenSHMEM the following kinds of data objects are symmetric: Since only the symmetric variables can be modified and communicated by all PEs, all applications that use the OpenSHMEM library save application critical data in symmetric variables. We use this information during our check-pointing scheme so that only a fraction of the memory consumed by the application is required to be backed up.

2.2.2 OpenSHMEM Routines and Usage

OpenSHMEM routines can be separated into seven categories. For each category, we will describe it and provide a code example to illustrate its usage.

1. **Library Setup and Query**

   (a) Initialization: The OpenSHMEM library environment is initialized.

(b) Query: The local PE may get the number of PE running the same program and its unique integer identifier.

(c) Accessibility: The local PE can find out if a remote PE is executing the same binary, or if a particular symmetric data object can be accessed by a remote PE, or may obtain a pointer to a symmetric data object on the specified remote PE on shared memory systems.

List. 2.1 shows the usage of how to initialize the runtime environment, and how to retrieve number of PEs and PE number of self. *shmem_init* is the setup function, it should be called before all other shmem function calls. *shmem_pe_accessible* can check if specific PE is accessible from caller PE.

Listing 2.1: OpenSHMEM Library Setup and Query APIs

```
1  #include <stdio.h>
2  #include <shmem.h>
3  int main(){
4      shmem_init();
5      int me=shmem_my_pe();
6      int npes=shmem_n_pes();
7      if(shmem_pe_accessible(0)){
8          printf("PE 0 is accessible from %d\n", me);
9      }else{
10         printf("PE 0 is not accessible from %d\n", me);
11     }
12     return 0;
13 }
```

2. **Symmetric Data Object Management**

(a) Allocation: All executing PEs must participate in the allocation of a

symmetric data object with identical arguments.

(b) Deallocation: All executing PEs must participate in the deallocation of the same symmetric data object with identical arguments.

(c) Reallocation: All executing PEs must participate in the reallocation of the same symmetric data object with identical arguments.

List. 2.2 demonstrates the symmetric memory usage in C programming language. In this example program, global variable *a*, static variable *c*, and memory allocated by *shmem_malloc*, i.e. **d* is symmetric. Local variable *b* is not symmetric. For ELF object type file, *a* and *c* are in *.bss* and *.data* segment, while *b* will generated on runtime stack. *shmem_malloc* is a collective operation, which allocate symmetric memory on all PEs. When symmetric memory allocated by *shmem_malloc* is no longer used, user need to call *shmem_free* to release the memory.

Listing 2.2: Symmetric Data Object

```
1  #include <shmem.h>
2  int a; /* Global variable a is symmetric */
3  int main()
4  {
5      int b; /* local variable b is not symmetric */
6      static int c; /* static variable is symmetric */
7      shmem_init();
8      /* memory allocated by shmem_malloc is symmetric */
9      int *d=shmem_malloc(sizeof(int));
10     shmem_free(d);
11     return 0;
12 }
```

3. **Remote Memory Access**

   (a) PUT: The local PE specifies the source data object (private or symmetric) that is copied to the symmetric data object on the remote PE.

   (b) GET: The local PE specifies the symmetric data object on the remote PE that is copied to a data object (private or symmetric) on the local PE.

Listing 2.3: Remote Memory Access

```
1  #include <shmem.h>
2  int t,me;
3  int main()
4  {
5      shmem_init();
6      me=shmem_my_pe();
7      if(me%2==0){
8          shmem_int_get(&t, &me, 1, me+1);
9      }
10     if(me%2==1){
11         shmem_int_put(&t, &me, 1, me-1);
12     }
13     shmem_quiet();
14     return 0;
15 }
```

List. 2.3 demonstrates how to use put/get function calls to access remote symmetric data. This program should run with even number of PEs. Line 7-9 has the same effect with line 9-13. Since the PE number always start from 0, this program will pass PE number of odd number PE to its left even number PE neighbor, Fig.2.3. The difference of put and get is that get series functions are blocking functions, which means after the function call, the data transfer

totally finished and target symmetric memory holds the correct value. While, put series function are natually non-blocking, which mean after the function call, data transfer is queued but not necessarily finished. In line 13 we explicitly call the function *shmem_quiet()* which is used to guarantee the finish of pending put operations.

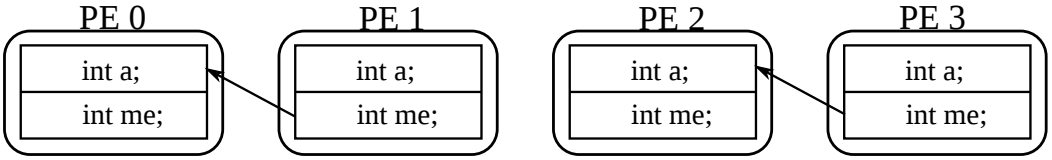


Figure 2.3: OpenSHMEM Remote Memory Access

4. **Atomics**

   (a) Swap: The PE initiating the swap gets the old value of a symmetric data object from a remote PE and copies a new value to that symmetric data object on the remote PE.

   (b) Increment: The PE initiating the increment adds 1 to the symmetric data object on the remote PE.

   (c) Add: The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE.

   (d) Compare and Swap: The PE initiating the swap gets the old value of the symmetric data object based on a value to be compared and copies a new value to the symmetric data object on the remote PE.

   (e) Fetch and Increment: The PE initiating the increment adds 1 to the

symmetric data object on the remote PE and returns with the old value.

(f) Fetch and Add: The PE initiating the add specifies the value to be added to the symmetric data object on the remote PE and returns with the old value.

List .2.4 demonstrates the use of *shmem_add*. This is an atomic function. Different PEs can add remote symmetric value through this function without considering data race confliction. This can be used to implement a global counter.

Listing 2.4: OpenSHMEM Atomic Operation

```
1  #include <shmem.h>
2  int main()
3  {
4      int me, old;
5      static int dst;
6      shmem_init();
7      me=shmem_my_pe();
8      old=-1;
9      dst=22;
10     if(me==1){
11         old=shmem_add(&dst, 44, 0);
12     }
13     shmem_barrier_all();
14     printf("%d: old=%d, dst=%d\n", me, old, dst);
15     return 0;
16 }
```

5. **Synchronization and Ordering**

(a) Fence: The PE calling fence ensures ordering of PUT, AMOs, and memory store routines to symmetric data objects with respect to a specific

destination PE.

(b) Quiet: The PE calling quiet ensures completion of remote access operations and stores to symmetric data objects.

(c) Barrier: All or some PEs collectively synchronize and ensure completion of all remote and local updates prior to any PE returning from the call.

List. 2.5 shows the usage of *shmem_barrier_all()*, this is a collective operation to synchronize all PEs. *shmem_barrier_all* also call *shmem_quiet* and *shmem_fence* internallly, which will guarantee all pending operations before barrier point be finished, and all put operations after barrier will not have misodering to finish before the put operations before barrier point.

Listing 2.5: OpenSHMEM Barrier

```
1 #include <shmem.h>
2 int main()
3 {
4     shmem_init();
5     ...
6     shmem_barrier_all();
7     ...
8     return 0;
9 }
```

6. **Collective Communication**

(a) Broadcast: The *root* PE specifies a symmetric data object to be copied to a symmetric data object on one or more remote acPE (not including itself).

(b) Collection: All acPE participating in the routine get the result of concatenated symmetric objects contributed by each of the acPE in another symmetric data object.

(c) Reduction: All acPE participating in the routine get the result of an associative binary routine over elements of the specified symmetric data object on another symmetric data object.

List. 2.6 demonstrates a sum to all reduction function call example. This function will calculate sum from PEs' *src* and store the result in each PE's *dst*. Note that there are two parameters *pWrk* and *pSync* used for the function. This is because all communications in OpenSHMEM are based on symmetric memory. *pWrk* is used for storing middle results and *pSync* is used for synchronization. These two symmetric areas can be generated by using *shmem_malloc* or global/static variables. *pSync* need to be correctly initialized before usage. An extra *shmem_barrier_all* need to be performed to guarantee all PE correctly initialized *pSync*.

Listing 2.6: OpenSHMEM Collective Operation

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <shmem.h>
#ifndef MAX
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
#endif
static const int nred = 1;
int src,dst;
int main ()
{
    int i;
    long *pSync;
    int *pWrk;
    int pWrk_size;
    shmem_init ();
    pWrk_size = MAX (nred / 2 + 1,
       SHMEM_REDUCE_MIN_WRKDATA_SIZE);
    pWrk = (int *) shmem_malloc (pWrk_size * sizeof (*pWrk));
    assert (pWrk != NULL);
    pSync = (long *) shmem_malloc (SHMEM_REDUCE_SYNC_SIZE *
       sizeof (*pSync));
    assert (pSync != NULL);
    for (i = 0; i < SHMEM_REDUCE_SYNC_SIZE; i += 1) {
        pSync[i] = SHMEM_SYNC_VALUE;
    }
    src = shmem_my_pe () + 1;
    shmem_barrier_all ();
    shmem_int_sum_to_all (&dst, &src, nred, 0, 0, 4, pWrk,
       pSync);
    printf ("%d/%d   dst =", shmem_my_pe (), shmem_n_pes ());
    printf (" %d", dst);
    printf ("\n");
}
```

7. **Mutual Exclusion**

   (a) Set Lock: The PE acquires exclusive access to the region bounded by the symmetric *lock* variable.

   (b) Test Lock: The PE tests the symmetric *lock* variable for availability.

   (c) Clear Lock: The PE which has previously acquired the *lock* releases it.

   List. 2.7 shows an example on how to use lock to protect critical section among PEs.

Listing 2.7: OpenSHMEM Lock

```
#include <stdio.h>
#include <unistd.h>
#include <shmem.h>
long L = 0;
int main (int argc, char **argv)
{
    int me;
    shmem_init ();
    me = shmem_my_pe ();
    shmem_barrier_all ();
    if (me == 1) {
        sleep (3);
    }
    shmem_set_lock (&L);
    printf ("%d: sleeping 1 second\n", me);
    sleep (1);
    printf ("%d: sleeping...done\n", me);
    shmem_clear_lock (&L);
    shmem_barrier_all ();
    shmem_finalize ();
    return 0;
}
```

2.2.3 UCCS

The work described in this thesis is part of the reimplementation of OpenSHMEM University of Houston delivery, and the core part is based on the UCCS library. UCCS is a communication middleware that aims to provide a high-performing low-level communication interface for implementing parallel-programming models. Processing unit is called context in UCCS. The interface of UCCS is aming to provide

a very thin wrapper of network hardware. This design is to minimize the software overheads and achieve the goal of low-latency communication.

Chapter 3

Fault Tolerance

Exascale computing ability is one of the main goals current HPC systems want to achieve. Millions of CPU cores, GPGPU accelerator and low latency and high bandwidth networks are equipped to current HPC machines. To reach the target of exascale, not only advanced computing hardwares are required, good softwares and programming models are also necessary, they decided how well we can use existed hardwares' computation ability[24, 8, 9].

There are different problems in front on the road to exascale computing. Fault tolerance is one of the most important one. With the increasing of system units, according to the Mean Time To Failure (MTTF) definition[35] , a more complex systems with more units will have lower MTTF. Shorter MTTF will causing great uncertainty on the running time. Traditional sharing way of HPC system need the users to apply fixed time slots for their programs. The uncertainty of MTTF brings faults and failures in random way leads to the failure of whole program running thus

creats unpredictable time and financial costs for their task solving.

One method around this issue is that try to use units with longer MTTF, but the system integration degree speed is way more faster than the increasing of single unit's MTTF. The cost of increasing MTTF of single unit growing faster. It is counter-productive when using this method to increase system's MTTF. Also the MTTF of system limited by both hardwares and softwares. A system with fault tolerance ability will be the goal of all future large HPC systems design.

Due to the complexity and variety of faults, only with fault tolerance design considered in each layer of system design thus to form a fullstack fault tolerance schema, can the system deal with all different faults including hardware and software failures[14].

In the following sections, we will briefly talk about the fault tolerance techniques used before, including automatic fault tolerance and algorithm based fault tolerance. We will also talk about the software model and libraries we used in our implementation.

3.1 Automatic Fault Tolerance

Automatic fault tolerance is sometimes referred as masked fault tolerance due to the fact that it will deal with fault in system and library level by itself, user don't need to change any line of their code to use the fault tolerance feature[18, 31, 34].

Checkpoint-base rollback[11] and log-based[10] rollback are two most common

used techniques in automatic fault tolerance.

3.1.1 Checkpoint-based Rollback

- **Uncoordinated Checkpointing**

  Uncoordinated checkpointing has the maximum flexibility in checkpints decision. [27]. Processes can do checkpoints their middle status based on their own overhead or other merits. The drawback of uncoordinated checkpointing is that it will generate inconsistent backup between processes which might lead to error computation results.

- **Coordinated Checkpointing**

  Coordinated checkpointing requires processes to orchestrate their checkpoints in order to form a consistent global state. Global barrier to block the whole program is the most common way for coordinated checkpointing[23, 17, 48, 1]. Coordinated checkpointing can guarantee the consistency between processes. The drawback is that the overhead of synchronization is high.

- **Communication-induced Checkpointing**

  Communication-induced checkpointing trys to combine the benefit of both coordinated checkpointing and uncoordinated checkpoiting. The core idea is to do checkpointing when there is no data dependencies. And synchronize between processes for consistency when dependency related operation like incoming data arriving which overwrite local data happends.

3.1.2 Log-based Rollback

Opposed to checkpoint-based rollback recovery, log-based roolback exploits the fact that process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of nondeterministic event[27][21]. Log-based rollback hold the assumptions that nondeterminitic and determinants can be logged to stable storage. If a process whose state depend on nondeterministic event which cannot be reproduced, this process is called an orphan. There are three kind of logging methods based on how they deal with orphan process.

- **Pessimistic Log**

  Pessimistic log guarantees that orphans are never created due to a failure. With the expense of high failure free performance overhead.

- **Optimistic Log**

  This method allows orphans to be created due to failures, it reduces the performance overhead while complicates the recovery procedure.

- **Causal Log**

  Causal log attempts to combine the advantages of low performance overhead and fast output commit, buy they may require complex recovery and garbase collection.

Automatic fault tolerance is the most compelling fault tolerance from the user's perspective due to the zero change to source code. Coordinated checkpoint has the relative low overhead thus has been the most popular automatic fault tolerance

strategy choice. But even with this fact, automatic fault tolerance still suffers pretty high performance overhead which motivates people to find better ways for fault tolerance[13][10]. ABFT is another kind of attempt people focused on later.

3.2 Algorithm Based Fault Tolerance (ABFT)

Algorithm Based Fault Tolerance (in some cases refered as Application Base Fault Tolerance) does not rely on the FT features provided by the system like automatic fault tolerance have, it is an ad-hoc solusion for specific algorithm and program.

Opposed to masked fault tolerance, ABFT is sometimes referred to as unmask fault tolerance. Failures are not hiding from users like in masked fault tolerance, the fault detector will notify the user level in some way include but not limited to return value, signal or timeout. To have the fault tolerance feature, user need to modify their source code to deal with failures.

Davies[22] proposed an algorithm-based fault tolerance for High Performance Linpack (HPL). In this work they tailor the FT checkpointing to fail-stop manner instead of periodical checkpoint to achieve neglegible FT overhead. [37] and [25] implemented ABFT for LU decomposition and transposition. Wang[51] implemented ABFT fast fourier transformation (FFT) network.

ABFT method requires the users to design the fault tolerance feature by themselves. Even though this way gives them maximum free for the FT design, and usually a well implemented ABFT program would have lowest overhead, but due to

the complexity to design an FT algorithm and the lacking support from the library, ABFT only achieved limited successes[32, 7, 6].

3.3 User Level Fault Mitigation (ULFM)

ULFM is not a standalone fault-tolerance strategy, it tries to provide feasible routines to help users construct their own ABFT by providing language or library level fault tolerance support. In ABFT user need to write all codes for their fault mitigation which would involve complex process and memory processing. ULFM will provide an abstract layer for failure detection and fault mitigation which would greatly alleviate the pain by doing ABFT from scratch.

3.3.1 FT-MPI

Efforts toward fault tolerance in MPI have previously been attempted[40]. The most notable past effort is FT-MPI[28]. The fault tolerance of FT-MPI is focus on the communicator reconstruction. Several recovery modes were available to the user.

- **SHRINK**: The communicator is shrink so that there are no holes in its data structures. The ranks of the processes are changed, forcing the application to recall MPI_COMM_RANK.
- **BLANK**: This is the same as SHRINK, except that the communicator can now contain gaps to be filled in later. Communicating with a gap will cause an

invalid rank error. MPI_COMM_SIZE will return the size of the communicator, not the number of valid processes within it.

- **REBUILD**: This mode forces the creation of new processes to fill gaps. The new processes can either be places in to the empty ranks, or the communicator can be shrank and the processes added the end. This is used for applications that require a certaain size to execute as in power of two FFT solvers.

- **ABORT**: Is a mode which effects the application immediately an error is detected and forces a graceful abort. The user can not trap this, and only option is to change the communicator mode to one of the above modes.

Communications within the communicator are controlled by message mode for the communicator which is either:

- **NOP**: No operations on error. I.e. no user level message operation are allowed and all simply return an error code. This is used to allow an application to return from any point in the code to a state where it can take appropriate action as soon as possible.

- **COUT**: All communication that is NOT to the effected/failed node can continue as normal. Attempts to communicate with a failed nodde will return errors until the communicator state is reset.

FT-MPI lack the user notification and has weak recovery ability. This limits FT-MPI from broadly used. There is no standardization procedure of FT-MPI and it was mostly used as a play ground for understanding the fundamental concepts.

3.3.2 Run Through Stabilization (RTS)

Another effort to introduce failure handling mechanisms is the Run Through Stabilization proposal[39]. There are two main goals of RTS proposal:

- Keep application continue run without affected by affected processes.
- Provide notifications to processes of failure information via error handlers.

RTS requires a perfect failure detector for fail-stop process failure (i.e., a process is permanently stopped, often due to a crash)[16].

From the perspective of one process, other processes can be in one of the following states (prefixed with MPI_RANK_STATE_): OK, FAILED, or NULL. Processes with state OK are executing normally. Processes with state FAILED have been detected by MPI as failed-stop. Processes with state NULL are failed processes treated as if their ranks are MPI_PROC_NULL.

Listing 3.1: RTS Validation API set

```
1 MPI_Comm_validate{_all}(MPI_Comm c, int *newfailures);
2 MPI_Comm_validate_get_num_state{_all}(MPI_Comm c, int type, int
    *count);
3 MPI_Comm_validate_get_state{_all}(MPI_Comm c, int type, int
    incount, int *outcount, MPI_Rank_info rank_infos[])
4 MPI_Comm_validate_get_state_rank{_all}(MPI_Comm c, int rank,
    MPI_Rank_info *rank_info)
5 MPI_Comm_validate_set_state_null(MPI_Comm c, int incount,
    MPI_Rank_info rank_infos[])
```

A process uses the validation function in Listing 3.1 to update access and modify the know state of a process in a communicator. MPI_Comm_validate variants are designed for local recognition and point-to-point communication. MPI_Comm_validate_all variants are designed for collective communication.

Besides the validation procedures, RTS introduced the failure handlers for uniform failure notification. Huge complexity imposed by it avoided RTS to eventrually become the MPI standard.

3.3.3 ULFM for current MPI

The prominent interfaces proposed to enable effective support of ULFM for MPI applications is descriped in [5]. This new ULFM model was build around three concepts:

1. Simplicity, the API shoul be easy to understand and use in most common scenarios.

2. Flexibility, the API should allow varied fault tolerence models to be build as external libraries.

3. Absence of deadlock, no MPI call (point to point or collective) can block indefinitely after a failure, but must either succeed or raaise an MPI error.

In the new ULFM proposal, the monitor of processes need not to actively communicate with the processes and it also want to avoid use the high overhead global synchronization to guarantee consistency. Another design point is that return status

on remote processes are no longer indicator of failure, instead of that, only on a particular rank the failure is reported. The following functions provide the basic blocks for maintaining consistency and enabling recovery of the state of MPI.

- **MPI_COMM_FAILURE_ACK & MPI_COMM_FAILURE_GET_ACKED**: These two calls is used for failure notifications.
- **MPI_COMM_REVOKE**: This API is used for revoke any operation on the failed processes which may lead the program to deadlock status.
- **MPI_COMM_SHRINK**: This API is used to create a new communicator which excluded the failed processes. This is a collective communication.
- **MPI_COMM_AGREE**: This operation provides an agreement algorithm when strong consistency is necessary.

Several applications, ranging from Master-Worker to tightly coupled[3, 43], are currently being refactored to the new ULFM of MPI. The new progress of MPI ULFM can be found on `http://fault-tolerance.org`fault-tolerance.org.

3.4 PGAS related ULFM

Research in the area of fault tolerance for the PGAS programming model is sparse. Ali, et al.[4] proposed an application-specific fault tolerance mechanism. They achieved fault-tolerance using redundant communication and shadow copies. They

evaluated the approach by implementing it as a part of Global Arrays, a shared-memory programming interface for distributed systems, and using NWChem [50], a framework specifically for computational chemistry problems. Our approach is more encompassing and not limited to a specific application or kernel.

A fault tolerance mechanism based on GASPI[46] uses time-out mechanisms for all non-local procedures. Though this is more general purpose, it cannot advantageously use the unique features of OpenSHMEM. Other researchers have proposed fault-aware and fault-tolerant models for message passing models, particularly MPI. Bland et al.[5] have proposed and evaluated user level failure mitigation extensions to MPI. This provides an interface for handling faults by the user (HPC application) without aborting the entire MPI job [6]. Graham et al. [29] have proposed similar models and evaluated MPI collective algorithms in the presence of faults [38].

In this thesis, we explore the best practices in the fault-tolerance domain to propose a solution addressing the fault tolerance challenge for OpenSHMEM.

Chapter 4

Models

In this section, we classify types of errors in the HPC environment and discuss popular fault tolerance techniques. Based on this foundation we design a Fault Tolerance model for OpenSHMEM.

4.1 Faults Definition

In order to define the fault tolerance model for OpenSHMEM it is important to classify and overview typical *High Performance Computing* (HPC) system faults. The fault tolerance community has identified three major classes of faults:

- **Permanent fault** is a result of a permanent malfunction/failure in the system. The failure is considered permanent when it causes long time unavailability of resources in the system, which requires administration intervention such as

system restart or hardware replacement.

Typically, these faults are caused by permanent hardware failures in power supply, network, CPU, or memory hardware. In certain cases, these faults might be caused by a substantial software error such as OS kernel panic and crash.

- **Transient fault** is a result of a temporary malfunction/failure in the system. Temporary failure is a short term event that results in a temporary unavailability of the system. Once system availability is back it is expected that the application availability is back with the preserved software stack.

  This type of fault is typically caused by temporary malfunctions such is a bit flip in memory or temporary connection problem in the network.

- **Silent fault** (often references as "silent error") is the result of a malfunction/-failure that never got detected or got detected after extended period of time. These faults may result in invalid application behavior or incorrect computational results.

  This type of error may be caused by bit flip in the memory that is not protected with error correction code (ECC).

Silent faults we omit from this work, because these are difficult to detect and are not well studied.

In this thesis, we focus on handling permanent and transient types of faults. Similar to most fault tolerance designs, we treat the permanent and transient faults

in the same way; once the fault occurs the problematic node (or component) is marked as unavailable and notification is issued to the system manager.

4.2 Fault Handling Approaches in HPC

Fault handling approaches are well studied and researched by the HPC community. The *Message Passing Interface* (MPI) community is getting ready to introduce fault tolerance support in upcoming versions of the MPI standard. In addition, some MPI implementations already support checkpoint/restart capabilities in their libraries. Charm++, parallel programing languages also provides a built-in checkpoint/restart mechanism that supports multiple recovery algorithms. The majority of the fault tolerance approaches can be classified into the two main categories:

4.2.1 Masked Approach

In the masked approach the system, a communication library or hardware does not expose the errors to the application by handling it internally within a system.

The checkpoint/restart mechanism, which is also called rollback in the literature, is one of the most popular masking approaches. The checkpoint mechanism allows dumping the memory of an entire process to a storage (disk or memory). The process of taking a checkpoint may occur automatically within a certain period of time or may be explicitly requested by the application. The checkpoint of the distributed application requires substantial storage resources and time. For example, OLCF's

TITAN [36] system has 710TB of memory and the storage system reaches 240GB/s of a peak bandwidth. Based on this information we may estimate the it takes $\frac{710TB}{240GB/s} \approx 50min$ to checkpoint the entire system under ideal conditions. As a result the solution is applicable for relatively small systems, but is challenging to scale with the size of future systems.

4.2.2 Unmasked Approach

The unmasked approach assumes that the fault is not masked and propagated to the application level. This approach substantially simplifies the software stack of a HPC system, since the major burden of the application recovery is the responsibility of the application developer. The application is responsible for handling and recovering from the fault, while the system software stack is responsible for providing the mechanisms for stabilization of the application. Once the fault is propagated, the application selects a strategy for application recovery. Since the application has the full perspective of the current state of execution, it allows the application developer to implement the best strategy for application recovery.

4.3 Fault-Tolerance Design

In order to develop fault-tolerance design, first we identify critical design points and then based on the conclusions we define the model.

4.3.1 Design Points

Based on the previously discussed research work in the field we identified the following design points:

- **Coverage**

  Transient faults should be handled the same way as permanent faults. If the transient fault was not handled (masked) by hardware or device driver recovery mechanisms, the failure is considered as a permanent one.

- **Minimize Overheads**

  One of the primary reasons for the overheads is a lack of information about the *importance* (or usage) of various regions of the memory. As a result, the majority of the proposed solutions checkpoint the entire memory of the application.

  From this perspective, OpenSHMEM defines a convenient memory layout that separates the memory to symmetric and private regions (Fig. 2.2). We use this information to save only communicable symmetric data objects.

- **Choice of approach**

  The unmasked approach provides a lot of flexibility for fault handling on the application level. This is one of the reasons why the MPI community is moving towards this approach. We follow the same approach and expose the basic set of operations enabling implementation of fault tolerant applications.

4.3.2 API Behavior Requirements

Some APIs need to be redesigned in fault tolerance OpenSHMEM to guarantee fault tolerance work correctly. List below are the requirements for all APIs in fault tolerance OpenSHMEM implementation.

- Processes' failures should not trigger exit of other PEs. In some implementations with bootstrap program, monitoring process will cause global exit of all PEs, this behavior need to be change to allow the contigous running while failure exists.

- Any API call that involves a failed process must not block indefinitely. APIs should have return logic from failure to allow process enter fault tolerance procedure instead of block forever.

- APIs' behavior need to be consistent with failed PE's absent and substitute PE's participation. From users perspective, substitute PE should not have any identity awareness after fault mitigation. No further behavior changes are allowed due to the substitutes. For example, some implementations use internal counter for *shmem_barrier_all*, this counter need to be consistent after fault mitigation stage.

- Substitute PE need to inherit failed PE's context include PE number. Same with last requirement, user should not change their code specifically for substitute PE. PE number and other PE identity information should be kept same with failed PE.

4.3.3 Checkpoint-and-Restart Model for OpenSHMEM

The checkpoint mechanism allows dumping the memory of an entire process to a storage (disk or memory). The process of taking a checkpoint may occur automatically in a certain period of time or may be explicitly requested by the application. When failure occurs, for example a fatal failure of a node, the process can be restarted from the checkpoint. The checkpoint/restart mechanisms are divided into the two types:

- Checkpoint/restart with a coordinated rollback, where all *Processing Element*s (PEs) in the system are restored to the latest checkpoint
- Checkpoint/restart with a partial rollback, where only a subgroup of PEs affected by the failure is restored to the latest checkpoint.

Although theoretically this approach handles all possible errors in a way that is transparent to the application and users, it has a drawback. The checkpointing of a distributed application imposes substantial overheads of time and space.

One of the primary reasons for the overheads is a lack of information about the *importance* (or usage) of various regions of the memory. As a result, the majority of the proposed solutions checkpoint the entire memory of the application.

In order to mitigate the cost of the checkpoint/restart overhead researchers suggested implementation of *selective* memory checkpoint/restart, where the application explicitly specifies what memory regions are *critical*[49]. While it is possible to reduce the overhead using this approach, it offloads the burden from the programming model to the application level, where the developer has to identify the critical sections

of the memory.

From this perspective OpenSHMEM (and PGAS) defines a convenient memory model that separates memory to symmetric and private regions (Fig. 2.2). Since, the symmetric memory is the only memory that is visible to other PEs, it is the key element that represents the state of PE in a distributed system. Failure to access this memory leads to a failure of an OpenSHMEM application which is similar to the failure of a regular process or a failure to access the physical memory. These lead to a conclusion that in order enable fault-prone execution of the process, the OpenSHMEM library has to implement a capability that enables persistent access to remote memory. Since the symmetric memory is only part of the representation of PE in the system, it is critical for the OpenSHMEM fault tolerance model to recover this particular portion of the memory. We leverage the unique characteristic of the OpenSHMEM memory model to introduce explicit checkpoint/restart functionality for symmetric memory regions only. Such a pin-pointed checkpointing technique allows reduction of the overall amount of memory backed up in the checkpoint process.

The proposed checkpoint-and-restart model consists of the following core operations:

- **Checkpoint:** the operation checkpoints the current state of the OpenSHMEM symmetric heap and global variables. This process implicitly synchronizes; that ensures completion of all outstanding OpenSHMEM operations
- **Fault detection:** the operation detects a failure and notifies the OpenSHMEM application about the failure in a system.

- **Checkpoint recovery:** the operation rolls back the PE's symmetric memory and global variables to a previously stored checkpoint and a replacement PE takes over the role of the failed PE.

In order to implement the model and the above operations we propose the following extensions for the OpenSHMEM API:

- ***int shmem_checkpoint_all(void)*** - this routine is a combination of checkpointing and fault detection. The return value indicates if a fault occurred since the previous status check. If a fault is detected, it will return a fault state immediately without checkpointing. If no fault is detected in the last computational step, it will check-point all symmetric regions and return *SHMEM_FT_SUCCESS*. The routine is a collective operation and requires a participation of all active PEs.

- ***void shmem_query_fault(int **pes, int **pes_status, size_t *numpes)*** - this routine is used to retrieve a list of failed PEs and reasons for their failure. It allocates memory for the *pes* and *pes_status*. A user needs to free this memory when no longer needed. *numpes* gives how many PEs failed since previous status check.

- ***void shmem_restart_pes(int *pes, size_t numpes)*** - this routine is a coordinated rollback operation that restores the symmetric memory region from the last checkpoint on all PEs. It will activate *sleeping* PEs to transform to a working PE. The substitute PE or PEs will leave *start_pes* when activated, skip the *shmem_ft_algo_init()* part and finally join into this function call. They will

retrieve the backed up data of the dead PE, replace it and continue execution with the dead PE's PE number. The routine is a collective operation and requires a participation of all PEs.

- ***int shmem_ft_algo_init(void)*** - this routine returns *true* for the original PEs, while for newly spawned substitute PEs it returns *false*. The function is used to guard the initialization code in order to avoid initialization stages on substitute PEs.

Listing 4.1: ULFM frame

```
1  #include <shmem.h>
2  /* User algorithm states as global variables */
3  int step;
4  int main(int argc, const char *argv[])
5  {
6      start_pes(0);
7      if(shmem_ft_algo_init()){
8          /* User algorithm initialization goes here */
9      }
10     do{
11         if(SHMEM_FT_SUCCESS!=shmem_checkpoint_all()){
12             int *pes, *pes_status;
13             size_t numpes;
14             shmem_query_fault(&pes,&pes_status,&numpes);
15             shmem_restart_pes(pes,&numpes);
16             shmem_barrier_all();
17             free(pes);
18             free(pes_status);
19         }
20         else{
21             /* User algorithm computing loop goes here*/
22             step++;
23         }
24     }while(step<MAX_STEP);
25     /*User algorithm result report goes here*/
26     return 0;
27 }
```

In Listing 4.1 presents an pseudo code of an example that demonstrates how the above API can be used for an implementation of fault tolerant computation algorithms. On line 6 *start_pes()* is invoked in order to initialize the of OpenSHMEM

environment. Lines 10 to 24 represent the loop iterating over the core computational part of the algorithm. On an iteration of the loop the application invokes *shmem_checkpoint_all()* routine that serves the two goals. First it checks if a failure has occurred since last the checkpoint, and second if no error is detected (return value SHMEM_FT_SUCCESS) it executes the checkpoint operation.

If a failure is detected (return value SHMEM_FT_FAILURE), the algorithm enters the failure branch between lines 12 and 18. *shmem_query_fault()* identifies the indexes of failed PE and a potential reason. Then *shmem_restart_pes()* is used to recover the symmetric memory segments and activate the replacement PEs. Using the call all active PEs retrieve their last checkpoint data from a backup.

It is important to point out that once replacement PEs are activated and initialized within OpenSHMEM runtime environment (*start_pes*) they reach the *shmem_ft_algo_init()* routine. For the replacement PEs the function returns *false* and therefore these PEs skip the algorithm initialization step (line 8); the replacement PEs resumes execution from the last successful checkpoint. Once all PEs are rolled back to their previous step, the algorithm invokes *shmem_barrier_all()* to synchronize between all PEs. Then the application is free to release the temporary memory allocated by *shmem_query_fault()*.

Chapter 5

Implementation

5.1 Implementation Details

In this section we describe the details of the proof-the-concept implementation of the OpenSHMEM library with checkpoint/restart capabilities. Our implementation is based on OpenSHMEM reference implementation [45] developed by University of Houston and the UCCS communication library [47]. The memory checkpoint operation is implemented using the *shmem_checkpoint_all()* routine. The routine backups symmetric memory region, its meta-data and the application *bss* and *data* memory segments that are extracted from the ELF binary format. For the rest of the paragraph all above memory segments will be referenced as as a symmetric memory.

Our implementation of the checkpoint/restart is based on coordinated checkpoint methodology, therefore in an event of restart all participating PEs have to restore

their symmetric memories to a previous state. This leads to a fact that, in addition to maintaining the backup information of the remote PE symmetric memory, each PE has to maintain a copy of its own symmetric memory that is used for a local rollback in event of failure of the corresponding backup PE. As a results, the size of the shadow memory is nearly double of the size of the symmetric memory. It is worth pointing out that this memory consumption overhead is imposed by the nature of the coordinated checkpoint/restart mode. For an uncoordinated restart, which is out of scope of this thesis, the backup of the local memory is not required and therefore the memory consumption is substantially less. Our implementation of the OpenSHMEM checkpoint/restart supports both modes of recovery, nevertheless in this work we focus on an algorithm that requires coordinated recovery.

Another challenge of the work is associated with the spawning of a new PE for a replacement of failed one. The current OpenSHMEM runtime does not support spawning of a new process. In order to overcome this temporary limitations of the runtime, our implementation pre-spawns the replacement PE during the OpenSHMEM initialization process (*shmem_start_pes()*). The pre-spawn PEs are started in sleep mode and do not consume computational resources. Once failure occurs, one of the pre-spawn PEs get activated and the symmetric memory of the failed PE is transferred to the newly activated PE. The activation of the PE is a collective operation since all PEs in the system have to be update their address tables with the physical address (network and memory) of the replacement PE.

5.1.1 Memory backup and recovery

Since communication between PEs primarily uses symmetric memory, to use this model users are required to store their algorithm data, include computing steps, working set, and other state variables in global memory, static memory or memory retrieved through shmalloc. The uninitialized global and static variables will be put into executable files' *bss* segment, while initialized data in *data* segment.

All symmetric regions visible to user is fully backed up, for our current implementation it is the object file ELF, which includes *.bss* and *.data* segments from executable file plus the symmetric heap created at the initialization of the OpenSHMEM library environment. For simplicity, we did not exploit complex backup algorithm.We backup the memory chunk contiguously to an target memory area. The backup target memory area choice would be another topic. The choice should allow backup to enjoy an high transfer bandwidth and low latency while guarantee low failure coupling. Since the goal of this thesis is to provide a proof-of-conception implementation, we choose the shadow memory locates on next PE as backup target for simplicity. Thus next PE(with PE number $(mype + 1)\%npes$) will be the backup target PE and previous PE(with PE number $(mype - 1 + npes)\%npes$) will be the backup source PE, although, in practice these two PEs could be the same. When *shmem_checkpoint_all()* is called and no fault is detected, the memory backup procedure begins, all alive PEs use *shmem_get()* to grab its backup source PE's symmetric heap and store it to a symmetric but user invisible shadow memory area. This shadow memory region is created with size equal to sum of bss/data/heap sizes as recorded during library initialization. Fig 5.1 shows the backup of symmetric memory. This

backup mechanism will use twice the memory space of non-ft program.

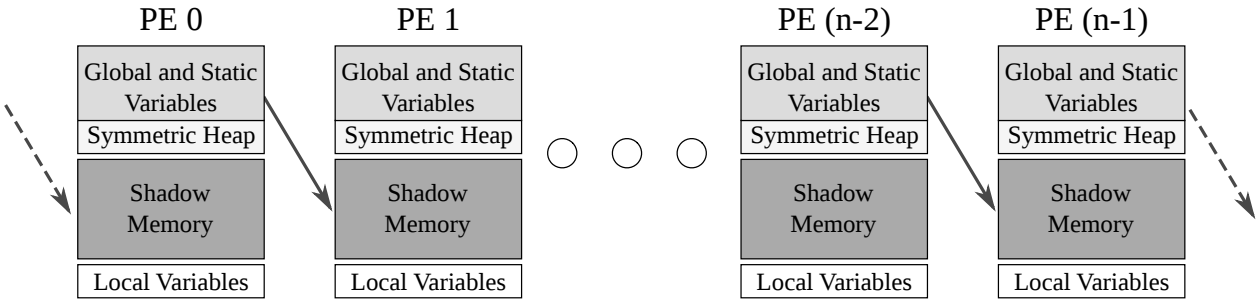


Figure 5.1: Symmetric Memory Backup

Single backup and dual backup

When a fault happens, all PEs are required to fallback to last checkpoint status, but failed PEs not only lose their own data, but also lose their backed up data for their backup targets, those backup targets are unable recover to the last checkpoint. In Fig 5.2, *f* is the failed PE, *f-1* is the backup source of *f*, *f+1* is the backup target of *f*. To guarantee all PEs have synchronized version of backed up data after failure recovery, each one need to get the symmetric memory backed up copy on the backup target PE. But for PE *(f-1)*, due to the failure happened on PE *f*, it can not retrive its backuped data from PE *f*. That means not all PE can successfully roll back to previous version of symmetric memory before failure happened.

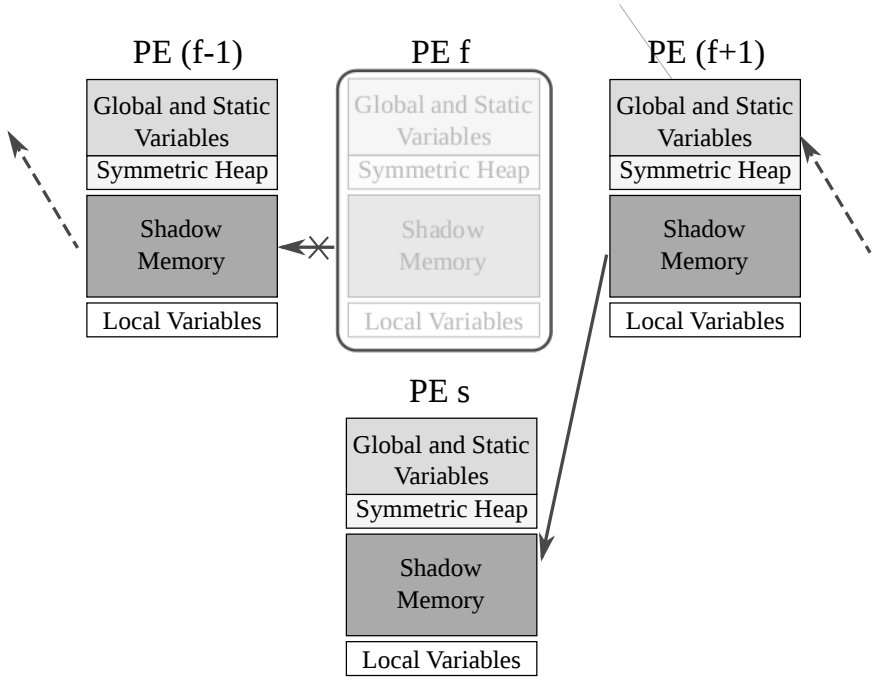


Figure 5.2: Symmetric Memory Recovery with Single Backup

For some algorithm with data inconsistence tolerance, like N-body problem, if most PEs recovered to old status while some PEs hold the new data, it is not harmful to continue the computing. For other algorithms, inconsistence data will cause computing error in following steps.

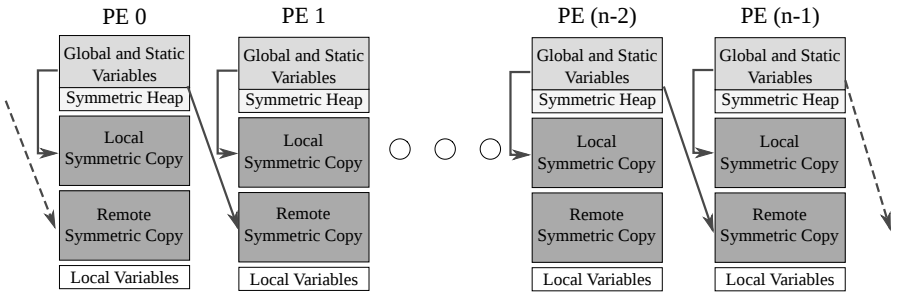


Figure 5.3: Symmetric Memory Backup with Dual Backup

Our solution to this problem is to use dual backup, a local buffer with same size to shadow memory is created in shmem initialization. During *shmem_checkpoint_all()*, each PE copy its own symmetric regions to the local buffer. If any PE failed, all alive PE could retrieve its last checkpoint from local backup, which also alleviate network overhead compare with remote recover. But in this way, the fault-tolerance program will use triple time memory space of non-ft program. Fig 5.3 illustrates dual backup procedure, symmetric variable will have two copies with backup target in remote shadow memory on target PE and local shadow memory.

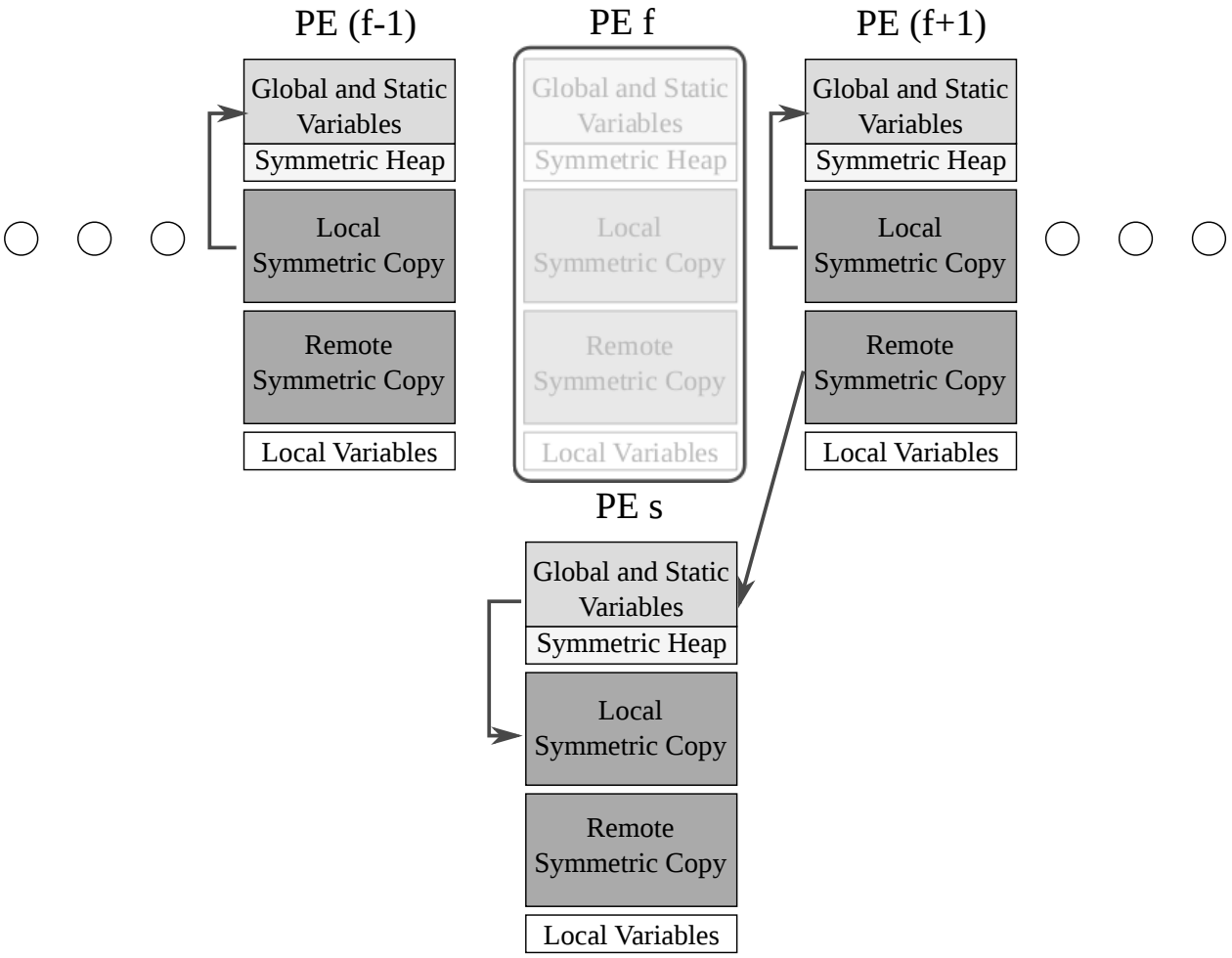


Figure 5.4: Symmetric Memory Recovery with Dual Backup

When failure happens, PEs alive will retrieve backed-up data from local memory. Substitute PEs is marked as PE *s* in figure, which retrieve symmetric copy from failed PE's backup target PE *f+1*. A backup operation from symmetric memory to local shadow memory is performed immediately after substitute PE received data. The goal of this operation is to guarantee PE *s* will get correct data from local copy

if another failure happened just after the fault recover stage.

We provided both way for user to choose to fit better their algorithm, by passing value “single” or “double” to environment SHMEM_FT_BACKUP_MODE let you choose which way you want to checkpoint backup symmetric memory.

5.1.2 Shadow memory implementation

Our OpenSHMEM UH implementation's software layer is illustrated in 5.5, on the bottom is comms layer, which build on top of UCCS. Functionality modules like put/get, atomic, barrier and others will use the internal function call provided by comms layer. The issue of previous implementation is that bss/data/heap segments use a static way for managing their start address, size and offset calculation. We reimplmented the comms layer and some higher layer module like memory and barrier. The new implementation use a table-driven style way for memory management and message routing. It allow us to creat segments with own flags and tags, thus we no longer limited to operate bss/data/heap segments.

The new implementation provided the ability to support shadow memory regions, during the startup, shadow memory is create the same way with bss/data/heap segments, the only difference is that it created with a different tag, in order to avoid being recursively backing up itself.

In order to store the backup each PE allocates a so-called *shadow* memory segment that is used to store the checkpoint of the symmetric, *bss*,*data* memory sections. In our implementation we used a simple checkpoint pattern where a PE i backs up

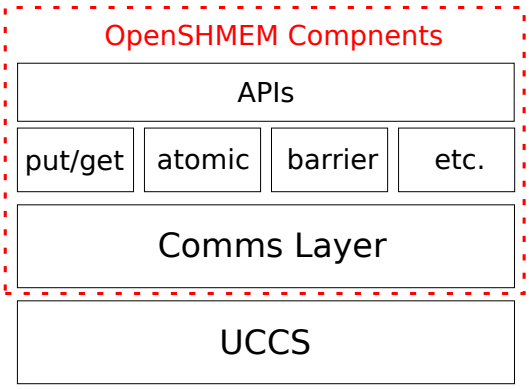


Figure 5.5: OpenSHMEM software layer

its memory segments to PE $(i+1)$, with the exception of the last PE N that backs up its segments to PE 0 (Fig. 5.1). For the memory transfer (backup) from the main memory to the shadow memory we use *shmem_mem_get* routine, which is available within the OpenSHMEM specification.

5.2 Assumptions

In this section we present some of core assumptions that have been made in respect to our implementation of the checkpoint/restart capabilities:

- Dynamic linkage

  The current implementation of the checkpoint/restart mechanism assumes that the application is linked dynamically with the OpenSHMEM library. The primary reason for this assumption is the fact that the current implementation

does not have the capability to distinguish between *bss* and *data* memory segments of the OpenSHMEM library and the application when those are linked statically. As a result, under the static linking the recovery of the *bss* and *data* sections of the application may reset the internal global state (global and static variables) of the OpenSHMEM library. This is temporary limitation of the current implementation that can be fixed by selective recovery of the sections or a simple removal of the global and static variables from the internal usage within the OpenSHMEM library implementation.

- Address space recovery

  When a new PE is restarted and the symmetric memory is recovered from a checkpoint, the address space of the checkpointed symmetric memory may not be aligned with the base address space of the symmetric memory and the data cached in it. This is a potential side-effect of virtual memory randomization that has been recently enabled in the linux kernel. In order to overcome this challenge we remap the symmetric heap memory of the recovered process to be aligned with the rest of process. Nevertheless, the remapping process is not guaranteed and potentially may fail if the selected virtual memory region is allocated for some reason. As an alternative approach, in order to avoid the problem altogether we suggest disabling the virtual memory address randomization feature in the kernel.

- Restricted use of *shmalloc*

In our current implementation of the checkpoint/restart capabilities the application is not allowed to call shmalloc in between of checkpoint operations. As a result, all the symmetric memory has to be allocated during the algorithm initialization stage. This restriction is imposed by the *dlmalloc*[41] memory allocation used in our implementation. *dlmalloc* uses a set of mutexes and opaque pointers to manage the *shmalloc* memory allocation requests, which makes the recovery of the meta-data very challenging. This is a temporary restriction and it can be addressed by implementation of a meta-data transferable allocator.

- Fault detection mechanism

  The current implementation of the OpenSHMEM library does not implement a fault detection mechanism. As a result, all the faults have to be simulated on the OpenSHMEM level. We use environment variables to set a set of global counters and failed PEs which can be detected by *shmem_checkpoint_all()*. When this function is called as the times same with the counters indicated, PEs in the failed set will silently stop their mainloop and went into sleep while PEs not in the set will return *SHMEM_PE_FAILED* immedieately.

Chapter 6

Evaluation

6.1 Testing Environment

We tested our scheme on a distributed memory cluster with 16 nodes. Each node has four 2.2 GHz 12-core AMD Opteron 6174 processors (12 cores per socket, 48 cores total), 64GB main memory and dual port Infiniband ConnectX HCA (20GB per second).

6.2 Overhead Analysis

For all our experiments the PEs are allocated by core, so PE allocation will exhaust all available cores on a node before moving to the next node. In our figures we refer to the OpenSHMEM implementation with fault tolerance (checkpoint/restart)

capabilities as *ft* and the one without such capabilities as *non-ft*. Once a failure is detected (for our experiment, we artificially introduced failures), the program loses results mid-computing. If the checkpointing backup time is T_b, recover time is T_r, single computing step time takes T_s, the total computing steps are s and f failures occur during a program execution; the time overhead of fault tolerance program T_{ft} can be calculated as:

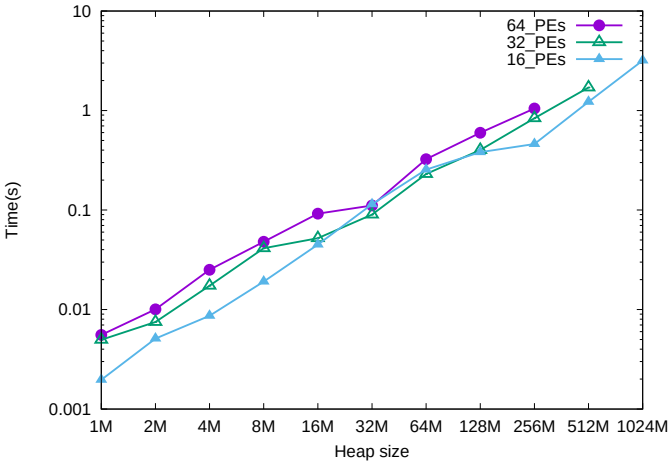
$$T_{ft} = s * T_b + f * (T_r + T_s)$$

O_{ft} is the fraction of the total time T_{ft_total} (fault tolerance version program full running time without faults) required for fault tolerance, it can be calculated as:

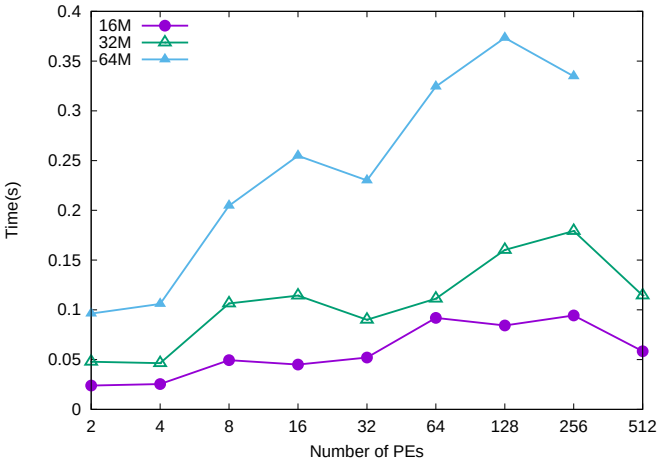
$$O_{ft} = T_{ft}/T_{ft\_total}$$

Fig. 6.1a shows checkpointing time with different heap sizes (some missing points caused by lacking system memory). The size of the global and static variables in the test program were less than 1K so we ignore them here. From the plot we observe that the backup time grows linearly with the heap size. With more PEs backup time shows an acceptable difference, this is mainly because the total amount of communication increases while the bandwidth between PEs is fixed. Fig. 6.1b shows a more detailed plot for the effect increasing the number of PEs has on the backup performance. As the plot shows, the increase of backup time when the number of PEs increase is very gradual, this means our fault tolerance checkpointing has good scalability characteristics.

Fig. 6.2a shows the time required for recovery with different heap sizes (some missing points caused by a lack of system memory). Here too, the size of global and static variables in test program were less than 1K so we ignore them here. From the plot we observe that the recovery time grows with the heap size, but it is below 10 seconds for 64 PEs with 1024 MB of symmetric heap size, which seems negligible considering the total execution time of long running applications. Fig. 6.2b shows the effect of increasing the number of PEs on recovery time. The recovery time is dependent on where the shadow region resides with respect to the location of the substitute PE. If the symmetric heap memory information is sent over the network, it will be affected by network traffic and related issues. This can be tackled in the future using topology information and using substitute PEs *nearest*(not on the same node) to the backup PE.

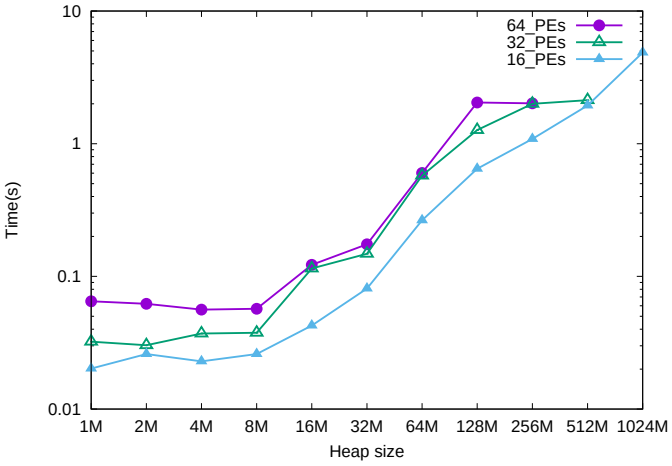


(a) Effect of symmetric heap size on checkpointing time

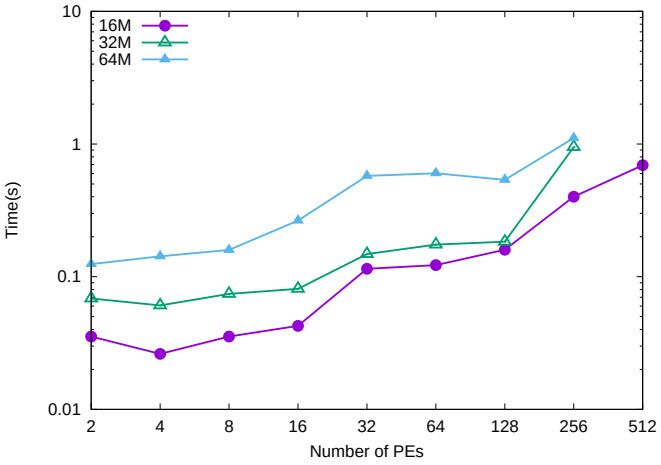


(b) Effect of number of PEs on checkpointing time

Figure 6.1: Effect on checkpointing time.



(a) Effect of symmetric heap size on recovery time



(b) Effect of number of PEs on recovery time

Figure 6.2: Effect on recovery time.

6.3 Jacobi Kernel

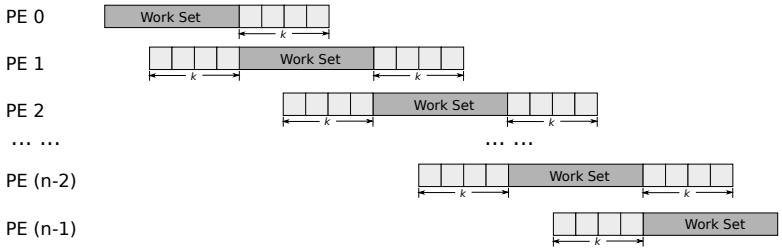


Figure 6.3: Jacobi 1D Stencil

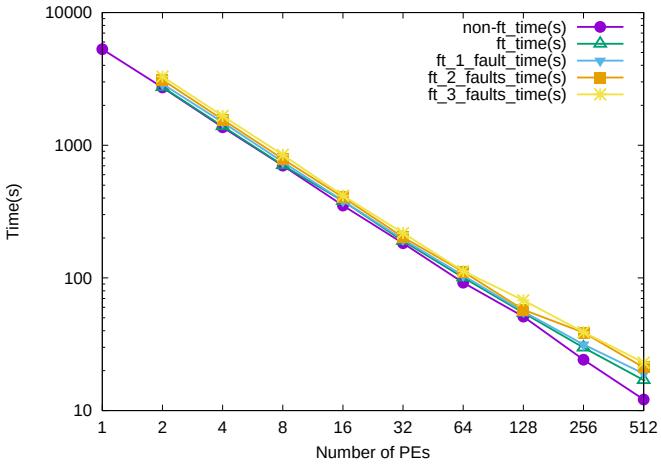
We wrote a Jacobi 1D stencil code to use OpenSHMEM and the proposed API to evaluate our fault tolerance model. Stencil codes perform a sequence of sweeps through a given array. In each time-step the stencil code updates all array elements. Our Jacobi kernel updates element cell value with the old value and its left and right neighbors, i.e, a window size three average operation. All elements need to update m times which is a user input parameter. The algorithm splits the whole array into segments and distributes them to different PEs. Each PE operates on a subset of the whole array with size $N/numpes$, where N is the size of the array and $numpes$ is a number of PEs in the system. For every elemental computation a value from their left neighbor and right neighbor is required. To update the border elements PEs need to retrieve neighboring PEs' border value to form a complete window. The *shmem_get()* operation is used after synchronization to retrieve that value because different PEs need to synchronize processing steps to avoid using a step mismatched value.

At every step the border cell will consume k size buffer, hence after k rounds the

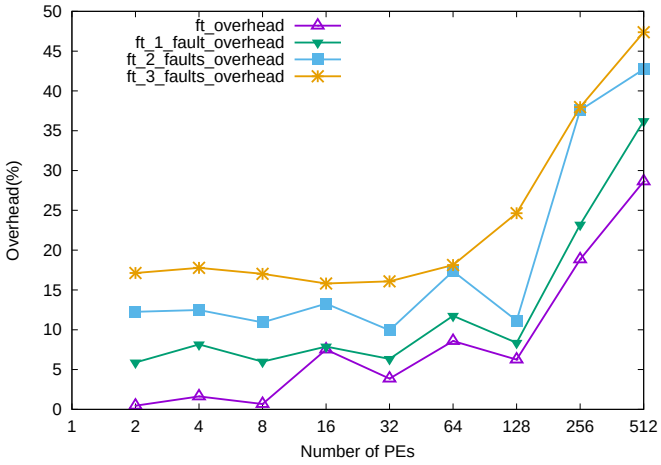
border buffer is completely consumed. After that, all PEs use *shmem_barrier_all()* to synchronize, then retrieve k sized buffers from neighboring PEs to perform the next round of computation. The computing will stop after total m rounds finished. Fig. 6.3 illustrates the buffer layout. We used a 64MB stencil with 4096 iterations and k=256 as buffer size in all tests. Each time the buffer is used up, we call *shmem_checkpoint_all()* once to back up the symmetric memory. Thus we have $4096/256 = 16$ checkpoints.

6.4 Performance Testing Results

Fig. 6.4a is the time plot of Jacobi 1D program, with *non-ft* version, *ft* version with 0, 1, 2 and 3 faults. In log-scaled time axis these five plots are close to parallel straight lines, that implies that the fault tolerance model overhead is stable based on the problem size. Fig. 6.4b shows the fault tolerance overhead as a percentage of the execution time of the Jacobi 1D program. We notice that depending on the number of PEs, for a fixed symmetric heap size, our fault tolerance overhead ranges from 0.4-45%, and most of the overhead is from recomputing time T_s.



(a) Jacobi 1D runtime with *non-ft* and *ft* as a function of a number PEs with 0, 1, 2, and 3 faults



(b) Jacobi 1D fault tolerance overhead with *non-ft* and *ft* as a function of a number PEs with 0, 1, 2, and 3 faults

Figure 6.4: Jacobi 1D runtime and overhead

Chapter 7

FutureWork

The model described in this thesis defined a clear procedure and a set of APIs to enable users exploiting fault tolerance features for OpenSHMEM. However, for the implementations, it leaves flexibility and undefined spaces. For example, in current implementation, we use a ring topology for PE backup target, but it is not optimal from both performance and stability perspective.

The future work of this topic will mainly around the topology and memory checkpoint/backup strategy.

7.1 Dynamic creation for substitute PE

In current implementation, substitute PEs are created with "Reserved" mode. We create more PEs than what really need during start up. Although these PEs are idle

for CPU resource, they still preallocate a fairy amount of memory. With dynamic creation for substitute PEs, we can reduce the memory waste. To support this feature, we need to have cooperation between the runtime (process starter) and the library. Runtime tools need to accept the request from OpenSHMEM library and create new process while the library need to have the ability to accept new process and perform correct initialization on it, including allocating and symmetrizing memory, assign PE number and setup communication endpoint.

7.2 Smart backup target chosen with architecture awareness

When node level failure happened, all PEs on this node will fail. If any failed PE chose a backup target on the same node, recovery will be impossible. To solve this problem, PEs need to have the knowledge of PE distributions on the running architecture. Based on this knowledge, PE can choose its backup target locate on a different node. The recommended implementation is to use hwloc[12] tools to retrieve these information. Fig.7.1 shows the optimal topology of the target choosing.

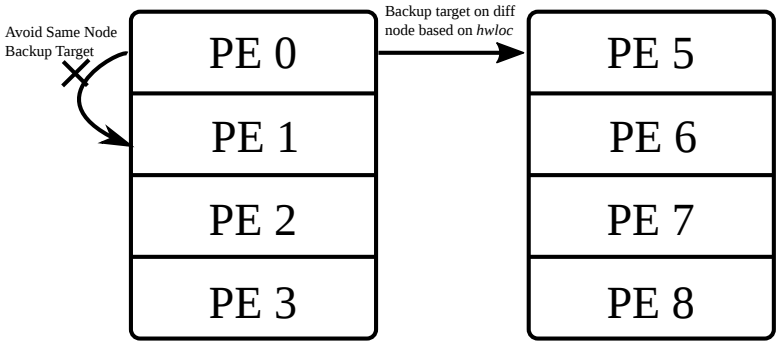


Figure 7.1: Choose target on different node based on hwloc information

7.3 Optional Permanent storage of program status

The model proposed in this thesis work is based on in-memory backup for symmetric data. Another traditional method for fault recovery is to dump staging status of program to the hard drive by which user can continue their program after node recovery. The drawback is that frequent disk writing will cause huge overhead. Thus this feature can be provided as an optional one to users, they can tailor their program for better granularity to perform staging persistence.

The format for backup should also be considered, there is no guarantee that memory layout of variable can be consistent with the system status before failure. One solution to this problem is to use key value pairs for global variable and key offset pair for symmetric heap variable.

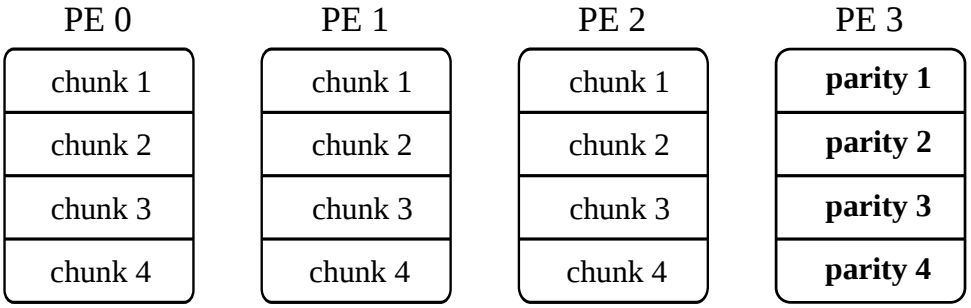


Figure 7.2: RAID4-like backup

7.4 RAID like backup mechanism

RAID-like backup mechanism is a way to overcome the problem of massive memory usage problem in this thesis work. In this backup mode, symmetric memory are separated to chunks with same size, Fig. 7.2. The simplest implementation would be RAID-4, in which case $n-1$ PEs doing work as the usual PE, the other one PE calculate XOR value of all the other PEs and store it to symmetric position. If any PE failure happened, substitute PE can recover symmetric memory from the other $n-1$ PEs. RAID4-like implementation has a main drawback, the PE store check bit need to communicate with all other $n-1$ PEs during checkpointing stage. It can be the bottleneck of the program and prone to failure.

RAID5-like memory backup is also an option to implement. In RAID5, the chunks with same offsets on different disks form a group, one of them served as parity check bit storage. For each next chunk, the disk who took charge of parity check would change to its next neighbour, Fig. 7.3. This can balance the communication overhead to all PEs. But in OpenSHMEM, similar memory layout is not possible for

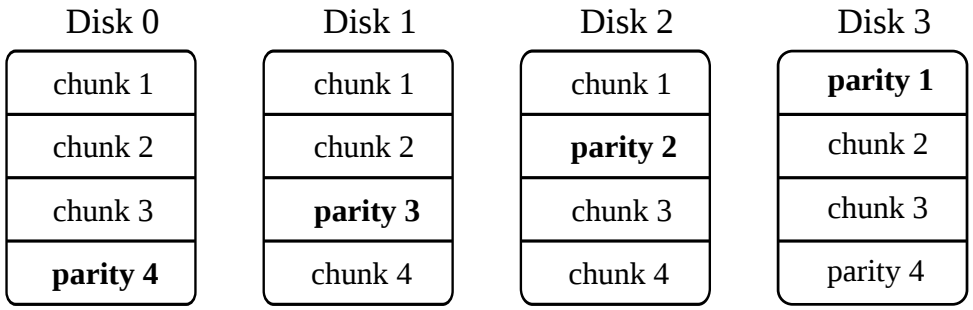


Figure 7.3: RAID5

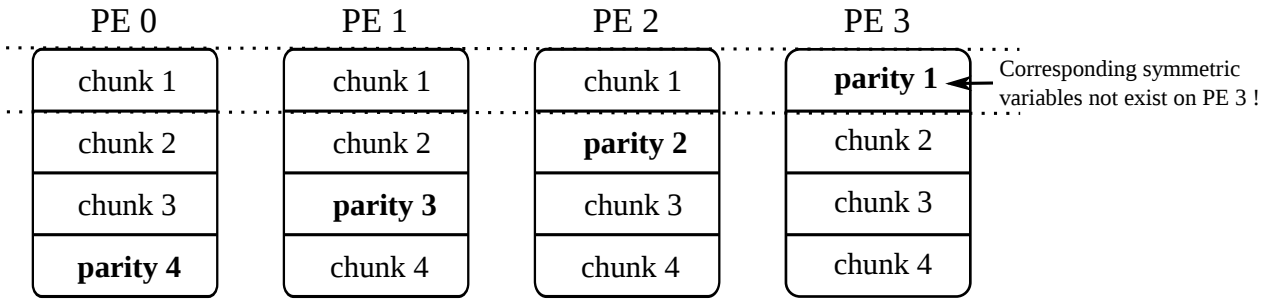


Figure 7.4: RAID5-like not work for symmetric variables

implementation. Because for current SHMEM implementation, symmetric variable is strictly require the symmetric offset for specific variable. A PE would lose its variable access due to the fact that corresponding already used as parity check, Fig. 7.4.

A solution to this problem is to use append mode for parity check, Fig. 7.5. In this mode, symmetric variables still effective and parity check can be performed in checkpointing stage and write to the dedicated position.

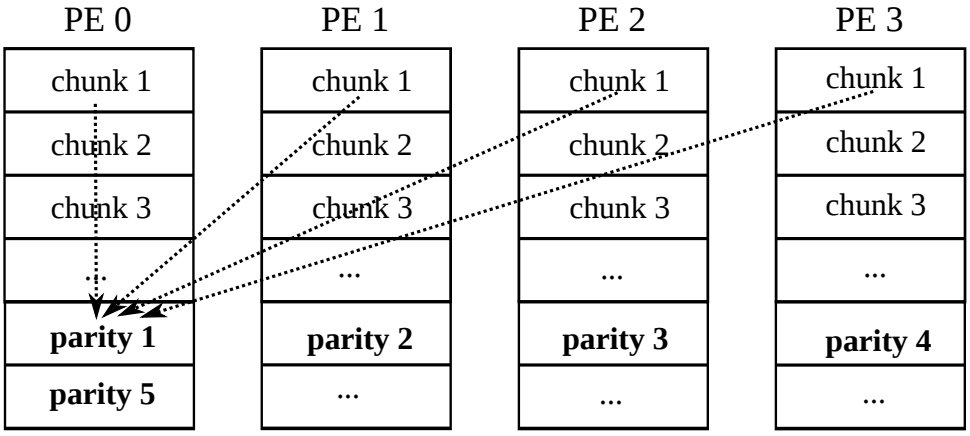


Figure 7.5: RAID5 append model backup

Chapter 8

Conclusion

In this thesis, we investigated the fault tolerance in HPC, from automatic fault tolerance, ABFT and ULFM. We also introduced the PGAS and OpenSHMEM programming model. Based on this model, we proposed an OpenSHMEM fault tolerance model. We investigated different types of faults and developed our checkpointing and restart fault tolerance model for OpenSHMEM. To the best of our knowledge, this is the first fault tolerance model implemented for OpenSHMEM. We provide a prototype API and evaluate based on the overhead observed using a Jacobi stencil code. We found that the checkpointing and restart mechanism will add about 0.4-45% overhead when compared with the non-ft version. For long running programs the overhead of checkpointing is dependent on the frequency of checkpointing.

We also talked abut the potential future work based on current progress. Including smart backup target choosing and RADI-like checkpointing implementation, to improve the stability and memory usage of current implementation. Based on these

improvement, the fault tolerance model will provide better scalability and mitigation features.

Bibliography

[1] A. M. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 167–176. IEEE, 1999.

[2] A. Aiken, P. Colella, D. Gay, S. Graham, P. Hilfinger, A. Krishnamurthy, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato, et al. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10:11–13, 1998.

[3] M. M. Ali, J. Southern, P. Strazdins, and B. Harding. Application level fault recovery: using fault-tolerant open mpi in a pde solver. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1169–1178. IEEE, 2014.

[4] N. Ali, S. Krishnamoorthy, N. Govind, and B. Palmer. A redundant communication approach to scalable fault tolerance in PGAS programming models. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 24–31. IEEE, 2011.

[5] W. Bland, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. A proposal for user-level failure mitigation in the mpi-3 standard. *Department of Electrical Engineering and Computer Science, University of Tennessee*, 2012.

[6] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of user-level failure mitigation support in mpi. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 193–203, Berlin, Heidelberg, 2012. Springer-Verlag.

[7] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. In *Euro-Par 2012 Parallel Processing*, pages 477–488. Springer, 2012.

[8] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[9] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. *International Journal of High Performance Computing Applications*, 28(2):210–224, 2014.

[10] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[11] A. Bouteiller, P. Lemarinier, G. Krawezik, et al. Coordinated checkpoint versus message log for fault tolerant mpi. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 242–250. IEEE, 2003.

[12] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[13] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols. *Future Generation Computer Systems*, 24(1):73–84, 2008.

[14] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[15] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[16] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[18] S.-E. Choi and S. J. Deitz. Compiler support for automatic checkpointing. In *High Performance Computing Systems and Applications, 2002. Proceedings. 16th Annual International Symposium on*, pages 213–220. IEEE, 2002.

[19] O. Community. Openshmem application programming interface, 2014.

[20] U. Consortium et al. Upc language specifications v1. 2. *Lawrence Berkeley National Laboratory*, 2005.

[21] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 108–115. IEEE, 1996.

[22] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, pages 162–171. ACM, 2011.

[23] G. Deconinck and R. Lauwereins. User-triggered checkpointing: system-independent and scalable application recovery. In *Computers and Communications, 1997. Proceedings., Second IEEE Symposium on*, pages 418–423. IEEE, 1997.

[24] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[25] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[26] D. Eachempati, H. J. Jun, and B. Chapman. An open-source compiler and runtime implementation for coarray fortran. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 13. ACM, 2010.

[27] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[28] G. E. Fagg and J. J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Recent advances in parallel virtual machine and message passing interface*, pages 346–353. Springer, 2000.

[29] G. E. Fagg and J. J. Dongarra. Building and using a fault-tolerant mpi implementation. *International Journal of High Performance Computing Applications*, 18(3):353–361, 2004.

[30] S. Ge, D. Eachempati, D. Khaldi, and B. Chapman. An evaluation of anticipated extensions for fortran coarrays. In *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*, pages 47–58. IEEE, 2015.

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

[32] W. Gropp and E. Lusk. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[33] P. Hao, P. Shamis, M. G. Venkata, S. Pophale, A. Welch, S. Poole, and B. Chapman. Fault tolerance for openshmem. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 23:1–23:3, New York, NY, USA, 2014. ACM.

[34] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[35] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[36] https://www.olcf.ornl.gov/titan/. Titan Super-Computer, 2013.

[37] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[38] J. Hursey and R. L. Graham. Preserving collective performance across process failure for a fault tolerant mpi. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1208–1215. IEEE, 2011.

[39] J. Hursey, R. L. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. G. Solt. Run-through stabilization: An mpi proposal for process fault tolerance. In *Recent Advances in the Message Passing Interface*, pages 329–332. Springer, 2011.

[40] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[41] D. Lea. Doug leas malloc (dlmalloc).

[42] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[43] S. Pauli, M. Kohler, P. Arbenz, P. Arbenz, and P. Arbenz. A fault tolerant implementation of multi-level monte carlo methods. In *PARCO*, volume 13, pages 471–480, 2013.

[44] S. Pophale, O. Hernandez, S. Poole, and B. M. Chapman. Extending the openshmem analyzer to perform synchronization and multi-valued analysis. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 134–148. Springer, 2014.

[45] S. S. Pophale. Src: Openshmem library development. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 374–374, New York, NY, USA, 2011. ACM.

[46] F. Shahzad, M. Kreutzer, T. Zeiser, R. Machado, A. Pieper, G. Hager, and G. Wellein. Building a fault tolerant application using the GASPI communication layer. *CoRR*, abs/1505.04628, 2015.

[47] P. Shamis, M. G. Venkata, S. Poole, A. Welch, and T. Curtis. Designing a high performance openshmem implementation using universal common communication substrate as a communication middleware. In *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, OpenSHMEM 2014, pages 1–13, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[48] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 526–531. IEEE, 1996.

[49] C. Team. Containment domains c++ api v0.1. http://lph.ece.utexas.edu/users/CDAPI, March 2014.

[50] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al. NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[51] S.-J. Wang and N. K. Jha. Algorithm-based fault tolerance for fft networks. *Computers, IEEE Transactions on*, 43(7):849–854, 1994.