2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

An Evaluation of Different I/O Techniques for Checkpoint/Restart

Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager and Gerhard Wellein
Erlangen Regional Computing Center, University of Erlangen-Nuremberg
Erlangen, Germany
{*faisal.shahzad, markus.wittmann, thomas.zeiser, georg.hager, gerhard.wellein*}*@rrze.fau.de*

***Abstract*—Today's High Performance Computing (HPC) clusters consist of hundreds of thousands of CPUs, memory units, complex networks, and other components. Such an extreme level of hardware parallelism reduces the mean time to failure (MTTF) of the overall cluster. The future of HPC urgently demands to develop environments that facilitate programs to run successfully even in the presence of failures. Checkpoint/Restart (C/R) is one of the most common techniques to provide fault tolerance. C/R is relatively easy to implement, but typically it introduces significant overhead in the runtime of the application. In this paper, a checkpointing technique is presented that significantly reduces the checkpoint overhead and is highly scalable. This is achieved by overlapping the I/O for writing the checkpoint with the computation of the application. For this asynchronous checkpointing technique, a theoretical model is developed to estimate the checkpoint overhead. An implementation of this technique is then benchmarked and compared with other checkpointing strategies. We show our approach to have marginal overhead as opposite to standard synchronous checkpointing for typical application scenarios. A comparison with the node-level checkpointing technique by using Scalable Checkpoint/Restart (SCR) library is also presented.**

***Keywords*-fault tolerance; asynchronous checkpointing; multi-stage checkpointing; checkpoint/restart; MPI**

I. Introduction

In the High Performance Computing (HPC) sector, an ever increasing demand in computational capacity has recently resulted in systems with as many as 1.5 million processors, reaching a maximum performance of around 16.3 PetaFlops according to the Top500 list of Nov. 2012 [1]. The HPC community expects that with the help of such extreme levels of hardware parallelism, the ExaFlop barrier is going to be crossed in 2018 [2]. Unfortunately, the reliability of each of the hardware component is not increasing correspondingly and thus the overall system reliability is reduced. Mean time to failure (MTTF) is a metric used to estimate the reliability of a system. Today, for large HPC systems, MTTF is usually in the order of days or weeks. The MTTF of future systems is expected to decrease further. Moreover, HPC applications have continued to grow in complexity and runtime. Thus, the risk of encountering component failure during a single application run becomes high. This results in an urgent need to develop fault-tolerant solutions for HPC applications.

Several techniques exist for providing fault tolerance in applications. Each fault tolerance technique introduces a certain amount of overhead to the application in terms of time and/or resources. Checkpoint/Restart (C/R) is one of the most popular fault tolerance techniques. The overhead for C/R comes in terms of time for writing and (in the case of a restart) reading the checkpoints, and is directly proportional to the size of the checkpoint. In most of the cases, the checkpoints are synchronously stored on the parallel file system (PFS). As the bandwidth of any PFS saturates quite easily by a low number of nodes, the scalability of such techniques is very limited. In this paper, we implement an asynchronous checkpointing technique that is highly scalable and reduces the PFS-level checkpoint overhead to a minimum. We compare this approach with other C/R techniques. The comparison between these approaches is made on the bases of their respective overheads induced on the application runtime. In order to demonstrate the worst case scenario, we use a benchmark application that creates large checkpoints.

The main contributions of this paper are:

1) Introduction of an asynchronous checkpointing technique based on dedicated threads.
2) A theoretical model for overhead estimation for asynchronous and synchronous checkpointing techniques.
3) A benchmark study of overhead comparison between our presented asynchronous checkpointing approach and other C/R approaches.

This paper is structured as follows. In Sect. II, we present a brief introduction to C/R based fault tolerance techniques and the related work on C/R based optimizations. The details of our C/R approach are presented in Sect. III. In Sect. IV, we develop a model to estimate the checkpoint overhead. Section V gives an overview of our experimental framework. The performance results of our C/R implementation in comparison with other implementations are presented in Sect. VI. Finally, Sect. VII gives the summary and concludes the paper.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.145

II. Background and Related Work

C/R is a classical and the most widely used fault tolerance technique. The state (also called *snapshot*) of each process is periodically stored on a stable storage. In case of a failure, processes can be restarted from these states. The *checkpoint latency* (also regarded *checkpoint overhead per checkpoint*) is the duration required to create and store a single checkpoint on the stable storage. Depending on the size, the checkpoint latency can become significantly high resulting in large overheads. The time between two consecutive checkpoints is called *checkpoint interval*. The optimal checkpoint interval is very important so that the runtime is not spent on generating useless checkpoints. One model for determining an optimal checkpoint interval is presented in [3].

Depending upon the degree of transparency and implementation level, there exist three different kinds of C/R [4]. An *application-level* C/R service is implemented inside the application by the user. In this implementation, the user stores only the information in the checkpoints that is required for restarting. This leads to the smallest possible checkpoint size. A *user-level* C/R service is linked to a program via libraries. Such implementations usually produce larger checkpoints as they cannot take advantage of application-specific information. A *system-level* approach captures the state of a process without requiring any changes to the application. This requires support from the operating system and eventually from the used software stack (e.g. the MPI implementation or the batch system). Open MPI [5] supports two kinds of C/R services [6]. The system-level implementation is based on the Berkeley Lab Checkpoint/Restart (BLCR) library [7]. The application-level implementation is called SELF, which is based on callback routines to perform C/R.

Considerable research has been carried out on reducing the overhead caused by C/R mechanisms. Apart from checkpoint compression and write aggregation techniques [8], [9], multi-level checkpointing is seen as the key element for reducing the checkpoint overhead on the Exascale level [10]. The Scalable Checkpoint/Restart (SCR) library [11] is a multi-level checkpointing library that offers three kinds of node-level checkpoints namely local-level, partner-level, partner-XOR-level. Apart from node-level checkpoints, the user can specify the frequency at which node-level checkpoints are flushed to PFS to create a global checkpoint. With this technique, usual node-level checkpoints can be taken more frequently and they cause less overhead to the application. Expensive PFS-level checkpoints which can cope with catastrophic failures are taken less frequently. In case of a failure, restart is either performed by reading node-level or PFS-level checkpoints (if the node-level checkpoint state is inconsistent). Although the node-level approach of creating checkpoints is highly scalable, the cost of PFS-level checkpoints is rather huge. A non-blocking checkpointing system has been recently added to the library via utilizing additional dedicated "staging nodes" that transfer the node-level checkpoints to PFS in an asynchronous manner [12]. Open MPI supports such a method via the "stage checkpoint" mechanism [13], in which the checkpoint is first stored on a node-level storage and then is asynchronously transferred to the PFS to make a global checkpoint.

In the past, frameworks such as MTIO [14] and DataStager [15] have been developed to improve the I/O efficiency on the large scale systems. Almost 15 years ago, background threads have been used to overlap the computation and I/O [16]. The performance gains of such implementation was shown for collective and non-collective I/O for different architectures. We revisit such a thread-based I/O approach for making asynchronous checkpoints in the context of multi-core SMT environments with hybrid MPI/OpenMP parallelization and achieve it by utilizing a two-stage checkpointing mechanism in the application. In [17], we presented initial results for our strategy. The present paper is an extension to our previous work with the addition of a theoretical model for asynchronous checkpointing overhead and a comparison with two other library-based C/R implementations, namely SCR and Open MPI Stage checkpointing.

III. Proposed technique

In this paper, we present a technique that reduces the PFS-level checkpoint latency to a minimum level. We follow an application-level C/R approach. In most cases, application-level C/R has performance advantages as only the relevant data has to be saved instead of everything when system-level C/R is used.

A two-stage checkpointing mechanism is implemented where the first stage involves the creation of an in-memory checkpoint. In the second stage this checkpoint is transferred to the PFS in the background without interrupting the application. In the ideal case, the second stage could be achieved utilizing non-blocking MPI-IO if the I/O is carried out in an asynchronous manner. A benchmark (shown in Fig. 1) is used to evaluate the ability of an MPI implementation to perform asynchronous I/O. A compute function, which performs compute-bound calculations for a configurable amount of time, is put between the calls to *MPI_File_iwrite()* and *MPI_Wait()*. If I/O is performed asynchronously, the duration of the computation will be hidden behind the duration of I/O until the first becomes larger than the latter. The experimental framework for this

```
get_walltime_(&starttime_total);
MPI_File_iwrite(fh, buf, ndoubles_write,
                MPI_DOUBLE, &request);
perform_calc(calc_time);      //==== DUMMY CALCULATION
MPI_Wait( &request, &status );
get_walltime_(&endtime_total);
```

Figure 1. Benchmark used to measure the asynchronous non-blocking MPI-IO capabilities of an MPI implementation.

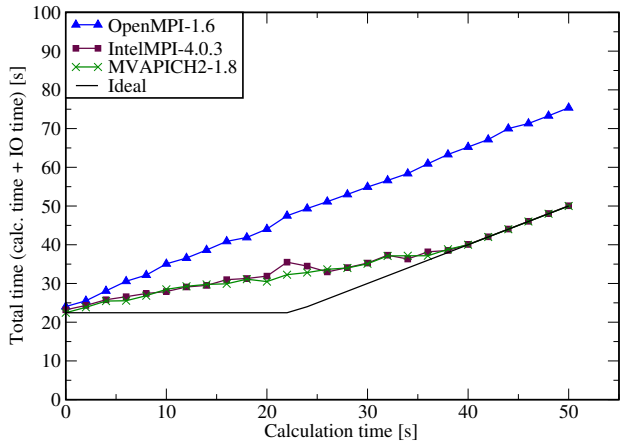


Figure 2. Benchmark analysis of the asynchronous non-blocking MPI-IO capabilities of different implementations. Ten MPI processes write a total of 20 GB of data to LiMa cluster's PFS using one node.

benchmark is described in Sec. V. Figure 2 shows that Open MPI does not perform asynchronous MPI-IO. Despite Intel MPI and MVAPICH2 hide the I/O behind the computation time, the implementations lack efficiency. Thus, using non-blocking MPI-IO routines for checkpointing purposes is not an adequate option.

In order to transfer the in-memory checkpoint to the PFS, we utilize a dedicated checkpointing thread (CP-thread) for each MPI process. The implementation relies on OpenMP, but in principle any other threading library can be used. Figure 3 shows the flow chart of the implementation. Each MPI process first creates two OpenMP threads, a worker thread and a CP-thread. Each worker thread is divided into another layer of threads via *nested* OpenMP parallelism. This gives the user the flexibility to select one dedicated CP-thread over as many worker threads as desired. The worker threads are typically distributed so that only one SMT core of each physical core is used. The CP-thread will reside on the same physical core as one of the worker threads, but bound to the remaining free SMT core. Thus, no extra physical core is used for the CP-thread which is idle most of the time. If SMT is not available, one physical core may be oversubscribed. The pinning of the processes is implemented within the application via the

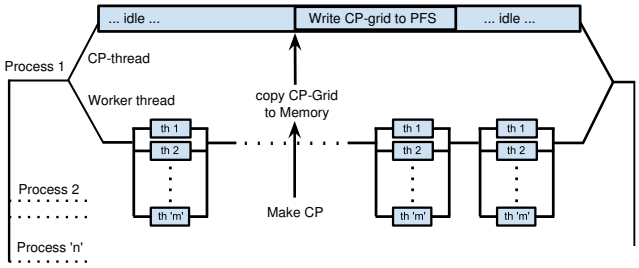


Figure 3. Program flow diagram: Each MPI process is divided into a worker thread and a CP-thread. Worker threads are further subdivided into another layer of threads. At checkpoint iteration, the worker thread signals the CP-thread to write the checkpoint.

sched_setaffinity() call. When a checkpoint is triggered (e.g. by iteration count, time, or an external signal), an in-memory checkpoint is made first by the worker thread(s). The second stage of checkpointing involves copying the in-memory checkpoint to the PFS and is carried out by the CP-thread. Figure 4 shows the pseudo code of how such an implementation could look like. During the checkpoint iteration, the worker thread signals the CP-thread that a PFS-level checkpoint can be performed. The CP-thread detects this and starts writing the data to the PFS. Each PFS-level checkpoint is created on a different set of files. As soon as the new checkpoint is complete, the last one is removed from the PFS. Thus, if a failure occurs while writing a checkpoint, a valid copy of the last checkpoint is still available for restarting.

Three schemes are followed and tested for their respective checkpointing overheads:

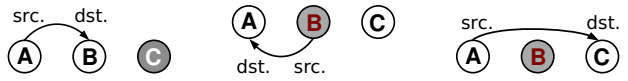
- *1 CP-thread per node:* This scheme has one MPI-process per node, with the number of worker threads equal to the number of physical cores on the node. In addition, one core of the node holds a CP-thread along with the worker thread but both threads are bound to separate virtual cores.
- *1 CP-thread per socket:* In this scheme, there is one MPI-process per socket each having the number of worker threads equal to the number of physical cores on the socket and one CP-thread in addition.
- *1 CP-thread per core:* In this scheme, each physical core has an MPI-process with an additional CP-thread attached to the second virtual core of the same physical core. If SMT is not available, physical cores may be oversubscribed.

In stencil type algorithms, "toggle grids" are frequently used, meaning that successive iterations of the algorithm write to alternating arrays. For such algorithms, the time to create the in-memory copy of a checkpoint can be avoided by introducing an extra checkpointing-grid (CP-grid). The CP-grid is in addition to the two "normal" source and

```
//========WORKER THREAD =========//
 while(current_time_step<=timesteps){
    computation_step();
    apply_BoundaryCondition();
    if(current_time_step==checkpoint_iter){
        CP_temp_swap=src_grid;
        src_grid=CP_grid;
        CP_grid=dst_grid;
        signal_write_checkpoint();
    }
    if(current_time_step==(checkpoint_iter+1)){
        src_grid=CP_temp_swap;
    }
    switch_grid_pointers(dst_grid,src_grid);
    ++current_time_step;
 }
```

```
//=========CP THREAD=========//
  while(!iteration_finished){
      wait_for_write_checkpoint_signal();
      if(signaled_write_checkpoint()){
          write_checkpoint_to_PFS();
      }
  }
```

Figure 4. Pseudo code of the worker (top panel) and CP-threads (bottom panel). During the checkpoint iteration, the worker thread signals the CP-thread that the checkpoint can be performed. The CP-thread then in turn starts writing the data to PFS.



(a) *checkpoint* iteration (b) *checkpoint* + 1 iteration (c) *checkpoint* + 2 iteration

Figure 5. Pointer switching mechanism for efficient in-memory checkpoint creation for stencil based algorithms. In a checkpoint iteration (a), A and B act as source and destination grids, respectively. B becomes the CP-grid and also serves as the source grid during $checkpoint + 1$ iteration (b). Afterwards, A and C act as source and destination grids respectively (c).

destination grids. By switching the grid pointers only, the time usually spent for creating the in-memory copy of the CP-grid from the currently updated grid can be completely avoided. Figure 5 depicts this pointer switching of the grids with respect to time-steps. In a checkpoint iteration (Fig. 5(a)), grids A and B serve as source and destination grids, respectively. After the update, B becomes the CP-grid and also serves as the source grid during the iteration $checkpoint + 1$ (Fig. 5(b)). The CP-thread starts to write B (CP-grid) to the file system. During the iteration $checkpoint + 2$ (Fig. 5(c)), grids A and C serve as source and destination grids, respectively.

It is worth highlighting that like other node-level C/R approaches, this one also requires additional memory in order to create the in-memory checkpoint. The size of this additional memory entirely depends on the amount of data the application has to save.

IV. Overhead Estimation

In a standard synchronous checkpointing method, each MPI process interrupts its computations after receiving a checkpoint signal. The computation is resumed after the checkpoint is fully written to PFS. Figure 6(a) shows a schematic timeline of an application with two synchronous checkpoints. The overhead is due to the time intervals of checkpoint I/O during which no computation can be performed. We define the following quantities:

$$\begin{aligned} t_{O,s} &= \text{overhead for synchronous checkpoints} \\ t_{CP,s} &= \text{duration of a synchronous checkpoint} \\ S_{CP} &= \text{size of a single checkpoint in bytes} \\ B_{IO} &= \text{I/O bandwidth to the file system in bytes/s} \\ B_M &= \text{memory bandwidth of a node in bytes/s} \\ n &= \text{number of checkpoints} \end{aligned}$$

The overhead in this case can be written as:

$$\begin{aligned} t_{O,s} &= n \cdot t_{CP,s} \\ t_{O,s} &= n \cdot \frac{S_{CP}}{B_{IO}} \end{aligned} \tag{1}$$

With weak scaling, the total checkpoint size increases linearly with the number of nodes. In contrast, usually the I/O bandwidth B_{IO} scales poorly with the number of nodes. Thus the overhead is directly proportional to the number of compute nodes.

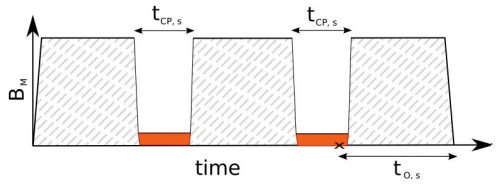
In the case of asynchronous checkpointing, the overhead is mainly due to competing memory accesses of the worker and CP-thread(s) for the duration of the checkpoint I/O. The scenario of asynchronous checkpoint is depicted in Fig. 6(b). We can calculate the overhead in case of asynchronous checkpointing. The model is based on the assumption that all I/O-time overlaps with computation (a single checkpoint time is less than the time between two consecutive checkpoints). We define the following notations:

$$\begin{aligned} t_{O,a} &= \text{overhead for asynchronous checkpoints} \\ t_{CP,a} &= \text{duration of an asynchronous checkpoint} \\ S_{CP,node} &= \text{checkpoint size per node in bytes} \\ B_{M,CP} &= \text{memory bandwidth used for checkpoint-I/O in bytes/s} \end{aligned}$$

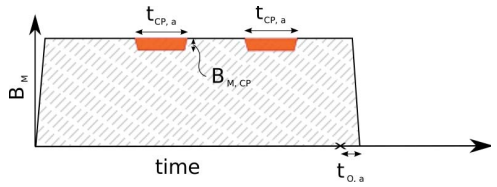
The overhead for asynchronous case can be formulated as

$$B_M \cdot t_{O,a} = n \cdot B_{M,CP} \cdot t_{CP,a}$$

For I/O purposes, the amount of data traffic (reads/writes) between memory and processes can be m times larger than the file size itself. This factor depends on the specific implementation of fwrite()/MPI_File_write*() functions and



(a) Synchronous checkpointing



(b) Asynchronous checkpointing

Figure 6. A memory bandwidth utilization model with checkpoints in synchronous and asynchronous cases. During a checkpoint, only a small portion of memory bandwidth is utilized for checkpoint I/O. The rest of the memory bandwidth resource remains idle in the case of synchronous checkpointing whereas it is utilized for computation in the case of asynchronous checkpointing.

the number of different buffers they use (our study reveals this factor to be between 5-7 for Open MPI). Thus,

$$B_{M,CP} = \frac{m \cdot S_{CP,node}}{t_{CP,a}}$$

The total overhead $t_{O,a}$ then is

$$t_{O,a} = \frac{m \cdot S_{CP,node}}{B_M} \cdot n \tag{2}$$

which remains a constant factor in the case of weak scaling. Thus the overhead is independent of the number of nodes. Furthermore, the checkpoint overhead becomes independent from the bandwidth to the PFS.

V. Test Bed

For performance evaluation, we have used RRZE's LiMa[1] cluster and HLRS's HERMIT[2] cluster.

LiMa: This cluster comprises 500 compute nodes equipped with two Intel Xeon 5650 "Westmere" CPUs (six physical cores, two-way SMT cores) running at the base frequency of 2.66 GHz. The ccNUMA system has two locality domains, each with 12 GB RAM (24 GB in total). The STREAM (scale) benchmark [18] achieves a bandwidth of around 40 GB/s (20 GB/s per socket). Simultaneous multithreading (SMT) and "Turbo Mode" are enabled. The system is equipped with QDR InfiniBand (IB) and GBit Ethernet interconnects. The cluster is connected to a Lustre PFS via the IB network. The aggregated bandwidth of the PFS is around 3 GB/s.

HERMIT: This is a CRAY XE6 cluster with 3552 nodes, each with dual socket AMD Opteron 6278 (Interlagos) processors. Each processor consists of 16 cores running at 2.3 GHz. All nodes are equipped with at least 32 GB of RAM. The system uses the CRAY Gemini interconnect. It is connected to a Lustre PFS with an aggregated I/O bandwidth of approximately 150 GB/s.

For benchmarking, the Open MPI library (version 1.6) was used. We have used a prototype CFD solver based on the lattice Boltzmann method, which is a stencil type algorithm with toggle grids [19]. The runtime of the benchmark depends on the time-steps and the number of domain cells. The number of domain cells also determines the checkpoint size. In order to simulate the worst case scenario, memory is used to its maximum size thus creating large checkpoints. Both the SCR-library and our implementation can easily be adapted for other kinds of algorithms.

VI. Results

We compare three different C/R strategies and their relative overheads. These are node-level checkpointing, synchronous, and asynchronous PFS-level checkpointing.

checkpoint (`SCR_FLUSH`=2). In this case, each global checkpoint to PFS adds 25-35 times more overhead than creating only node-level checkpoint.

In order to validate our model of checkpointing overhead presented in Sect. IV, we use the likwid tool [20] on LiMa to analyze the memory bandwidth pattern during the application run with checkpoints made on PFS. Figure 7 shows the memory bandwidth pattern of a single socket of a LiMa node with synchronous and asynchronous checkpointing, respectively. Two checkpoints are written to LiMa's PFS, each with 6.1 GB. The memory bandwidth of each socket is measured using `likwid-perfctr` every 500 ms. At the time of creating a synchronous checkpoint, the used memory bandwidth drops to a low level due to the much smaller I/O bandwidth to the PFS. The durations for checkpointing during which no work is performed, add up to make the large overhead. Furthermore, the effective I/O bandwidth per node (which is utilized for checkpointing) decreases with increasing number of nodes and thus the resulting overhead gets larger.

On the other hand, with asynchronous checkpointing, the memory bandwidth utilization does not drop during checkpoint creation as both the checkpointing and the computations are being carried out at the same time. This makes the asynchronous technique an efficient way to create

[1]LiMa cluster at the Erlangen Regional Computing Center (RRZE): http://www.hpc.rrze.fau.de/systeme/lima-cluster.shtml

[2]HERMIT cluster at the High Performance Computing Center Stuttgart (HLRS): http://www.hlrs.de/systems/platforms/cray-xe6-hermit

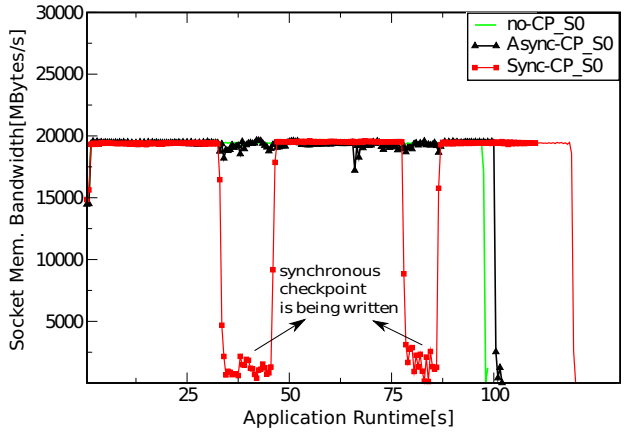


Figure 7. Bandwidth utilization of a single socket of a LiMa node during an application run with asynchronous (black) and synchronous (red) checkpointing. Two checkpoints are taken to LiMa's PFS, each of 6.1 GB. In case of asynchronous checkpointing, checkpoint I/O is overlapped with computation which keeps the memory bandwidth utilization at maximum level all the time. As a result, the overall runtime of the application is also reduced. For reference, the bandwidth utilization in case of no checkpointing is also shown (green).

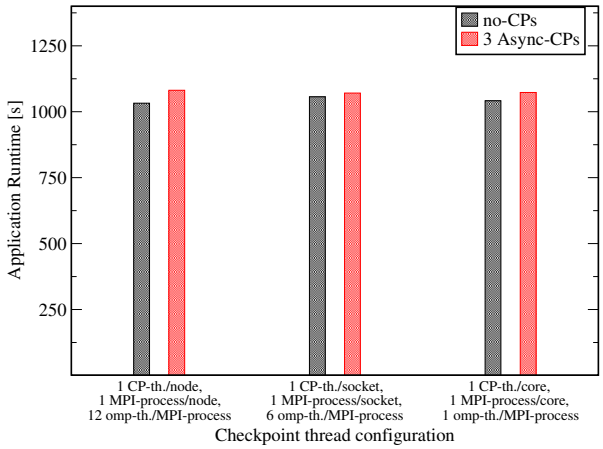


Figure 8. Runtime comparison of different checkpoint thread configuration schemes on LiMa. 32 nodes are utilized with the checkpoint size of 200 GB each.

checkpoints with minimal overhead. Our theoretical model for asynchronous checkpointing (2) suggests an overhead of 2.2s ($n=2$, $S_{CP,node}=6.25$ GB, $B_M=40$ GB/s, $m=7$), which is close to the actual overhead of 2.6s in this case.

In Sec. III, we have described three possibilities for having dedicated checkpoint threads in the application, i.e., 1 CP-thread/core, 1 CP-thread/socket and 1 CP-thread/node. Figure 8 shows the performance comparison of these three approaches with and without writing checkpoints. In this case, 32 LiMa nodes have been used with a domain size of $336 \times 336 \times 10752$ cells and 2000 time-steps, resulting in a total of 200 GB per checkpoint. When no checkpoint is created, the performance remains similar in all three cases since dedicated CP-threads are idle and pinned to one of the free SMT cores of a physical core. The creation of three asynchronous checkpoints adds a little overhead in all the cases, but the overhead is similar for each configuration. Because of its similarity with the pure MPI application, we have selected "1 MPI-process/core (1 CP-thread/core)" scheme for further analysis.

A comparison of the overhead between a naïve synchronous PFS-level checkpointing, our presented asynchronous technique and SCR checkpointing (partner-level, PFS-level) for the LiMa cluster is shown in Fig. 9. In this case, 128 nodes are used resulting in 1536 MPI processes in total. The domain consists of $288 \times 288 \times 36864$ cells, which gives an aggregated checkpoint of size 510 GB. In case of no checkpoint, the performance of all configurations is similar. The slight variation in the runtime exists due to network performance fluctuation. As expected, the time for I/O increases linearly for the case of synchronous and SCR PFS-level checkpoints. On the other hand, the checkpoint I/O causes only negligible overhead for the case of asynchronous PFS-level and partner-level checkpoints. A single checkpoint introduces an overhead of 13% in case of synchronous PFS-level checkpoints, whereas it is 1.3% for the case of asynchronous PFS-level checkpoints and 1% for creating partner-level checkpoints. For the case of four checkpoints, the overhead becomes 51% for synchronous PFS-level checkpoints, 2.5% for asynchronous PFS-level checkpoints, and 1.4% for partner-level checkpoints. Although the overhead for partner-level checkpoint is smallest, yet in order to create global PFS-level checkpoints, the asynchronous checkpointing technique proves to be far more economical than the usual synchronous checkpointing.

On the HERMIT system we were not able to install the SCR library. Thus, Fig. 10 shows the comparison only between synchronous and asynchronous PFS-level checkpointing for the HERMIT cluster. In this case, 256 nodes are used with "1 MPI-process/core (1 CP-thread/core)" resulting in a total of 8192 processes running for 8000 time-steps. For the case of asynchronous checkpointing, each core is oversubscribed with an additional CP-thread. The domain size is $368 \times 368 \times 94208$ cells, which gives an aggregated checkpoint size of 2.3 TB. Each synchronous checkpoint adds 5.6% overhead to the overall runtime. On the other hand, each asynchronous checkpoint only adds 0.2% overhead. The overhead for four checkpoints is 23% in the synchronous and 0.85% in the asynchronous case.

Due to the fact that with asynchronous checkpointing the time needed for I/O is overlapped with computation, an

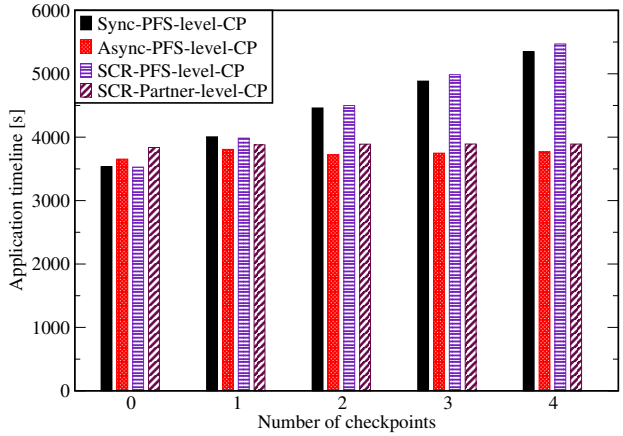


Figure 9. Benchmark of overhead analysis on the LiMa cluster for asynchronous PFS-level checkpointing and synchronous PFS-level, node-level checkpointing. 128 nodes are used in this case with "1 MPI-process/core" making a total of 1536 MPI processes. The aggregated size of each checkpoint is 510 GB. For synchronous PFS level checkpointing, the overhead increases linearly with the number of checkpoints. On the other hand, the overhead is minimal in case of node-level checkpointing and asynchronous PFS-level checkpointing.

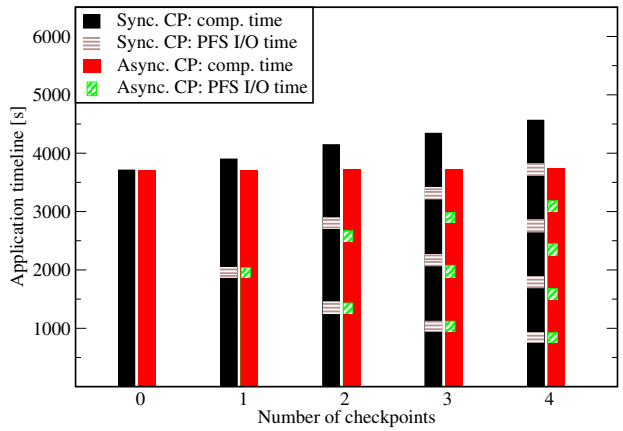


Figure 10. Benchmark of overhead analysis on the HERMIT cluster for synchronous and asynchronous PFS-level checkpointing. 256 nodes are used in this case with "1 MPI-process/core" making a total of 8192 MPI processes. The aggregated size of each checkpoint is 2.3 TB which is stored on a Lustre based file system. The overhead increases linearly for the synchronous case while it remains minimal for asynchronous case.

upper limit for the number of low cost checkpoints exist. This can be calculated as

$$\text{max. number of async. CPs} = \frac{\text{Total runtime without CPs}}{\text{I/O time for a single CP}}.$$

Exceeding this number of checkpoints will introduce high overhead, as the next checkpoint will have to wait for the completion of the previous one. As with synchronous checkpointing, this model will be feasible as long as MTTF of the system remains greater than the time to make an asynchronous checkpoint.

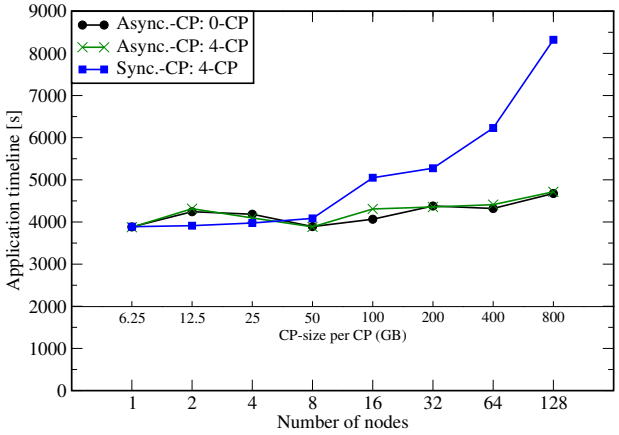


Figure 11. A weak scaling study of checkpointing overhead with four synchronous vs. asynchronous checkpoints on the PFS.

In order to determine the scaling capability of our method, we have performed a weak scaling test on up to 128 nodes of the LiMa cluster. Figure 11 depicts a relative comparison between synchronous vs. asynchronous checkpoints in the weak scaling scenario with four checkpoints. The overhead of synchronous checkpointing increases proportional to the number of nodes and the number of checkpoints. On the other hand, the overhead of asynchronous checkpointing remains nearly constant. This makes asynchronous checkpointing suitable option for large applications.

We have also tested Open MPI's capability of stage checkpoint, in which a checkpoint is first stored on an on-node storage and then copied to the file system asynchronously. A comparison of the performance overhead between Open MPI's stage mechanism and our presented asynchronous checkpointing technique is shown in Fig. 12 for checkpoints taken on LiMa's PFS. The "filem_rsh_max_incoming" (number of simultaneous copy operation from local checkpoint to global checkpoint directory) parameter is made equal to the number of MPI processes in order to have the same effect as our asynchronous application. In this case, four nodes are used with an aggregated checkpoint size of 25 GB each. Our presented checkpointing technique shows a clear advantage over the SELF library stage checkpointing option. The introduction of checkpoint ability by enabling "-am ft-enable-cr" flag itself introduces 3.2% overhead in this case. Each SELF-stage checkpoint introduces additional 4% overhead. In case of four checkpoints, the overhead for SELF-stage checkpointing is 18%, whereas it is only 0.35% for our presented technique.

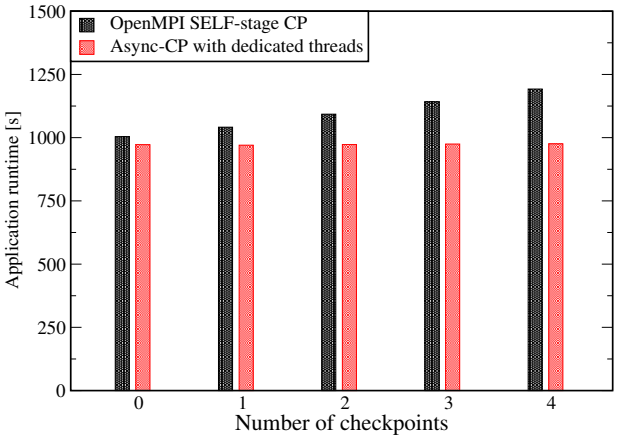


Figure 12. Overhead comparison between Open MPI stage option and our asynchronous checkpointing technique. In this case, 4 LiMa nodes are used with an aggregated checkpoint size of 25 GB each.

VII. Summary

The checkpointing overhead is one of the primary reasons that hinder the wide use of the checkpoint/restart technique to introduce fault tolerance in applications. In this paper, we studied the asynchronous checkpointing technique by the introduction of dedicated checkpointing threads on the application-level. We have followed a hybrid MPI/OpenMP approach with multi-level checkpointing. We have formulated a theoretical model to calculate the overhead of asynchronous and synchronous checkpointing. Using nested OpenMP parallel approach, we compared three different scenarios of checkpointing threads (1 CP-thread/node, 1 CP-thread/socket, 1 CP-thread/core) and found their performance to be similar. A comparison between the asynchronous checkpointing with the synchronous checkpointing and a node-level checkpoint approach is preformed. Although the node-level checkpoint approach costs least overhead, an asynchronous checkpointing technique with a dedicated CP-thread was found to be an optimal approach to create parallel file system level checkpoints. The overhead introduced by our asynchronous checkpointing method is similar to node-level checkpointing. We have also compared our method with Open MPI's Stage checkpointing mechanism and found that our method has clear advantages in terms of performance. The idea of asynchronous checkpointing can be extended to a large number of applications and significant advantage can be gained in terms of checkpointing overhead. The idea is particularly useful for applications that have to checkpoint large amounts of data.

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) and partly by Federal Ministry of Education and Research (BMBF) under project "A Fault Tolerant Environment for Peta-scale MPI-solvers" (FETOL) (grant No. 01IH11011C).

References

[1] Top500, "List of 500 most powerful computers."

[2] J. Dongarra, P. Beckman, and et al., "The International Exascale Software Roadmap," *International Journal of High Performance Computer Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[3] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," in *Proceedings of the 2003 International Conference on Computational Science*, ser. ICCS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 3–12.

[4] J. Hursey, "Coordinated Checkpoint/Restart Process Fault Tolerance for MPI Applications on HPC Systems," Ph.D. dissertation, Indiana University, Bloomington, IN, USA, July 2010.

[5] Open MPI, http://www.open-mpi.org/.

[6] Fault Tolerance Research at Open Systems Laboratory, http://osl.iu.edu/research/ft/ompi-cr/.

[7] BLCR, "Berkeley Lab Checkpoint/Restart," https://ftg.lbl.gov/projects/CheckpointRestart/.

[8] J. Cornwell and A. Kongmunvattana, "Optimized I/O Operations for Checkpoint Creation in BLCR," in *Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering*, 2011, pp. 284–289.

[9] S. Mishra, "Design and Implementation of Process Migration and Cloning in BLCR," Master's thesis, North Carolina State University, North Carolina, USA, 2011.

[10] J. Daly et al., "Inter-Agency Workshop on HPC Resilience at Extreme Scale," http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf, Tech. Rep., Feb. 2012.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for HPC, Networking, Storage and Analysis*, Washington,DC, USA, 2010, pp. 1–11.

[12] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 19:1–19:10.

[13] J. Hursey and A. Lumsdaine, "A composable runtime recovery policy framework supporting resilient HPC applications," Indiana University, Bloomington, Indiana, USA, Tech. Rep. TR686, August 2010.

[14] S. More, A. Choudhary, and I. Foster, "MTIO a multi-threaded parallel I/O system," in *In Proceedings of the Eleventh International Parallel Processing Symposium*, 1997, pp. 368–373.

[15] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 39–48.

[16] P. M. Dickens, "Improving collective I/O performance using threads," in *In Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999, pp. 38–45.

[17] F. Shahzad, M. Wittmann, T. Zeiser, and G. Wellein, "Asynchronous checkpointing by dedicated checkpoint threads," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 289–290.

[18] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007.

[19] G. Wellein, T. Zeiser, G. Hager, and S. Donath, "On the single processor performance of simple lattice Boltzmann kernels," *Computers & Fluids*, vol. 35, no. 8–9, pp. 910–919, Nov. 2006.

[20] LIKWID tool suite, http://code.google.com/p/likwid/.