VERIFIABLE APPLICATION-LEVEL CHECKPOINT AND RESTART FRAMEWORK FOR PARALLEL COMPUTING

I. Gankevich[a], I. Petriakov, A. Gavrikov, D. Tereshchenko, G.Mozhaiskii

Saint Petersburg State University, 13B Universitetskaya Emb., St Petersburg 199034, Russia

E-mail: [a] i.gankevich@spbu.ru

Fault tolerance of parallel and distributed applications is one of the concerns that becomes topical for large computer clusters and large distributed systems. For a long time the common solution to this problem was checkpoint and restart mechanisms implemented on operating system level, however, they are inefficient for large systems and now application-level checkpoint and restart is considered as a more efficient alternative. In this paper we implement application-level checkpoint and restart manually for the well-known parallel computing benchmarks to evaluate this alternative approach. We measure the overheads introduced by creating and restarting from a checkpoint, and the amount of effort that is needed to implement and verify the correctness of the resulting programme. Based on the results we propose generic framework for application-level checkpointing that simplifies the process and allows to verify that the application gives correct output when restarted from any checkpoint.

Keywords: fault tolerance, message passing interface, MPI, miniFE, NAS parallel benchmarks, DMTCP

Ivan Gankevich, Ivan Petriakov, Anton Gavrikov, Dmitrii Tereshchenko, Gleb Mozhaiskii

1. Introduction

Current parallel computing technologies do not have automatic fault tolerance built in, and researchers rely on external tools and application developers to make applications tolerant to cluster node failures. Popular message passing interface (MPI) provides means of communication but does not provide means to manage application state. As a result the state of many applications that use MPI is stored in local and global variables that are not managed by MPI and can not be automatically saved to and restored from the file (or any other medium). This deficiency lead to the creation of external tools that periodically stop MPI application, dump memory contents of all parallel processes to the file and resume the execution [1,2].

This technique is called *system-level* checkpoint and restart, and it has obvious disadvantage of being inefficient for the large number of parallel processes and saving too much data if the checkpoint is triggered during some peak memory usage application phase. An alternative approach is to modify the application to save all the variables to the file every *n*-th iteration of the main programme loop and restore them from the file before the main programme loop starts. This approach is called *application-level* checkpoint and restart, and it is more efficient that system-level checkpoints because it saves the minimum amount of data that is required to restore the application from the file.

In this paper we evaluate application-level and system-level checkpoint and restart on a set of parallel applications. We implement application-level checkpoints for NAS Parallel Benchmarks [3] and miniFE [4] applications, measure the overhead and programming effort, and compare and contrast them to system-level checkpoints created with DMTCP [2]. Based on the experience that we obtained we write MPI-Checkpoint library that contains a set of routines that can be added to MPI to help manage application global state and implement application-level checkpoints.

2. MPI-Checkpoint library

The closest library that provides checkpoint and restart functionality for MPI applications is CRAFT [5], however, this library is written in C++ and is not compatible with C and Fortran applications. Our approach is to reuse functionality provided by MPI to simplify our library: we can reuse data type handling and global process communication. From this perspective, our library can be considered as a set of routines that can be added to MPI to provide application state management via checkpoints, rather than a standalone full-featured checkpoint library.

Our library provides the following routines:

- *MPI_Checkpoint_create* — open checkpoint file for writing;
- *MPI_Checkpoint_write* — write application state to the file;
- *MPI_Checkpoint_restore* — open checkpoint file for reading;
- *MPI_Checkpoint_read* — read application state from the file;
- *MPI_Checkpoint_close* — close the file.

They are used as follows. Every *n*-th iteration of the main loop each MPI process creates itss own checkpoint file and writes application state (the values of all relevant local and global variables) to this file. All files are stored in the same directory and their names equal the ranks of the corresponding MPI processes. Before the main loop each MPI process tries to restore from the checkpoint file: on success the values of all relevant variables are read from the file and the loop starts from the corresponding iteration.

From a technical standpoint, the public interface of the library permits the usage of any medium to store checkpoints (file systems, main memory of spare nodes etc.), but reference implementation supports only file systems. Input/output is implemented using memory-mapped files and is efficient for the large files as the old pages that has already been read from/written to the file are discarded from the memory.

From the users' perspective, in order to restore from the checkpoint the environment variable *MPI_CHECKPOINT* should be set to the file system path of the checkpoint directory. Since every MPI process works with its own checkpoint file, they can be stored either in parallel or local file system. If the local file system is used, the processes should be restarted on *exactly* the same cluster nodes to be able to read from these files. The advantage of this approach, however, is the fact that it

may be more scalable than parallel file system, because the cluster network is not used for the input/output.

3. Benchmarks

Using MPI-Checkpoint library we implemented application-level checkpoints for NAS parallel benchmarks and miniFE. Our approach is based on the fact that most parallel batch processing applications follow bulk-synchronous parallel model [6]: they are organised in a series of sequential steps (main loop) that are internally parallel. After each step there is a synchronisation point and here we create checkpoint file. We restore from the checkpoint file before the main loop. The disadvantage of this approach is that the initialisation of the programme (i.e. the code before the main loop) is performed once again before the restoration. The approach is presented in listing 1.

```
int step_min = 0;
MPI_Checkpoint checkpoint = MPI_CHECKPOINT_NULL;
int ret = MPI_Checkpoint_restore(MPI_COMM_WORLD, &checkpoint);
if (ret == MPI_SUCCESS) {
    MPI_Checkpoint_read(checkpoint, &step_min, 1, MPI_INT);
    ... // read more variables
    MPI_Checkpoint_close(&checkpoint);
}
for (int step=step_min; step<=step_max; ++step) {
    ... // some application logic code
    int ret = MPI_Checkpoint_create(MPI_COMM_WORLD, &checkpoint);
    if (ret == MPI_SUCCESS) {
        MPI_Checkpoint_write(checkpoint, &step, 1, MPI_INT);
        ... // write more variables
        MPI_Checkpoint_close(&checkpoint);
    }
}
```

Listing 1. Main loop augmented with application-level checkpoint and restart functionality. Public library calls are marked with blue.

Using this approach we implemented checkpoints for CG, EP, FT, IS, LU, MG, BT benchmarks and for the reference implementation of miniFE, and it took moderate amount of effort. We stored initial code without checkpoints in Git [7] and then in each commit we implemented a checkpoint for a particular benchmark. According to Git log[1] we spent only four working days for *all* the benchmarks to write and verify all the code that is needed for the checkpoints, the rest of the time was spent on improvement of the library public interface, implementing Fortran public interface, compression and memory-mapped input/output.

We verified the correctness of the application that was restarted from the checkpoint by using the automated verification code that is built in the NAS parallel benchmarks and by comparing the residual of miniFE benchmark. If we produce a checkpoint every iteration we get a set of directories containing checkpoint files (one directory for each iteration). Then we restart the application using each such directory and perform verification of the application output. If all verifications succeed, then application-level checkpoints code is correct (i.e. we saved all the required variables). For many real-world applications verification can be implemented as bytewise comparison of the output data; for applications that use pseudo-random numbers integral properties of the output can be used for verification.

In addition to application-level checkpoints we implemented DMTCP checkpoints in our library. When DMTCP mode is enabled, the library on each call to *MPI_Checkpoint_create* instructs the coordinator process to create full application memory dump. *MPI_Checkpoint_restore* is a no-op in this mode since the restoration happens using the shell script generated by DMTCP.

[1] https://github.com/igankevich/npb-checkpoints, https://github.com/igankevich/miniFE-checkpoints

4. Results

We ran performance benchmarks multiple times for all applications, for both DMTCP and MPI-Checkpoint modes with varying number of MPI processes. We measured checkpoint size on the disk, checkpoint creation time (overhead) and total execution time of the application. We used parallel file system GlusterFS, that is deployed on the same cluster nodes where the applications run, to store the checkpoints. Full testbed configuration is listed in table 1.

Table 1. Hardware and software configuration

DMTCP	version 2.6.0, arguments: *--no-gzip*
MPICH	version 3.3.2, environment variables: *HYDRA_RMK=user*
NPB	version 3.4, class C
miniFE	version 2.0, parameters: *nx=300, ny=300, nz=300*
Compiler	GCC 7.5.0, compilation flags: *-O3 -march=native*
Cluster	6 nodes, 2 processors per node, 4 cores per processor, 2 threads per core (96 threads in total), 1 Gbit network switch
GlusterFS	version 8.0, two replicas for each file (the same nodes and network switch as the cluster)

Performance benchmarks showed that the total size for both MPI and DMTCP checkpoints grows linearly with the number of MPI processes: the growth rates for miniFE application are 0.2% and 4% per node (16 parallel processes) respectively (see fig.1). For miniFE both MPI and DMTCP checkpoint creation time decreases with the number of processes (see fig.1); this can be explained by the fact that the network switch single port throughput is fully utilised, but the overall switch throughput is not (its utilisation increases with the number of ports used). For all NAS parallel benchmarks (except MG) this time decreases when we go from single node to two-node configuration, and then increases (see fig.2); the decrease in this case can be explained the same way. The increase after two nodes can be explained by the fact that the parameters of NAS parallel benchmarks are determined from the number of MPI processes.

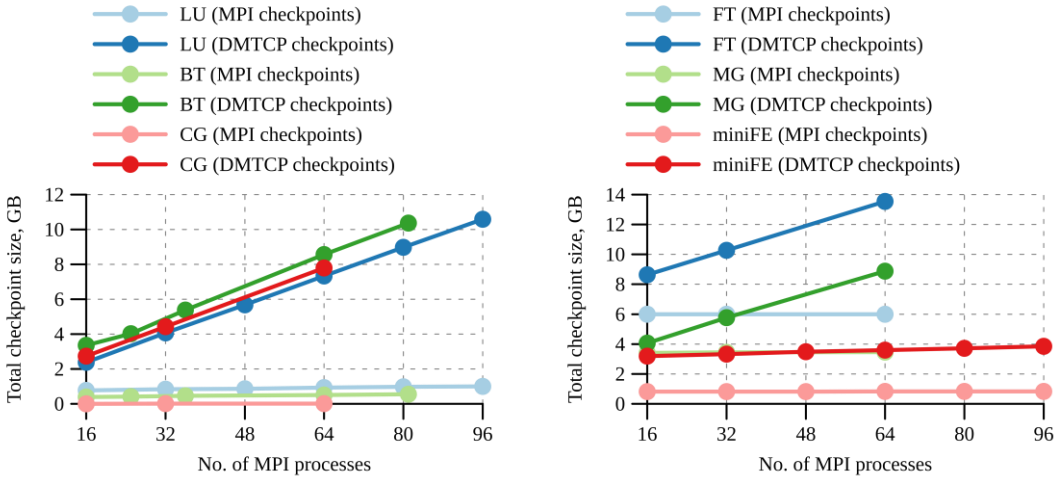

Figure 1. The total size of checkpoint files for DMTCP and MPI

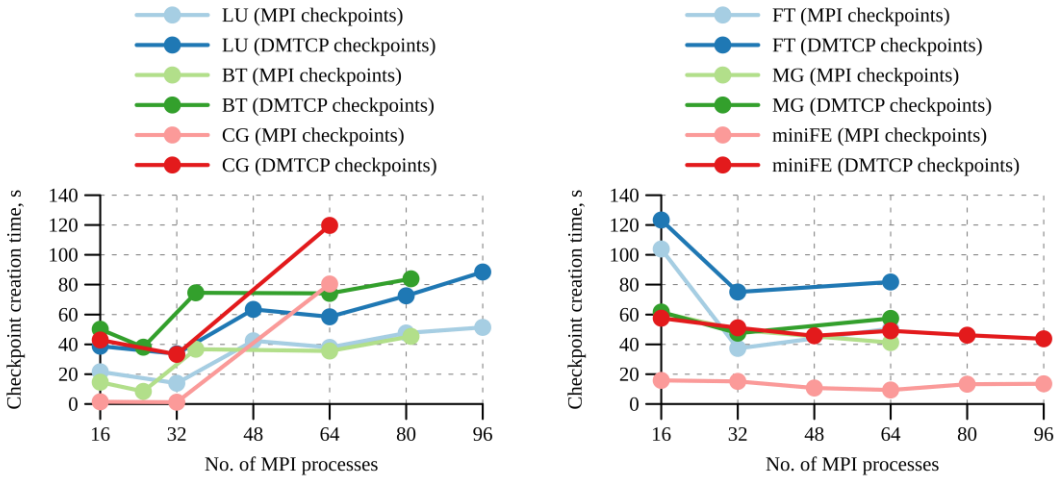

Figure 2. MPI and DMTCP checkpoint creation time for NAS parallel benchmarks and miniFE

5. Conclusion

We evaluated application-level and system-level checkpoint and restart on a set of benchmarks that replicate behaviour of real-world applications. Contrary to our expectations we found out that it takes little programming effort to implement application-level checkpoints for someone who sees the source code of the application for the first time. Our performance benchmarks confirmed that application-level checkpoints are much smaller in size and take less time to create compared to system-level checkpoints. Our cluster was too small to confirm that the time needed to create checkpoint files increases with the number of nodes (our benchmarks showed that it actually decreases or does not change much). Based on our experience we proposed minimal set of routines that can be added to MPI to create application level checkpoints.

We believe that the effort that application developers need to put into implementing application-level checkpoints is much smaller than the effort application users put into configuring system-level checkpoints: during our benchmarks we encountered several cases when the programme restarted from DMTCP checkpoint hanged, DMTCP does not work with OpenMPI library (we did not find working solution of this problem), DMTCP does not work if one wants to restart the application on a different set of nodes. For efficiency and reliability reasons developers of new MPI applications should consider implementing application-level checkpoints. Hopefully, this paper and our public-domain library[2] would help in this regard.

6. Acknowledgement

This work is supported by Council for grants of the President of the Russian Federation (grant no. MK-383.2020.9).

References

[1] Hargrove P. H., Duell J. C. Berkeley lab checkpoint/restart (BLCR) for Linux clusters //Journal of Physics: Conference Series. – IOP Publishing, 2006. – Vol. 46. – Issue 1. – P. 067.

[2] Ansel J., Arya K., Cooperman G. DMTCP: Transparent checkpointing for cluster computations and the desktop //2009 IEEE International Symposium on Parallel & Distributed Processing. – IEEE, 2009. – P. 1-12.

[3] Van der Wijngaart R. F., Wong P. NAS parallel benchmarks version 2.4. – 2002.

[2] https://github.com/igankevich/mpi-checkpoint

[4] Heroux M. A. et al. Improving performance via mini-applications //Sandia National Laboratories, Tech. Rep. SAND2009-5574. – 2009. – Vol. 3.

[5] Shahzad F. et al. CRAFT: A library for easier application-level checkpoint/restart and automatic fault tolerance //IEEE Transactions on Parallel and Distributed Systems. – 2018. – Vol. 30. – Issue 3. – P. 501-514.

[6] Valiant L. G. A bridging model for parallel computation //Communications of the ACM. – 1990. – Vol. 33. – Issue 8. – P. 103-111.

[7] Torvalds, L., Hamano, J.: Git: fast version control system (2010). http://git-scm.com.