UNIVERSITY OF CALIFORNIA
RIVERSIDE

FT-RT-TDDFT: Fault Tolerant Real-Time Time-Dependent Density Functional Theory on High Performance Computing Systems

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Vineeth Bhaskar Suvarna

March 2024

Thesis Committee:
Dr. Zizhong Chen, Chairperson
Dr. Yan Gu
Dr. Daniel Wong

The Thesis of Vineeth Bhaskar Suvarna is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my academic advisor, Professor Zizhong Chen, whose support has been instrumental in the completion of this thesis.

To my family for all the support.

ABSTRACT OF THE THESIS

FT-RT-TDDFT: Fault Tolerant Real-Time Time-Dependent Density Functional Theory on High Performance Computing Systems

by

Vineeth Bhaskar Suvarna

Master of Science, Graduate Program in Computer Science
University of California, Riverside, March 2024
Dr. Zizhong Chen, Chairperson

HPC systems are continuously experiencing exponential growth in their scale. The issue of fault tolerance in these systems is becoming increasingly important for applications like Real-Time Time-Dependent Density Functional Theory (RT-TDDFT) that run for extended periods. Checkpoint - restart is a common method to achieve fault tolerance in HPC systems. In this thesis, we analyze the performance of single file checkpoint-restart implementation in RT-TDDFT where data is collectively checkpointed to a single file, and find that storing the checkpoints in persistent storage adds significant performance overhead. We demonstrate multi-file checkpoint-restart in RT-TDDFT by creating multiple checkpoint files to improve the performance of checkpointing. We further reduce the performance overhead using in-memory checkpoint-restart where checkpoints are stored in-memory instead of persistent storage. We perform a comparative analysis and show that significant performance gains are achieved using multi-file and in-memory checkpoint-restart over single file checkpoint-restart. In this way, we implement multi-

file and in-memory checkpoint-restart for fault tolerant RT-TDDFT on high performance computing systems.

Contents

Contents viii
List of Figures ix
1. Introduction 1
2. Background 2
3. Method 6
   3.1. Single File Checkpoint-Restart 11
   3.2. Multi-File Checkpoint-Restart 18
   3.3. In-Memory Checkpoint-Restart 24
4. Results 33
   4.1. Preliminary Analysis 33
   4.2. Comparative Analysis 36
5. Discussion 43
6. Conclusion 45
Acknowledgments 46
Bibliography 47

List of Figures

Figure 3.1: Single File Checkpoint-Restart 12
Figure 3.2: Multi-File Checkpoint-Restart 19
Figure 3.3: In-Memory Checkpoint-Restart 26
Figure 4.1: Percentage of time spent in writes vs storage methods 34
Figure 4.2: Writing speeds vs Lustre FS stipe count 35
Figure 4.3: Writing speed vs number of nodes 35
Figure 4.4: Profiling result for large samples 36
Figure 4.5: Checkpointing time vs checkpointing frequency 37
Figure 4.6: Program time vs checkpointing frequency 38
Figure 4.7: Percentage time spent in checkpointing 39
Figure 4.8: Recovery time vs checkpoint type 40
Figure 4.9: Percentage time spent in checkpointing for higher data size 41

1. Introduction

Checkpointing and restart is used to provide fault tolerance in high performance computing systems. As the number of processing nodes increases, the probability of failure in these nodes increases. If a failure occurs, applications need to restart the computations from the start thereby wasting computing resources. An application that checkpoints the data can be recovered from the checkpointed point on failure. Checkpointing can consume a significant portion of the execution time due to the time spent in storing the checkpoint on stable storage.

In this thesis, we implement fault tolerance for Real-Time Time-Dependent Density Functional Theory (RT-TDDFT) implementation in Qbox [5] using multi-file and in-memory checkpoint-restart. Qbox is a large-scale parallel implementation of First-Principles Molecular Dynamics that runs on high performance computing systems. It implements a disk-based single file checkpoint wherein all the MPI tasks write their data to a single file on the disk using MPI collective I/O. A preliminary analysis of the performance of the checkpointing in HPC systems found writing speed to be the bottleneck and hence we focused on improving the writing speeds. We reduced the bottleneck by multi-file checkpoint-restart for fault tolerant RT-TDDFT. Instead of writing a single large file, each MPI task wrote its data into separate files using POSIX I/O instead of MPI collective I/O to the disk which improved the performance of checkpointing. Writing a file to a disk still introduces latency and writing overhead. To

reduce this overhead, we then stored the checkpointed data in the memory instead of the disk. We then show a comparative study on the improvement of checkpointing performance of in-memory checkpoint-restart and multi-file checkpoint-restart with that of single-file checkpoint-restart in RT-TDDFT on high performance computing systems.

2. Background

The usage of HPC systems is continuously increasing due to the increasing need for computing power. These systems come with their fair share of problems [7] and one of them being fault tolerance. They can go through hardware or software failures as the applications running in these systems are long running, sometimes running days or months. A failure could mean that days of computations could be lost, and the computations must start over again. These applications are usually written in MPI (Message Passing Interface) is used to write parallel software and a failure in MPI results in the application to completely stop and abort all the operations.

Due to these reasons, fault tolerance is an important feature in high performance systems. Different methods of fault tolerance techniques approaches are used to achieve this. Proactive fault tolerance involves taking corrective action before failure occurs. This requires us to predict the failure and take appropriate action to avoid the failure. Failure prediction algorithms or hardware sensors [49] can be used to predict the failure. RAS log file or Reliability, Availability, and Serviceability log files are generally used to implement prediction algorithms for prediction [48]. However, false negatives of these

algorithms can have a very high impact. After failure is detected, corrective action could be migration methods like process-level migration and VM migration [49]. Process-level migration [52] involves the transfer of a process from one node to another on failure whereas VM migration involves the transfer of a VM from one node to another [50,51]. Redundancy can also be used for fault tolerance. Hardware redundancy is created by adding more physical components that can be used on failure. This helps in providing fault tolerance for hardware failures. Having redundant is a form of software redundancy where one behaves as an active process and others as a passive process. ABFT or Algorithm based fault tolerance [22,24,25] is to detect and correct failures that happened during computation after the termination of the computation. ABFT usually is not applicable to all algorithms, but the overhead is low when it applies. Offline ABFT algorithms detect and correct the failure after the program has completed execution; however, this may not work if the number of failures exceeds a particular threshold [10]. Online ABFT algorithms [16, 33] correct the failures dynamically during execution. This is more useful for long running HPC applications where failure rates are high. Recovery methods are also employed for fault tolerance. Recovery can consist of either forward recovery [10] or rollback recovery. Recovery [1] is to bring the system to a consistent state after a failure. Forward recovery is to bring the system to a new state so it can continue without repeating the computation. This method tries to let the systems recover by on its own to a normal execution and depends on algorithms that are designed for such cases [11]. On the other hand, rollback recovery consists of checkpointing, failure

detection, and recovery. Checkpointing stores the state of the program from which the program can recover after failure.

There are two types of rollback-recovery methods that can be used: Checkpointing-based and Log-based. Checkpointing based rollback-recovery based fault tolerance involves going back to the most recent consistent state. Since it is a distributed system, different processes could be at different stages of computation and hence there is a need to find a consistent global previous state [11]. Protocols are designed depending on the way checkpointing is carried out. In uncoordinated checkpointing, each process checkpoints at its own feasible point, and hence it can be difficult to find a globally consistent state during recovery [21]. Each process can checkpoint when it is feasible or when the state to be stored is minimal. Dependency among the processes can be used to find the point to recover from. However, this could cause excess storage use to multiple checkpoints or the recovery point could reach initial state called domino effect [45]. In coordinated checkpointing [20], all the processes synchronize the checkpoints. This way all the processes maintain the state at the same point and recovery becomes easier. Algorithms like Chandy and Lamport's algorithm are used [54]. Log-based rollback recovery is similar to checkpoint-based rollback recovery just that here the changes are recorded in a log [45]. The log is called the determinant, and the process is recovered using a checkpoint and then applying the determinants. We focus on two checkpoint-based rollback-recovery methods, multi-file and in-memory checkpoint-restart in our thesis.

Checkpoint-based rollback-recovery can done at different levels [1,3]. System-level checkpointing involves the checkpointing being performed at the system-level and it is transparent to the user and no change is required in the application [55]. It can be implemented in OS kernel or hardware level. However, kernel-level source code might not always be available, and the implementation might not be portable to other platforms. Hardware-level checkpointing is done by building additional hardware that is transparent to the user. BLCR [47] is an example of system-level checkpointing. User-level checkpointing uses a library that is linked to the application [56]. Hence this method is not transparent. A commonly known library implementation is libckpt [9]. In application level checkpointing [3], the checkpointing is implemented in the source code by the programmer [46]. This gives the flexibility to checkpoint only the required data that is needed to restart the application at the time when it is most feasible. It can run on heterogeneous systems but is not transparent to the user. The checkpointed data is stored in the persistent storage and the program is restarted from the checkpoint when a failure is detected. Implementation of checkpoints in the application requires a good understanding of the application by the implementer and it needs to be done for every application that needs to be checkpointed. However, it gives high flexibility to implement checkpointing and the checkpointing can be implemented in a way that works best for the intended application. We implement application-level checkpoint-restart for fault tolerant RT-TDDFT.

Research has shown that writing checkpoints on a stable storage is the dominant cost in checkpointing [8, 53]. Various techniques are designed to reduce this overhead [1,3]. Incremental checkpointing, checkpointing is done in increments and only the changed portion is checkpointed [10, 9]. The part that is not changed is recovered from the previous checkpoints. Techniques like page-level incremental checkpoint [57] where page-faults are used to find modified pages. Compilers can be instrumented to add checkpointing code to the source code during compilation. One of the projects that implements this is C3 [53] which automates checkpointing for MPI programs. In forked-checkpointing [9], parent process forks a child process. The child process carries out checkpointing and the parent process continues the execution. Other techniques like memory exclusion checkpointing [59], diskless checkpointing [60], and compression checkpointing [61] are also used to reduce the overhead.

In this thesis, we implement application-level checkpoint-restart for fault tolerant RT-TDDFT in HPC systems. We implement two new approaches of checkpoint-restart, starting from existing single file checkpoint-restart implementation we implement multi-file and in-memory checkpoint-restart and demonstrate the performance benefits of the two approaches.

3. Method

The architecture of Qbox [6] is designed to run in thousands of processors to perform first-principles simulations. Qbox implements First-Principles Molecular Dynamics

(FPMD) [11] an atomistic simulation method and a large part of the computation involves computing the electronic structure. This computation is done using Density Functional Theory. During simulation, one-particle electronic wavefunctions are calculated using Kohn-sham equations. This calculation involves linear algebra and Fourier transforms. The wavefunction Fourier coefficients are block-distributed on a two-dimensional grid. The third and fourth levels of data distribution include k-points and spins. Hence the wavefunction coefficients are distributed in a 4-dimensional grid.

The Sample data structure in Qbox represents the data needed to simulate the physical system being simulated in Qbox [12]. It includes the set of atoms and species. It also includes the wavefunctions that consist of multiple slater determinants all distributed in the 2D structure. The wavefunction matrix is of the form: (spins * k-points * bands * plane wave basis). Here, (bands * plane wave basis) represents the slater determinants. This matrix is distributed in a 4D grid of MPI tasks. All the sample information is needed to be present in the checkpointed data for proper recovery during a failure.
The algorithms used to modify the samples are called steppers [5]. We experiment checkpoint-restart on Real-time Time-dependent density functional theory stepper. As shown in Algorithm 3.1, it is an iterative algorithm that updates the wavefunction on every iteration. Every iteration requires the wavefunction in the present state and the wavefunction in the previous iteration (Line 7) to generate the updated wavefunction. Because of this, we need to store two samples (Line 6, 16) having the given wavefunctions during checkpointing for recovery.

Algorithm 3.1:

```
1.  run_rt_sample_stepper(sample)
2.  {
3.      for (rtiter)
4.      {
5.          compute1();
6.          checkpoint(sample_prev);
7.          wavefunction_prev = sample_prev.wavefuntion;
8.          compute2(sample_prev, wavefunction_prev);
9.          for(rtitscf)
10.         {
11.             for(rtite)
12.             {
13.                 compute3();
14.             }
15.         }
16.         checkpoint(sample);
17.     }
18. }
```

The number of iterations required is defined by the parameters ritter, rtitscf, and rtite. For recovery as shown in Algorithm 3.2, we need to read both the samples. We first read the previous sample and stored the wavefunction and then read the present sample to restart the computation.

Algorithm 3.2:

```
1. sample = recover(sample_prev)
2. wavefunction_prev = sample.wavefunction;
3. sample = recover(sample)
4. run_rt_sample_stepper(sample, wavefunction prev)
```

Failure was simulated using MPI_Allreduce (Line 7) where all the MPI tasks send their status whether they are healthy or not as shown in Algorithm 3.3. If a task fails at a particular iteration, it is detected by all the tasks. Once a failed task is detected, all the tasks restart their computation from the last checkpointed state (Line 10-14).

Algorithm 3.3:

```
1. if(iter == failure_iter && rank() == failure_rank)
2. {
```

```
3.          Sample.rt_failure = true;
4.    }
5.
6.    bool failure = false;
7.    MPI_Allreduce(&rt_failure, &failure, 1, MPI_C_BOOL, MPI_LOR,
8.          MPI_COMM_WORLD);
9.
10.   if(failure)
11.   {
12.         Sample.rt_restart = true;
13.         return;
14.   }
```

Qbox keeps the sample on which simulations are performed in a single XML file using MPI collective I/O and these sample files could increase in sizes eventually causing I/O bottleneck. We instead first break the single file into multiple files and write using POSIX I/O to further parallelize the writes and improve I/O performance and then we forgo disk writing and instead store the data in the memory thereby further improving the performance.

We talk about the single file checkpoint-restart techniques of checkpointing implemented in RT-TDDFT. Then we talk about how we implemented multi-file and in-memory

checkpoint-restart technique in the same. We ran the iterative RT-TDDFT algorithm where the data was checkpointed between iterations. Failure was simulated using MPI_Allreduce where all the MPI tasks send their status whether they are healthy or not. If a task fails at a particular iteration, it is detected by all the tasks. Once a failed task is detected, all the tasks restart their computation from the last checkpointed state.

3.1. Single File Checkpoint-Restart

Single File Checkpoint-Restart uses the SampleWriter and SampleReader implementation present in the QBox source code. Data is checkpointed as a single file by all the MPI tasks collectively using MPI_File_write_at_all. Before writing the data, the data is shuffled using MPI_Alltoallv so that all the MPI tasks now hold the data that it is supposed to write. The file is written to Lustre File System using MPI collective I/O where the stipe count is kept according to the size of the checkpointing file. On failure, the programs recover to the previous checkpointed state. All the tasks clear the present data and start reading the checkpointed file. The file is read in parallel by all MPI tasks and reshuffled using MPI_Alltoallv. The computation then resumes from the last checkpointed state.

Figure 3.1 shows the flow of single file checkpoint-restart. PE*n* is the MPI task where n denotes the rank of the task each holding data D*n*. All the tasks checkpoint their data as a single file denoted as C_D in the disk storage.

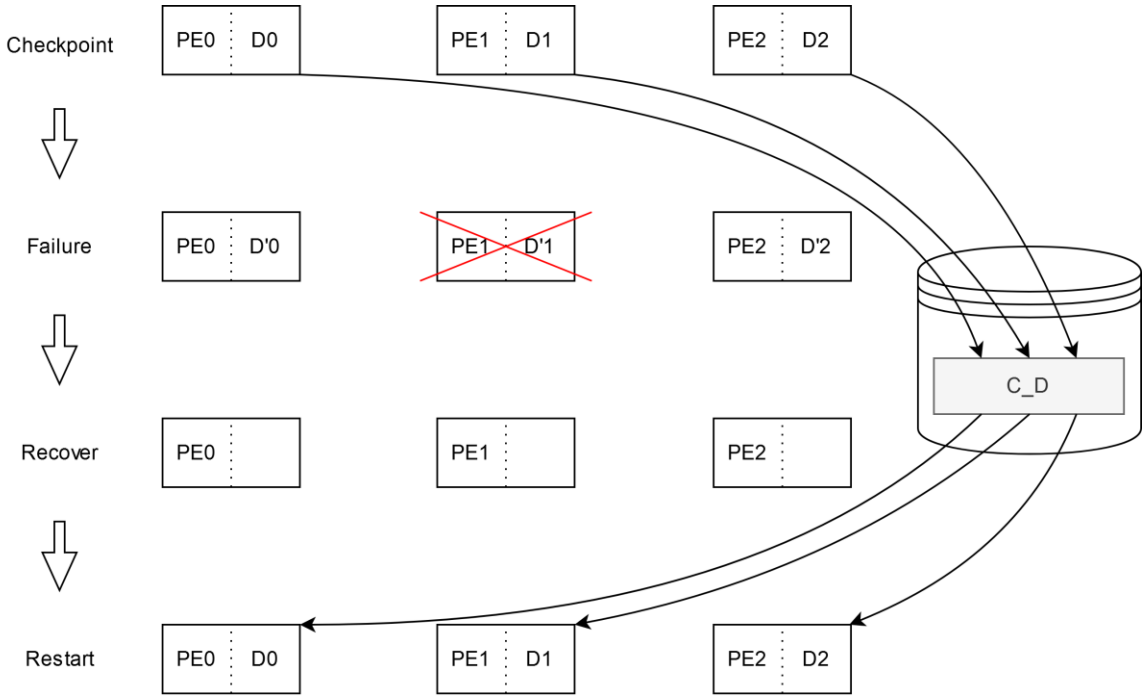

Figure 3.1: Single File Checkpoint-Restart

We used Perlmutter's Scratch, an all-flash Lustre file system. On failure, all the tasks go through recovery where they clear the modified data after checkpointing present in the memory denoted in the figure as D'*n*. All the tasks then read the checkpointed data C_D where each task then holds the previous checkpointed data D*n* and then the computation is continued from the previously checkpointed state.

The following algorithm shows how sample writing in single file checkpointing is performed.

Algorithm 3.4:

```
1.  checkpoint(sample)
2.  {
3.      MPI_File fh;
4.      MPI_File_open(sample_file_name, &fh);
5.      SharedFilePtr sfp(fh);
6.      if(rank == 0)
7.      {
8.          // data written in XML format
9.          write_header_data(sfp);
10.         write_atom_data(atom);
11.     }
12.     // gather distributed 4D wavefunction matrix
13.     vector<vector<string> > sdstr; // serialized data of local data present
14.     for (spin_local)
15.     {
16.         const int ispin = isp_global(isp_loc);
17.         for (kpoint_local)
18.         {
19.             // serialize
```

```
20.                     // slater_determinant[spin_local][kpoint_local]
21.                     // into sdstr[spin_local][kpoint_local]
22.                     st_sting;
23.                     for(band_local in slater_determinant[spin_local][kpoint_local])
24.                     {
25.                             //perform 3D fft_backward
26.                             fft_backward(band_local);
27.                             // redistribute segments between processes;
28.                             MPI_Alltoall(band_local);
29.                             st_string += serialize(band_local);
30.                     }
31.             sdstr[spin_local][kpoint_local] = st_sting;
32.             // store serialized data in XML form;
33.             }
34.     }
35. }
36. // sdstr now contains all data to be written
37. for(global_spin)
38. {
39.     for(global_kpoint)
40.     {
41.             MPI_Barrier(MPI_COMM_WORLD);
```

```
42.             offset = 0;
43.             spin_local = get_local_spin(global_spin);
44.             kpoint_local = get_local_kpoint(global_kpoint);

45.             if(spin_local>=0 && kpoint_local>=0)
46.             {
47.                     // compute offset of data on current task
48.                     offset = MPI_Scan(local_size, local_offset)
49.             }
50.             MPI_File_write_at_all(sfp.file(),offset,sdstr[spin_local][kpoint_local]);
51.     }
52. }
```

Algorithm 3.4 starts with a shared file pointer and initially rank 0 writes the metadata and the atom and species data since this data is common across all the MPI tasks (Line 6-11). Then the 4D distributed wavefunction is written. First, every task serializes its own share of local data into a string by iterating in two-dimensions: spin and kpoint (Line 14-35). For each band within the slater determinant a 3D FTT backward is performed in coordination with all the tasks (Line 26). The data segments are redistributed among the involved tasks using MPI_Alltoall (Line 28) and then serialized into a string. After this process, all the tasks contain the data they are supposed to write. Now all the tasks write into the file in coordination. All the tasks again iterate in two dimensions with a

MPI_Barrier before every iteration (Line 41). The tasks which hold the data for the specific spin and kpoint (Line 45) concurrently write into the file by first calculating the file offset for themselves using their local offset size and MPI_Scan of all the offsets (Line 48). Then all the tasks write into the file MPI_File_write_at_all (Line 50).

Recovery takes place in a similar approach as shown in the following algorithm:

Algorithm 3.5:

```
1.  restart(sample_file_name)
2.  {
3.      // All the tasks read the sample in parallel
4.      file infile(sample_file_name);
5.      // Create a distributed 2D matrix to store parsed wavefunction data
6.      DoubleMatrix gfdata(gctxt);
7.      // all the processes read their share of data from string to numerical values
8.      XMLProcess(infile, gfdata);
9.      // All the tasks read the atom and species data and store it in the sample
10.     // Wavefunction handler reads the data from gfmatrix and performs redistribution
11.     AtomSetHandler(sample, infile)
12.     vector<complex<double> > wftmp
13.     for(global_spin)
```

```
14.     {
15.         for(global_kpoint)
16.         {
17.             MPI_Barrier(MPI_COMM_WORLD);
18.             offset = 0;
19.             spin_local = get_local_spin(global_spin);
20.             kpoint_local = get_local_kpoint(global_kpoint);
21.
22.             if(spin_local>=0 && kpoint_local>=0)
23.             {
24.                 for(band_local)
25.                 {
26.                     // redistribute data from gfdata to wftmp
27.                     MPI_Alltoallv(&gfdata, &wftmp)
28.                     fft_forward(slater_determinant[spin_local]
29.                         [kpoint_local][band], wftmpr);
30.                 }
31.                 MPI_Barrier(MPI_COMM_WORLD);
32.             }
33.         }
34.     }
35. }
```

We see in Algorithm 3.5 that the checkpointed file is read in parallel by all the tasks (Line 4) and the atom and species data is parsed (Line 11) and stored in the sample. The wavefunction data is parsed from the XML file and then stored in a distributed matrix gfdata initially (Line 8). The 4D distributed wavefunction matrix of the sample is filled from the gfdata. The wavefunction is iterated in two dimensions: spin and kpoint having a barrier after every iteration. Each band is then iterated where gfdata for that band is redistributed and into wftmp which is a 2D grid (Line 27) and forward FFT is performed to change the data from real basis to Fourier basis and then saved in the 4D wavefunction grid (Line 28).

Thus, we see how checkpoint and restart were performed using a single file in RT-TDDFT code.

3.2. Multi-File Checkpoint-Restart

For multi-file checkpoint-restart, every MPI task writes its own data as a separate file. Each task writes its own data as a separate file in the Lustre File System using POSIX I/O having stripe length 1. Here, reshuffling of data is not required as each task is just writing its own data. On failure, the programs restart to the previous checkpointed state like the single file checkpoint-restart approach. The task clears their present data and then starts reading their checkpointed files. On completion, the MPI tasks resume their computation.

Multi-file checkpoint-restart is demonstrated in Figure 3.2. PE*n* is the MPI task where n denotes the rank of the task each holding data D*n*.

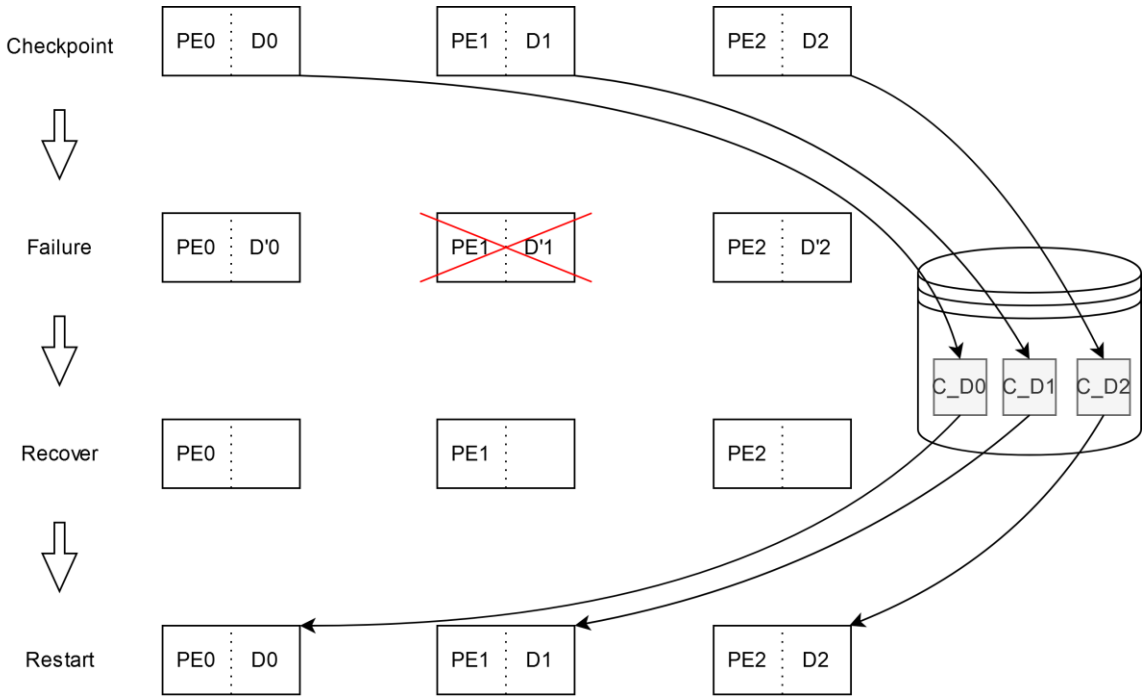

Figure 3.2: Multi-File Checkpoint-Restart

While checkpointing each task stores the data in the disk i.e., Perlmutter's Scratch as a separate file, and hence each task would have its own checkpointed file C_D*n*. On failure, the tasks recover by first clearing their modified data D'*n* and then reading the checkpointed file C_Dn from the disk. The computation then restarts from the previous checkpointed state.

The following algorithm shows how we implement multi-file checkpointing.

Algorithm 3.6:

```
1.  checkpoint(sample)
2.  {
3.      ofstream os;
4.      // each task writes a separate file
5.      os.open(sample_file_name + rank);
6.      // data written in XML format by all the tasks
7.      write_header_data(os);
8.      write_atom_data(os, atom);
9.
10.     for(global_spin)
11.     {
12.         for(global_kpoint)
13.         {
14.             spin_local = get_local_spin(global_spin);
15.             kpoint_local = get_local_kpoint(global_kpoint);
16.             // each task writes only the data local to itself
17.             st_sting;
18.             if(spin_local>=0 && kpoint_local>=0)
```

```
19.                     {
20.                         st_string.clear()
21.                         for(band_local in slater_determinant
22.                             [spin_local][kpoint_local])
23.                         {
24.                             //perform 3D fft_backward
25.                             fft_backward(band_local);
26.                             // redistribute segments between process;
27.                             MPI_Alltoall(band_local);
28.                             st_string += serialize(band_local);
29.                         }
30.                     } else
31.                     {
32.                         st_string.clear();
33.                         // this is needed for recovery
34.                         st_string << “mention meta data for kpoint and spin”
35.                     }
36.             os << st_string;
37.     }
38.     MPI_Barrier(MPI_COMM_WORLD);
39. }
```

Algorithm 3.6 shows the approach for multi file checkpoint. We see that each task creates a separate file for itself (Line 5) and all the tasks write the atom and species information in their own file (line 7 and 8). For writing the wavefunction, every task iterates through the 4D grid and checks if it has the part of the grid being iterated (Line 18). If it has the data for that iteration, each band within the slater determinant a 3D FTT backward is performed in coordination with all the tasks having the data (Line 25). The data segments are redistributed amongst the involved tasks using MPI_Alltoall and then serialized into a string (Line 27 and 28). If the task does not have the data, the metadata of the spin and kpoint is added (Line 34). We see that after this step we do not need to coordinate the write between the processes unlike the single file approach and we used POSIX I/O instead of MPI collective I/O and the checkpointing is completed.

Recovery takes place as shown in the following algorithm:

Algorithm 3.7:

```
1.  restart(sample_file_name)
2.  {
3.      // All the tasks read their own sample checkpointed file in parallel
4.      // Create a local 2D matrix to store parsed wavefunction data
5.      file infile(sample_file_name + rank());
6.      vector<vector<double>> mul_gfdata;
```

```
7.  // all the processes read their data from string to numerical values
8.  XMLProcess(infile, mul_gfdata);
9.  // All the tasks read the atom and species data and store it in the sample
10. AtomSetHandler(sample, infile)
11. // Wavefunction handler reads the data from mul_gfdata
12. for(global_spin)
13. {
14.         for(global_kpoint)
15.         {
16.                 MPI_Barrier(MPI_COMM_WORLD);
17.                 spin_local = get_local_spin(global_spin);
18.                 kpoint_local = get_local_kpoint(global_kpoint);
19.                 for(global_band)
20.                 {
21.                         band_local = get_local_band(global_band)
22.                         if(band_local)
23.                         {
24.                         fft_forward(slater_determinant[spin_local]
25.                                 [kpoint_local][band], mul_gfdata[spin_local]
26.                                 [kpoint_local][band_local]);
27.                         }
28.                         MPI_Barrier(MPI_COMM_WORLD);
```

```
29.                         }
30.                 }
31.         }
32.  }
```

We see in Algorithm 3.7 that every file reads its own checkpointed file. The wavefunction data is parsed from the XML file and then stored in a local matrix mul_gfdata initially (line 5 and 8). The 4D distributed wavefunction matrix of the sample is filled from the mul_gfdata. The wavefunction is iterated in two dimensions: spin and kpoint having a barrier after every iteration. Here, every band is iterated and only if the band is part of the local band of the process (Line 19 and 22), the process carries out FFT transformation. All these steps are synchronized using MPI_Barrier (Line 28).
Hence, we saw how we implemented multi-file checkpoint-restart for fault tolerant RT-TDDFT.

3.3. In-Memory Checkpoint-Restart

In In-Memory Checkpoint-Restart, every MPI task stores a copy of its data in its own memory. Along with storing its own data, each task also stores the data of the previous task i.e. task *i* will store data of task *i –1*. This sending and receiving of data is performed using MPI_Sendrecv. This is because, on failure, the failed task will lose the data present in its memory. The healthy tasks can recover from the data already present in their own memory. The failed task *i* will recover its data from its data stored in adjacent *i + 1* tasks

memory by using MPISend and MPI_Recv and store the data in its own memory. It will also store its adjacent task *i – 1* data in its memory to maintain a consistent state. Once this is done, all the tasks can recover by reading the data from its own memory and continue the computation.

Figure 3.3 shows the working of in-memory checkpoint-restart. PE*n* are the MPI tasks where n denotes the rank of the task each holding data D*n*. During the checkpointing phase, each task creates a checkpoint of its own data that will be present in its memory. It is represented as C_D*n* for task PE*n*. After that, every task sends the same checkpointed data to the next task to be stored in its memory represented as C_D*(N-1)* for task PE*n*. Thus, every MPI task PE*n*, after checkpointing, has its data D*n,* and checkpointed data C_DN and C_D*(N-1)*. On failure, the failed task loses all the data present in the memory including the checkpoints stored. During recovery, all the healthy tasks clear their own modified data D'*n* and the failed task does not have any data due to failure. During restart, the healthy tasks recover from reading from their own checkpointed data in memory. The failed task goes through a different process. Assuming that PE1 task failed, PE1 receives its checkpointed data C_D1 from PE2, stores the checkpointed data in its own memory as C_D1, and then recovers the data from the checkpointed data from its memory as D1. PE1 also receives PE0's checkpointed data C_D0 and stores it in its memory. This way we have a consistent state after restart and the computation continues from the previous checkpointed state.

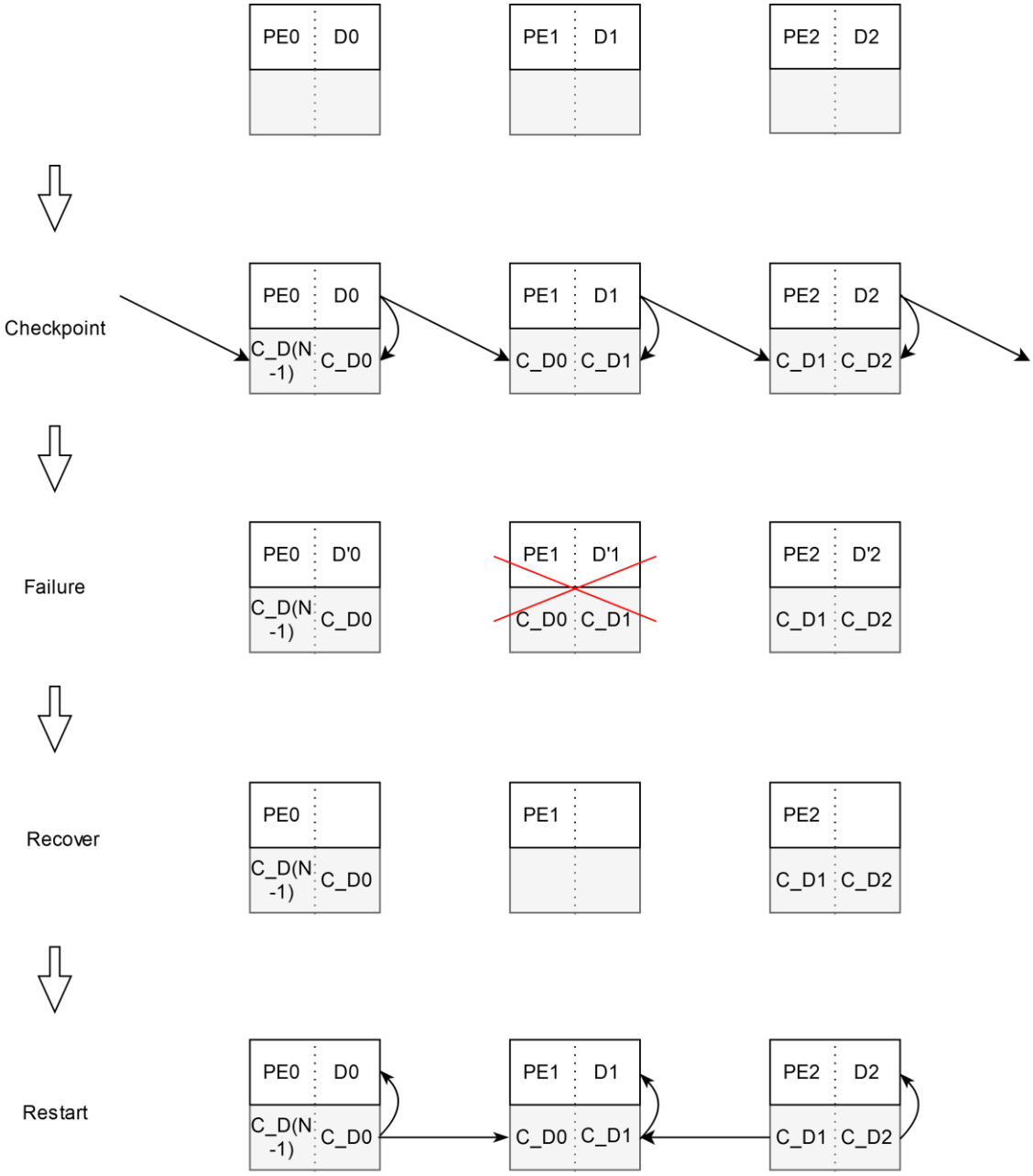

Figure 3.3: In-Memory Checkpoint-Restart

The following algorithm shows how we implement in-memory checkpointing.

Algorithm 3.8:

```
1.  checkpoint(sample)
2.  {
3.      // the below two data structures map the name of the
4.      // checkpoint file to the checkpoint data
5.      // std::unordered_map<std::string, std::string> swriter.chkpt;
6.      // std::unordered_map<std::string, std::string>  swriter.buddy_chkpt;
7.
8.      ostringstream os;
9.      write_header_data(os);
10.     write_atom_data(os, atom);
11.
12.     // This part of the algorithm is same as the multi-file approach, only
13.     // difference being data being written to ostringstream instead of the file
14.
15.     // Send and receive size of checkpoint data from the adjacent tasks
16.     int my_count = (int)os.tellp();
17.     int buddy_count;
18.     MPI_Status status_sendrecv;
```

```
19.
20.     int next_task = (MPIdata::rank() + 1) % MPIdata::size();
21.     int prev_task = (MPIdata::size() + MPIdata::rank() - 1) % MPIdata::size();
22.
23.     MPI_Sendrecv(&my_count, 1, MPI_INT, next _task, 0, &buddy_count, 1,
24.             MPI_INT, prev_task, 0, MPI_COMM_WORLD, &status_sendrecv);
25.
26.     // storing the tasks own checkpoint in-memory
27.     chkpt[filename] = os.str();
28.     os.clear();
29.
30.     buddy_chkpt[filename].resize(buddy_count);
31.
32.     char* osStringPtr = (char *)chkpt[filename].c_str();
33.     char* buddy_char = (char *)buddy_chkpt[filename].c_str();
34.
35.     // getting the checkpoint task of the other task and storing it in-memory
36.     MPI_Sendrecv(osStringPtr, my_count, MPI_CHAR, next_task, 0,
37.             buddy_char, buddy_count, MPI_CHAR,
38.             prev_task, 0, MPI_COMM_WORLD, &status_sendrecv);
39. }
```

The initial stage of Algorithm 3.8 is the same as multi-file checkpoint algorithm. All the data that is stored in the file is instead stored in the output string stream (ostringstream) (Line 8). Then we send and receive the size of the checkpoint data adjacent task using MPI_Sendrecv (Line 23). This is later used to specify the data size to receive from the adjacent task (Line 36). The task's own data is stored in the memory (Line 27) and the output string stream is cleared (Line 28). After that, the checkpoint data is sent and received from the adjacent tasks (line 36).

Recovery takes place as shown in the following algorithm:

Algorithm 3.9:

```
1.  restart(sample, my_failure)
2.  {
3.      // my_failure is a boolean value that specifies
4.      // the task is a failed task or a health task before recovery
5.
6.      // Below two MPI_Sendrecv is used to send and receive the health status i.e.
7.      // healthy or  failed to both the adjacent nodes
8.      bool buddy_failure_l;
9.      bool buddy_failure_r;
10.     MPI_Status status_sendrecv;
11.     int next_task = (MPIdata::rank() + 1) % MPIdata::size();
```

```
12.     int prev_task = (MPIdata::size() + MPIdata::rank() - 1) % MPIdata::size();
13.
14.     MPI_Sendrecv(&my_failure, 1, MPI_C_BOOL,next_task, 0,&buddy_failure_l, 1,
15.                  MPI_C_BOOL, prev_task, 0, MPI_COMM_WORLD,
16.                 &status_sendrecv);
17.
18.     MPI_Sendrecv(&my_failure, 1, MPI_C_BOOL, prev_task, 0, &buddy_failure_r,
19.                  1, MPI_C_BOOL, next_task, 0, MPI_COMM_WORLD,
20.                 &status_sendrecv);
21.
22.     // if both the task and the task holding the failed tasks checkpoint has failed
23.     // we cannot recover hence abort the recovery.
24.      if(my_failure && buddy_failure_r)
25.     {
26.             MPI_Abort(MPIdata::comm(),1);
27.      } else if(my_failure)
28.     {
29.             // if the task has failure the task receives its checkpointed data from
30.             // the adjacent task and the other adjacent tasks checkpointed data
31.
32.             int my_count;
33.             MPI_Recv(&my_count, 1, MPI_INT, next_task, 0,
```

```
34.                         MPI_COMM_WORLD, &status_sendrecv);
35.
36.             swriter.chkpt[uri].resize(my_count);
37.             char* my_char = (char* )swriter.chkpt[uri].c_str();
38.             MPI_Recv(my_char, my_count, MPI_CHAR, next_task, 0,
39.                         MPI_COMM_WORLD, &status_sendrecv);
40.
41.             MPI_Recv(&my_count, 1, MPI_INT, prev_task 0,
42.                         MPI_COMM_WORLD, &status_sendrecv);
43.
44.             swriter.buddy_chkpt[uri].resize(my_count);
45.             my_char = (char* )swriter.buddy_chkpt[uri].c_str();
46.             MPI_Recv(my_char, my_count, MPI_CHAR, prev_task, 0,
47.                         MPI_COMM_WORLD, &status_sendrecv);
48.         }
49.       else if(buddy_failure_l)
50.       {
51.             // if the adjacent task has failed send its checkpointed data
52.             //for the task to recover
53.             int buddy_count = swriter.buddy_chkpt[uri].size();
54.             MPI_Send(&buddy_count, 1, MPI_INT, prev_task, 0,
55.                         MPI_COMM_WORLD);
```

```
56.
57.         const char* osStringPtr = swriter.buddy_chkpt[uri].c_str();
58.         MPI_Send(osStringPtr, buddy_count, MPI_CHAR, prev_task, 0,
59.                   MPI_COMM_WORLD);
60.         } else if (buddy_failure_r){
61.                   // The task sends its own checkpointed data if the
62.                   // failed task has lost its data
63.                   int buddy_count = swriter.chkpt[uri].size();
64.                   MPI_Send(&buddy_count, 1, MPI_INT, next_task, 0,
65.                             MPI_COMM_WORLD);
66.
67.         const char* osStringPtr = swriter.chkpt[uri].c_str();
68.
69.         MPI_Send(osStringPtr, buddy_count, MPI_CHAR, next_task,
70.                   0,MPI_COMM_WORLD);
71.    }
72.    MPI_Barrier(MPI_COMM_WORLD);
73.  // After this point the same process is followed like the multi-file approach
74.  //instead of reading form file, data is read from the memory
75.  }
```

Thus, we see in Algorithm 3.9, 4 cases need to be handled while recovering. If both the tasks (Line 24) i.e., failed tasks and the task holding the failed tasks checkpointed data fails, we abort as it is not recoverable. If not, the failed task receives data from both the adjacent tasks (Line 41), and the adjacent sends the required data to the failed task using MPI_Sendrecv (Line 58 and 69). After this point, the execution continues similarly like the multi-file approach. Instead of reading from the file, data is read from the memory and the tasks are recovered. Thus, using the above approach we implemented in-memory checkpoint-restart for fault tolerant RT-TDDFT.

Thus, we saw how two new checkpoint-restart techniques of multi-file and in-memory checkpoint-restart were implemented in RT-TDDFT implementation that runs high performance computing systems.

4. Results

4.1. Preliminary Analysis

We performed a preliminary performance analysis on single file checkpoint-restart implementation in RT-TDDFT implementation. We analyzed serial writes where MPI collection I/O writes to Lustre File System with stripe count 1, 8, and 24. These tests ran for 100 iterations with checkpointing every 10 iterations.

As seen in Figure 4.1, time spent on writing keeps reducing as we use more parallelization in writing. Increasing the stripe count partitions the data into multiple OSTs thereby increasing the writing parallelism. This achieves higher writing speeds.

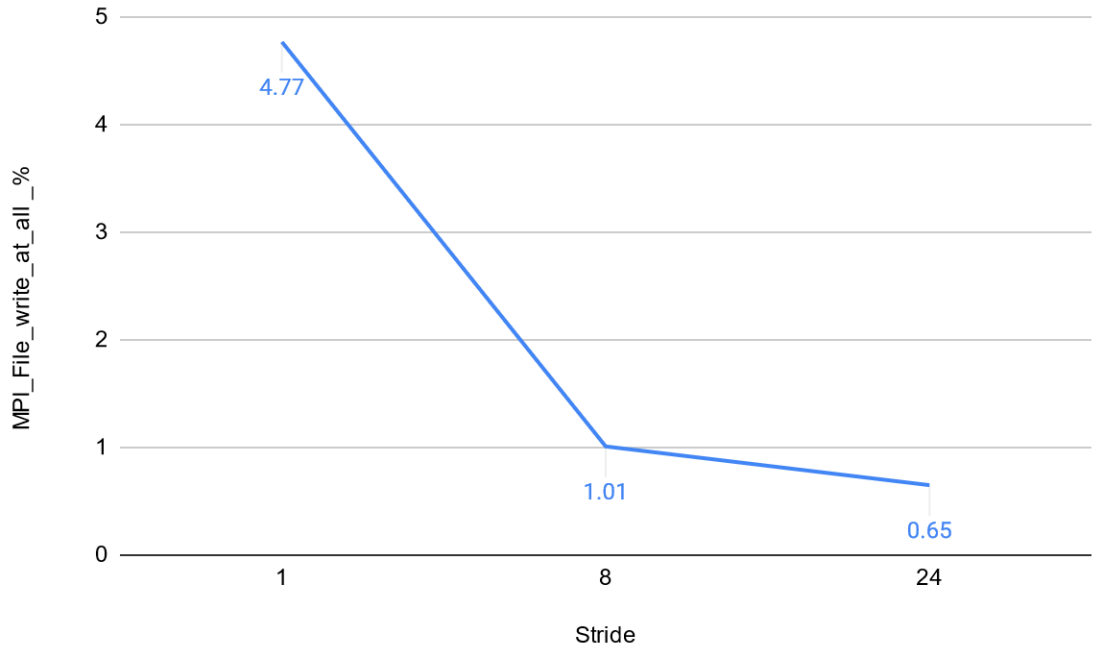

Figure 4.1: Percentage of time spent in writes vs storage methods

Figure 4.2 shows the increase in writing speed with an increase in stripe count of Luster File System. We tried increasing the number of nodes to achieve higher writing speeds. Figure 4.3 shows the writing speed changes with respect to an increase in the number of nodes. We see that writing speed does not improve with the increase in the number of nodes. This shows the bottleneck in the writing process.

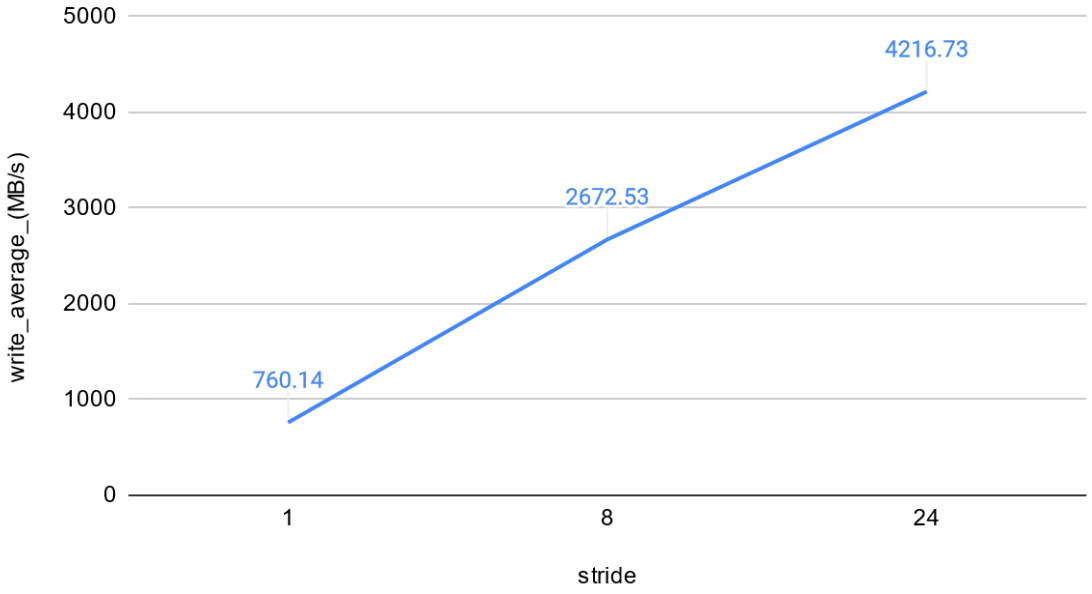

Figure 4.2: Writing speeds vs Lustre FS stipe count

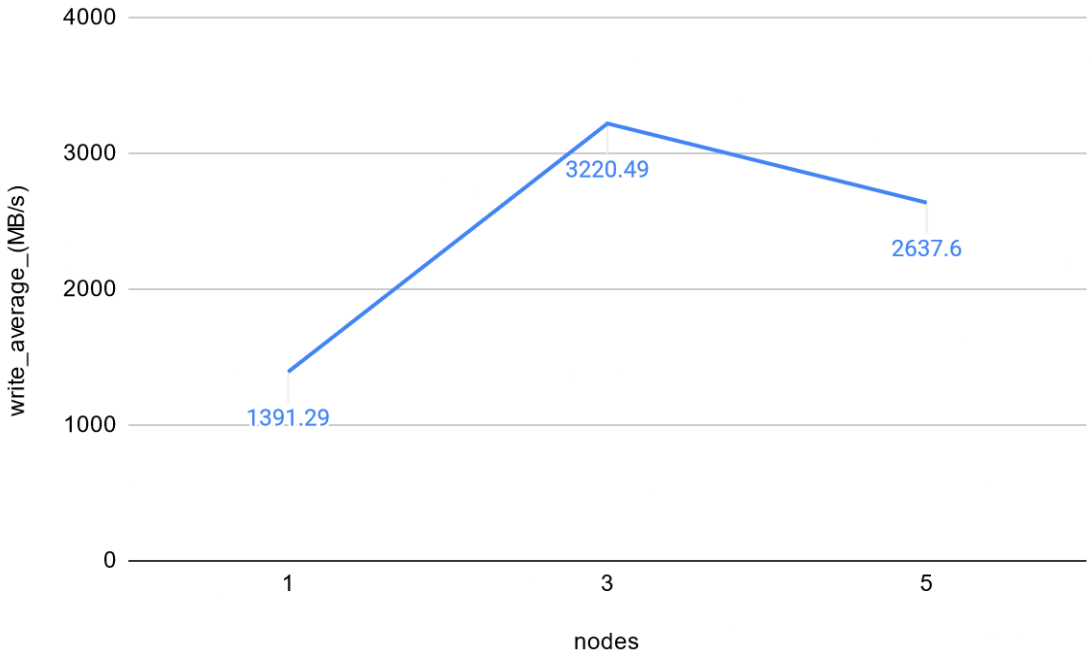

Figure 4.3: Writing speed vs number of nodes

To further increase the sample size, we ran a large sample having the checkpointed file size of around 120GB, checkpointing every iteration for 10 iterations and the profiling results showed that 14.01% time was spent on writing to the disk using MPI collective I/O as shown in Figure 4.4. Thus, we see a significant amount of time being spent in checkpointing is due to slow writing speed. An increase in data size could further increase the overhead hence there is a need to improve the performance of checkpointing.

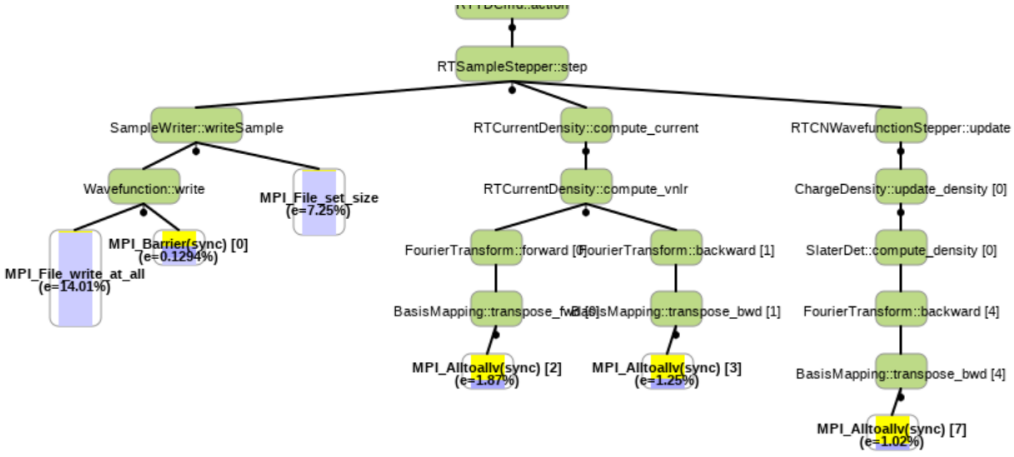

Figure 4.4: Profiling result for large samples

Thus, we see that writing speed is consuming a significant portion of the computation time for checkpointing in RT-TDDFT and is a bottleneck that needs to be improved to decrease the overhead of checkpointing.

4.2. Comparative Analysis

We performed a comparative analysis of our two newly implemented methods, multi-file checkpoint-restart and in-memory checkpoint-restart in RT-TDDFT with single-file

checkpoint restart. By trying different checkpointing frequencies in the iterations and comparing the time taken for checkpointing, program execution time, and recovery time. We performed the tests with 100 iterations on 5 nodes having 640 MPI tasks with around 4GB of checkpointing data. For single file checkpoint-restart, the files were written to the Lustre file system with stripe count 8 and for multi-file checkpoint-restart, the files were written with stripe count 1.

For time spent in checkpointing, as shown in Figure 4.5 we see that there is significant performance gain as the frequency of checkpointing increases. We see that there is an 85.93% decrease in checkpointing time from single file checkpoint-restart to multi-file checkpoint-restart and a 26.9% decrease in checkpointing time from multi-file checkpoint-restart to in-memory checkpoint-restart for checkpointing every iteration.

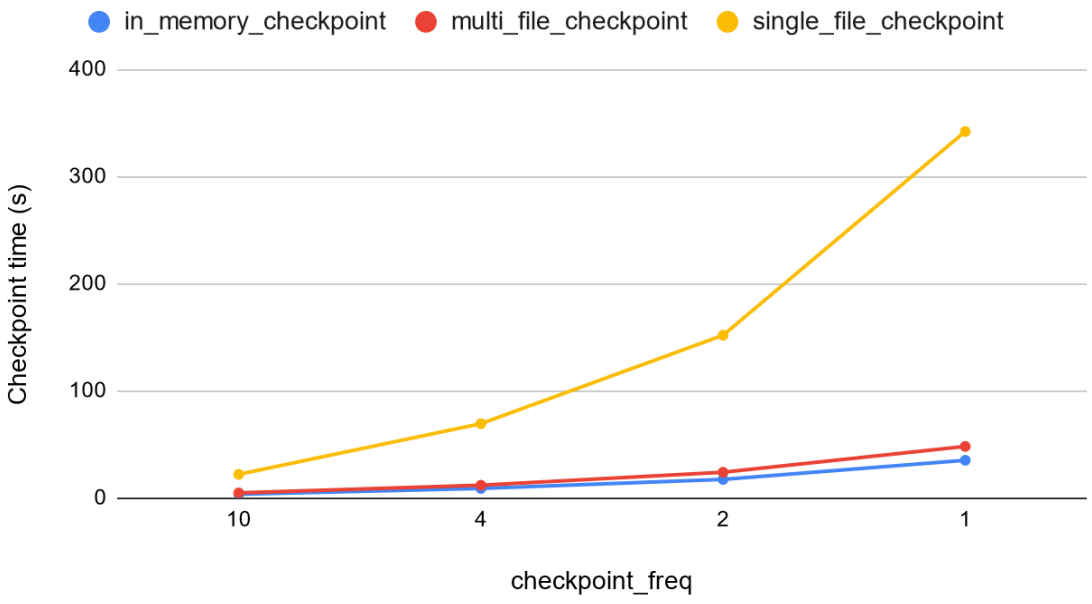

Figure 4.5: Checkpointing time vs checkpointing frequency

This shows that writing a single file to a disk is much slower than writing multiple files and writing to the main memory is much faster than writing to the disk.

We see similar results for the total program execution time as well as shown in Figure 4.6. There was a 38.55% decrease in program execution time from single file checkpoint-restart to multi-file checkpoint-restart and a 2.45% decrease from multi-file checkpoint-restart to in-memory checkpoint-restart for checkpointing every iteration. As the checkpointing time decreased the program execution time improved as well. We see an overall better result as the overhead of writing to the disk decreases.

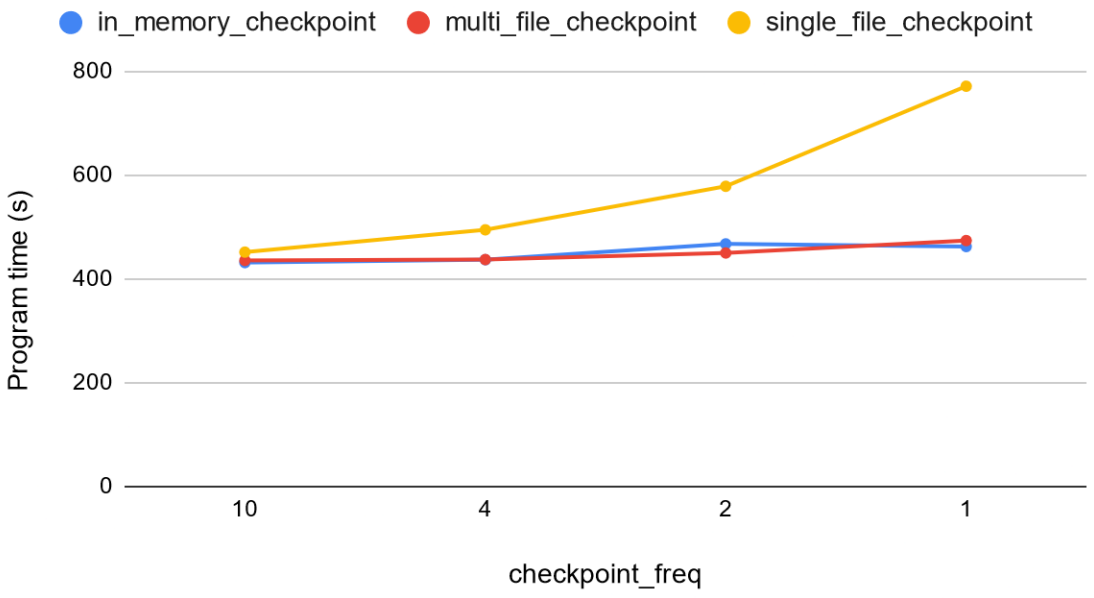

Figure 4.6: Program time vs checkpointing frequency

From these two results, we found that the percentage of time spent on checkpointing in the total execution time reduced from 44.33% to 10.14% from single file checkpoint-restart to multi-file checkpoint-restart and from 10.14% to 7.6% from multi-file checkpoint-restart to in-memory checkpoint-restart for checkpointing every iteration as shown in Figure 4.7. Thus, we see that the performance of checkpointing improved significantly as more parallelism is achieved due to having multiple files, and writing in-memory is much faster than writing to the disk.

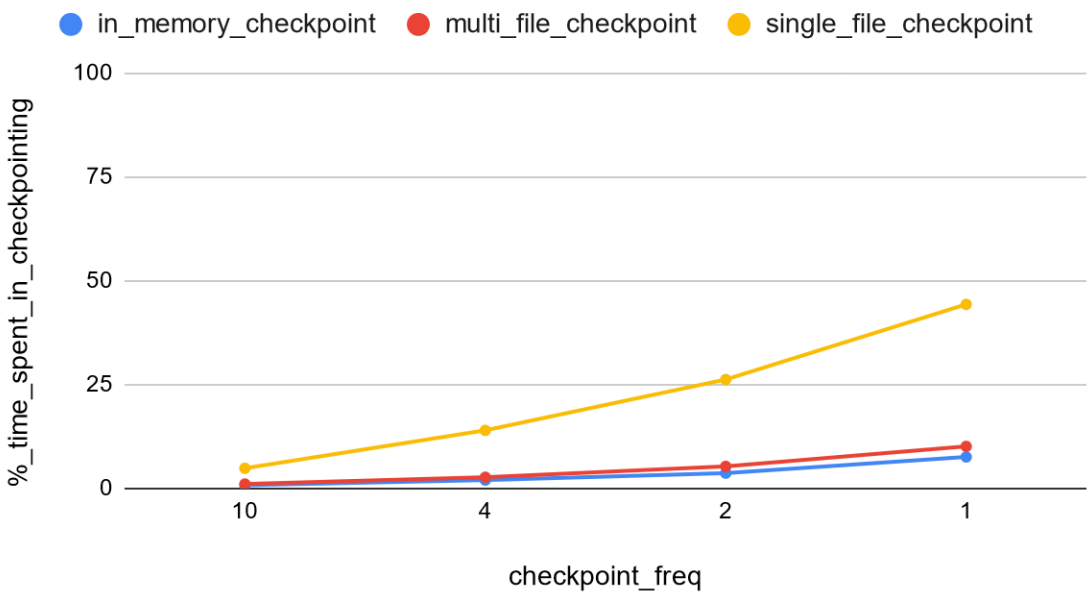

Figure 4.7: Percentage time spent in checkpointing

Having faster recovery time is important where failures are a common occurrence as too much time being spent on recovery would increase the program execution time. We also compared the recovery time taken for the three approaches.

As shown in Figure 4.8, we see a 67.56% decrease in recovery time from single file checkpoint to multi-file checkpoint-restart and a 6.53% decrease in recovery time from multi-file checkpoint-restart to in-memory checkpoint. This shows that time spent in recovery also decreases when separate files are used and when reading data in-memory.

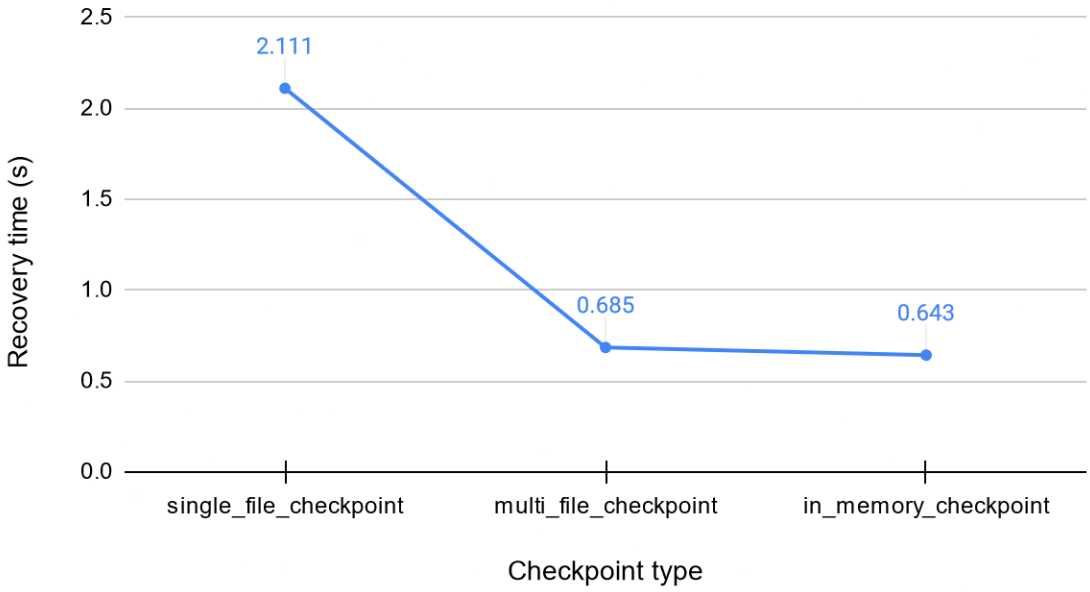

Figure 4.8: Recovery time vs checkpoint type

We ran a test for a checkpointing sample having 254GB of checkpointing data compared to the previous test of 4GB data. We found similar results for higher data sizes as well indicating good performance irrespective of the size of the checkpointing data. In Figure 4.9, we see that single file checkpoint-restart took 43.20% of execution time but multi-file checkpoint-restart and in-memory checkpoint-restart took just 12.69% and 10.53% of execution time respectively when checkpointing every iteration.

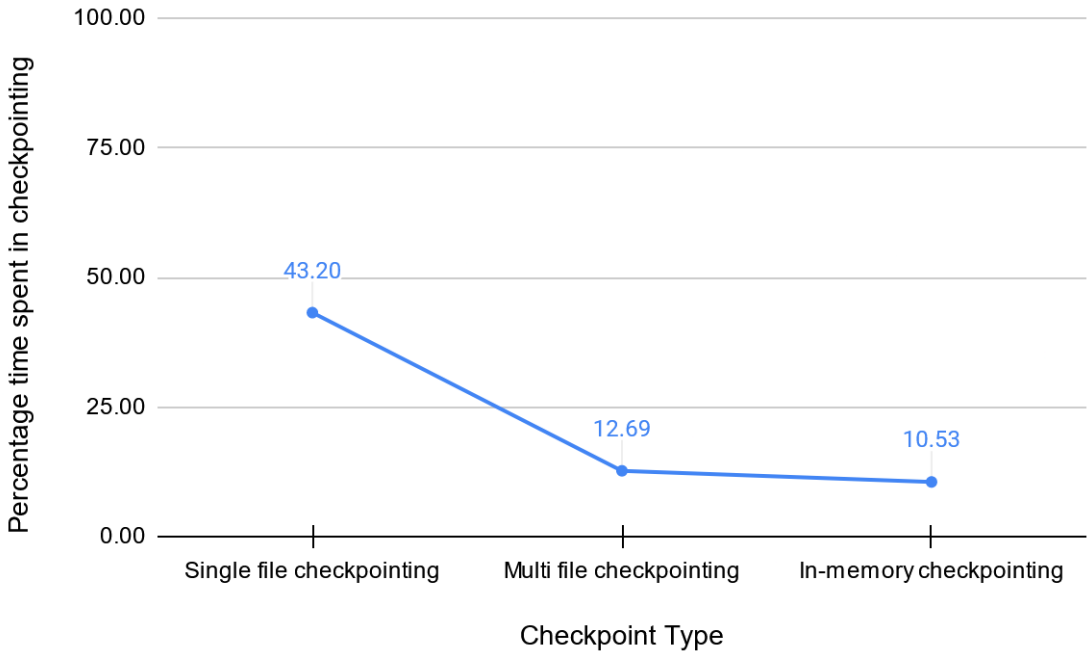

Figure 4.9: Percentage time spent in checkpointing for higher data size

We analyzed data transfer speeds for multi-file checkpoint-restart and in-memory checkpoint-restart. We ran tests on 5 nodes, 640 MPI tasks for saving a checkpoint file of around 130 GB. Each task had a file size of around 217 MB. The aggregate writing speed of all tasks includes computation time and serialization time was 20.08 GB/s and 20.25 GB/s for multi-file checkpoint-restart and in-memory checkpoint-restart respectively and individual task writing speed for the same was 33.61 MB/s and 33.9 MB/s.

For multi-file checkpoint-restart, the file stream writing without compute and serialization had an aggregate speed of 230.11 GB/s and an individual writing speed per task was 385.20 MB/s.

For in-memory checkpoint-restart, the MPI_Sendrcv bandwidth without compute and serialization had an aggregate speed of 260.38 GB/s and the individual writing speed per task was 435.88 MB/s.

	Multi-File Checkpoint-Restart	In-memory Checkpoint-Restart
Number of nodes, MPI tasks	5 nodes, 640 MPI tasks	
Write Time	6.47 s	6.42 s
Aggregate file size	130 GB	130 GB
File size per MPI task	217 MB	217 MB
Aggregate write speed (including computation + serialization time)	20.08 GB/s	20.25 GB/s
Write speed (including computation + serialization time)	33.61 MB/s	33.9 MB/s
Aggregate write speed	230.11 GB/s (ofstream)	260.38 GB/s (MPI_Sendrcv)
Write speed per MPI task	385.20 MB/s (ofstream)	435.88 MB/s (MPI_Sendrcv)
Bandwidth Limit	5 TB/s (Permultter Scratch Lustre FS, aggregate, 298 OST, 4864 perlmutter nodes) 25 GB/s (NIC, 1 per node) 204.8 GB/s (memory bandwidth per CPU, 2 CPUs/node, 64 cores/CPU)	

Table 4.1: Data transfer speed

5. Discussion

In this thesis, we implement checkpoint-restart in RT-TDDFT using multi-file and in-memory checkpoint-restart techniques. We performed a comparative analysis of multi-file and in-memory checkpoint-restart with single file checkpoint-restart. Compared to other techniques for performance improvement, multi-file checkpoint-restart, and in-memory checkpoint-restart provide multiple benefits.

Incremental checkpointing requires storing multiple increments of changes. This could increase the cost of storage. The recovery also becomes complex as multiple files involving changes are kept and need to be applied on recovery. Both multi-file and in-memory checkpoint-restart overwrite the original file so it does not involve finding the previous changes. This improves the ease of checkpointing and recovery. Forked-checkpointing creates a fork of the process to checkpoint the data which requires an additional overhead of creating a separate copy of the process. It is possible that both processes could be competing for resources. Multi-file and in-memory checkpoint-restart is much simpler in that it does not require creating unwanted copies of the resources and in-memory checkpoint-restart utilizes memory only for the data needed for recovery. Using compilers for instrumenting checkpointing code might not always be effective as it is not always possible to decide where to checkpoint and at what interval. Compared to that, our method gives us complete flexibility in deciding at what point and at what frequency to checkpoint. Compilers may not be able to fully utilize the parallelization of

multiple processes. In-memory checkpoint-restart utilizes parallelization by storing checkpoints in adjacent process memory and all the processes can perform this in parallel and the same goes for multi-file checkpoint-restart. All the processes write their checkpointing file in parallel utilizing complete parallelization. System level checkpointing is not necessarily portable as they are performed on the kernel level or hardware level. Our implementation is portable and does not depend upon the underlying kernel or hardware.

Our approach does have some limitations as well. In-memory checkpoint-restart requires the application to have enough free memory to store the checkpointing data of itself and its peers. However, an application having enough free memory benefits from this approach as extra disk space is not required and it helps in utilizing the free resources available. An application not having enough free memory can still fall back on multi-file checkpoint-restart and can benefit from the performance gains compared to the single file checkpoint-restart. There is also a possibility that the task holding the checkpointed data for a failed task also fails and the failed task is not able to recover. Such issues can be fixed by either storing the checkpoints to the disk with less frequency or by storing the checkpointed data with few a more tasks as well to have more redundancies. The multi-file checkpoint-restart approach could generate too many small files if the number of MPI tasks is high, and the data held by each task is less. Approaches like grouping tasks and generating single file for the group i.e., using a hybrid approach of single and multi-file checkpoint-restart can lower the cost of generating too many small files.

Thus, we see that both approaches give a lot of flexibility in the way of implementation compared to the other approaches used for checkpointing. Other approaches can be paired with it and can be further improved to reduce the cost of checkpointing.

6. Conclusion

Checkpointing an application provides fault tolerance in high performance computing systems. However, checkpointing comes with a dominant cost of writing on a stable storage. In this thesis, we analyzed the overhead of single-file checkpoint-restart present in Real-Time Time-Dependent Density Functional Theory implementation. We then decreased the overhead by creating multiple files for checkpointing. We further improved the checkpointing process by storing the data in-memory instead of the disk. These two techniques for checkpoint-restart showed significant checkpointing performance gains in RT-TDDFT computation in HPC systems. Similar results were seen in restoration times on failure as well. Restoring from multiple files or from memory performed better than restoring from a single file. Thus, we implement fault tolerant RT-TDDFT by in-memory and multi-file checkpoint-restart in high performance computing systems.

Acknowledgments

This work was supported by the U.S. Department of Energy,

Office of Science, Office of Advanced Scientific Computing Research,

Scientific Discovery through the Advanced

Computing (SciDAC) program under Award Number DESC0022209.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility.

Bibliography

[1] Egwutuoha, I.P., Levy, D., Selic, B. *et al.* A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J Supercomput* **65**, 1302–1326 (2013). https://doi.org/10.1007/s11227-013-0884-0

[2] Kalaiselvi, S., Rajaraman, V. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana* **25**, 489–510 (2000). https://doi.org/10.1007/BF02703630

[3] Walters, J.P., Chaudhary, V. (2006). Application-Level Checkpointing Techniques for Parallel Programs. In: Madria, S.K., Claypool, K.T., Kannan, R., Uppuluri, P., Gore, M.M. (eds) Distributed Computing and Internet Technology. ICDCIT 2006. Lecture Notes in Computer Science, vol 4317. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11951957_21

[4] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2003. Automated application-level checkpointing of MPI programs. SIGPLAN Not. 38, 10 (October 2003), 84–94. https://doi.org/10.1145/966049.781513

[5] F. Gygi, “Qbox: a large-scale parallel implementation of First-Principles Molecular Dynamics” (http://eslab.ucdavis.edu).

[6] F. Gygi, "Architecture of Qbox: A scalable first-principles molecular dynamics code," in IBM Journal of Research and Development, vol. 52, no. 1.2, pp. 137-144, Jan. 2008, doi: 10.1147/rd.521.0137.

[7] Shalf, J., Dosanjh, S., Morrison, J. (2011). Exascale Computing Technology Challenges. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds) High Performance Computing for Computational Science – VECPAR 2010. VECPAR 2010. Lecture Notes in Computer Science, vol 6449. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-19328-6_1

[8] J. S. Plank and J. Xu and R. H. B. Netzer, Compressed Differences: An Algorithm for Fast Incremental Checkpointing, University of Tennessee, CS-95-302, August, 1995, http://web.eecs.utk.edu/~jplank/plank/papers/CS-95-302.html, https://web.eecs.utk.edu/~jplank/plank/papers/CS-95-302.pdf

[9] Libckpt: Transparent Checkpointing under UNIX https://courses.cs.vt.edu/~cs5204/fall07-kafura/Papers/FaultTolerance/LibCkpt.pdf

[10] Cappello, Franck. (2009). Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. IJHPCA. 23. 212-226. 10.1177/1094342009106189.

[11] Gygi, Francois. (2006). Large-scale first-principles molecular dynamics: Moving from terascale to petascale computing. Journal of Physics: Conference Series. 46. 10.1088/1742-6596/46/1/038.

[12] Gygi, Francois. (2008). Architecture of Qbox: A scalable first-principles molecular dynamics code. IBM Journal of Research and Development. 52. 137 - 144. 10.1147/rd.521.0137.
[13] Milojicic DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration. ACM Comput Surv 32(3):241–299
[14] Clark C, Fraser K, Hand S et al (2005) Live migration of virtual machines. In: Proceedings of the 2nd conference on symposium on networked systems design and implementation, vol 2, May 2005, pp273–286
[15] Chen L, Avizienis A (1978) N-version programming: a fault-tolerance approach to reliability of soft-ware operation, June, Toulouse, France, pp 3–9
[16] Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods Z ChenACM SIGPLAN Notices 48 (8), 167-176 2013
[17] Johnson C, Holloway C (2007) The dangers of failure masking in fault tolerant software: aspects of a recent in-flight upset event. In: 2nd institution of engineering and technology international conference on system safety, pp 60–65
[18] Slivinski T, Broglio C, Wild C et al. (1984) Study of fault-tolerant software technology. NASA CR 172385, Langley Research, Center, VA
[19] Li K, Naughton JF, Plank JS (1994) Low-latency, concurrent checkpointing for parallel programs. IEEE Trans Parallel Distrib Syst 5(8):874–879
[20] Kalaiselvi S, Rajaraman V (2000) A survey of checkpointing algorithms for parallel and distributed computers, pp 489–510
[21] Wang Y-M, Chung P-Y, Lin I-J, Fuchs WK (1995) Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. IEEE Trans Parallel Distrib Syst 6(5):546–554
[22] Chen, Zizhong ; Dongarra, Jack. Algorithm-based fault tolerance for fail-stop failures. In: IEEE Transactions on Parallel and Distributed Systems. 2008 ; Vol. 19, No. 12. pp. 1628-1641.
[23] J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, G. E. Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, H. You, and S. S. Vadhiyar. 2006. Self-adapting numerical software (SANS) effort. IBM J. Res. Dev. 50, 2/3 (March 2006), 223–238. https://doi.org/10.1147/rd.502.0223
[24] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. 2013. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). Association for Computing Machinery, New York, NY, USA, Article 44, 1–12. https://doi.org/10.1145/2503210.2503226
[25] Tao, Dingwen & Song, Shuaiwen & Krishnamoorthy, Sriram & Wu, Panruo & Liang, Xin & Zhang, Eddy & Kerbyson, Darren. (2016). New-Sum: A Novel Online ABFT Scheme For General Iterative Methods. 43-55. 10.1145/2907294.2907306.
[26] L. Tan, S. L. Song, P. Wu, Z. Chen, R. Ge and D. J. Kerbyson, "Investigating the Interplay between Energy Efficiency and Resilience in High Performance

Computing," 2015 IEEE International Parallel and Distributed Processing Symposium, Hyderabad, India, 2015, pp. 786-796, doi: 10.1109/IPDPS.2015.108.
[27] Hakkarinen, Doug & Chen, Zizhong. (2010). Algorithmic Cholesky factorization fault recovery. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. 1 - 10. 10.1109/IPDPS.2010.5470436.
[28] Wu, Panruo & Guan, Qiang & DeBardeleben, Nathan & Blanchard, Sean & Tao, Dingwen & Liang, Xin & Chen, Jieyang. (2016). Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. 31-42. 10.1145/2907294.2907315.
[29] Z. Chen and J. Dongarra, "Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing," in IEEE Transactions on Computers, vol. 58, no. 11, pp. 1512-1524, Nov. 2009, doi: 10.1109/TC.2009.42
[30] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. 2015. Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition. IEEE Trans. Parallel Distrib. Syst. 26, 5 (May 2015), 1323–1335. https://doi.org/10.1109/TPDS.2014.2320502
[31] Chen, Z., Dongarra, J. (2005). Numerically Stable Real Number Codes Based on Random Matrices. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds) Computational Science – ICCS 2005. ICCS 2005. Lecture Notes in Computer Science, vol 3514. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11428831_15
[32] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey S. Vetter, and Sparsh Mittal. 2016. Algorithm-Directed Data Placement in Explicitly Managed Non-Volatile Memory. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16). Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2907294.2907321
[33] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2011. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In Proceedings of the second workshop on Scalable algorithms for large-scale systems (ScalA '11). Association for Computing Machinery, New York, NY, USA, 25–28. https://doi.org/10.1145/2133173.2133185
[34] Zizhong Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," 2008 IEEE International Symposium on Parallel and Distributed Processing, Miami, FL, USA, 2008, pp. 1-8, doi: 10.1109/IPDPS.2008.4536158
[35] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2018. Improving performance of iterative methods by lossy checkpointing. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18). Association for Computing Machinery, New York, NY, USA, 52–65. https://doi.org/10.1145/3208040.3208050
[36] Z. Chen, "Optimal real number codes for fault tolerant matrix operations," Proceedings of the Conference on High Performance Computing Networking,

Storage and Analysis, Portland, OR, USA, 2009, pp. 1-10, doi: 10.1145/1654059.1654089
[37] D. Hakkarinen and Z. Chen. Multilevel diskless checkpointing. IEEE Transactions on Computers, 62(4):772783, 201
[38] Liang, Xin & Chen, Jieyang & Tao, Dingwen & Li, Sihuan & Wu, Panruo & Li, Hongbo & Ouyang, Kaiming & Liu, Yuanlai & Song, Fengguang. (2017). Correcting soft errors online in fast fourier transform. 1-12. 10.1145/3126908.3126915
[39] J. Chen, X. Liang and Z. Chen, "Online Algorithm-Based Fault Tolerance for Cholesky Decomposition on Heterogeneous Systems with GPUs," 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, 2016, pp. 993-1002, doi: 10.1109/IPDPS.2016.81
[40] Chen, Z., & Dongarra, J. (2008). A scalable checkpoint encoding algorithm for diskless checkpointing. In Proceedings of IEEE International Symposium on High Assurance Systems Engineering|Proc. IEEE Int. Symp. High Assur. Syst. Eng. (pp. 71-79). IEEE Computer Society . https://doi.org/10.1109/HASE.2008.13
[41] Gygi, Francois. (2006). Large-scale first-principles molecular dynamics: Moving from terascale to petascale computing. Journal of Physics: Conference Series. 46. 10.1088/1742-6596/46/1/038
[42] Francois Gygi, Erik W. Draeger, Martin Schulz, Bronis R. de Supinski, John A. Gunnels, Vernon Austel, James C. Sexton, Franz Franchetti, Stefan Kral, Christoph W. Ueberhuber, and Juergen Lorenz. 2006. Large-scale electronic structure calculations of high-Z metals on the BlueGene/L platform. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC '06). Association for Computing Machinery, New York, NY, USA, 45–es. https://doi.org/10.1145/1188455.1188502
[43] F. Gygi et al., "Large-Scale First-Principles Molecular Dynamics simulations on the BlueGene/L Platform using the Qbox code," SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, WA, USA, 2005, pp. 24-24, doi: 10.1109/SC.2005.40
[44] B. Randell, "System structure for software fault tolerance," in *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220-232, June 1975, doi: 10.1109/TSE.1975.6312842
[45] Elnozahy, Elmootazbellah & Alvisi, Lorenzo & Wang, Yi-min & Johnson, David. (2002). A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys. 34. 10.1145/568522.568525.
[46] Walters, J.P., Chaudhary, V. (2006). Application-Level Checkpointing Techniques for Parallel Programs. In: Madria, S.K., Claypool, K.T., Kannan, R., Uppuluri, P., Gore, M.M. (eds) Distributed Computing and Internet Technology. ICDCIT 2006. Lecture Notes in Computer Science, vol 4317. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11951957_21
[47] J. Duell. (2005, Apr.) The design and implementation of berkely lab's linux checkpoint/restart. [Online]. Available: https://escholarship.org/uc/item/40v987j0

[48] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models," *International Conference on Dependable Systems and Networks (DSN'06)*, Philadelphia, PA, USA, 2006, pp. 425-434, doi: 10.1109/DSN.2006.18.

[49] Chakravorty, S., Mendes, C.L., Kalé, L.V. (2006). Proactive Fault Tolerance in MPI Applications Via Task Migration. In: Robert, Y., Parashar, M., Badrinath, R., Prasanna, V.K. (eds) High Performance Computing - HiPC 2006. HiPC 2006. Lecture Notes in Computer Science, vol 4297. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11945918_47

[50] Nagarajan, Arun Babu, Mueller, Frank, Engelmann, Christian, & Scott, Stephen L. Proactive Fault Tolerance for HPC with Xen Virtualization. United States.

[51] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. 2008. Efficient state transfer for hypervisor-based proactive recovery. In Proceedings of the 2nd workshop on Recent advances on intrusiton-tolerant systems (WRAITS '08). Association for Computing Machinery, New York, NY, USA, Article 4, 1–6. https://doi.org/10.1145/1413901.1413905

[52] S. Pertet and P. Narasimhan, "Proactive recovery in distributed CORBA applications," International Conference on Dependable Systems and Networks, 2004, Florence, Italy, 2004, pp. 357-366, doi: 10.1109/DSN.2004.1311905.

[53] J. S. Plank, Kai Li and M. A. Puening, "Diskless checkpointing," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972-986, Oct. 1998, doi: 10.1109/71.730527

[54] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[55] Gao, Qi & Yu, Weikuan & Huang, Wei & Panda, D.K.. (2006). Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand.. Proceedings of the International Conference on Parallel Processing. 471-478. 10.1109/ICPP.2006.26.

[56] Sankaran S, Squyres JM, Barrett B, et al. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. The International Journal of High Performance Computing Applications. 2005;19(4):479-493. doi:10.1177/1094342005056139

[57] Heo, Junyoung & Yi, Sangho & Cho, Yookun & Hong, Jiman. (2006). Space-Efficient Page-Level Incremental Checkpointing.. J. Inf. Sci. Eng.. 22. 237-246.

[58] Bronevetsky, G., Marques, D., Pingali, K. and Stodghill, P. 2003.C3: a system for automating application-level checkpoint-ing of MPI programs. In Proceedings of the 16th Interna-tional Workshop on Languages and Compilers for ParallelComputing (LCPC 2003), October 2003.

[59] Plank, James S. ; Chen, Yuqun ; Li, Kai et al. / Memory Exclusion : Optimizing the Performance of Checkpointing Systems. In: Software - Practice and Experience. 1999 ; Vol. 29, No. 2. pp. 125-142.

[60] Plank, James S. ; Li, Kai ; Puening, Michael A. / Diskless checkpointing. In: IEEE Transactions on Parallel and Distributed Systems. 1998 ; Vol. 9, No. 10. pp. 972-986.
[61] J. Plank, J. Xu, and R. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995