A Tool for Semi-Automatic Application-Level Checkpointing

Trung Nguyen Ba
Texas Advanced Computing Center
University of Texas at Austin
Austin, Texas
tantrung@gmail.com

Ritu Arora
Texas Advanced Computing Center
University of Texas at Austin
Austin, Texas
rauta@tacc.utexas.edu

***Abstract*—Computational jobs running on supercomputing resources at open-science data centers are often limited to a maximum number of compute-nodes and wall-clock time. However, many jobs need longer than the maximum allowed wall-clock time to complete. To overcome this limitation, applications can be checkpointed such that their execution state is saved before they time-out from the job-queue. Using their saved state, the applications can resume their computation from the point where they stopped in the previous run. When the checkpointing-and-restart mechanism is built within the application, it is called Application-Level Checkpointing (ALC). We are developing a tool for semi-automatic ALC of existing applications without requiring any manual reengineering of the applications. The memory footprint of the checkpoints written using our tool is small. Applications written in C/C++/MPI/OpenMP will be supported in the upcoming release of our tool, and in future, the tool will support Fortran and Python applications too.**

Keywords—application-level checkpointing; semi-automatic; interactive; job queue

I. INTRODUCTION

Supercomputing resources at open-science data centers are shared amongst multiple users. To ensure fair-share of computational and storage resources to all users at all times, these resources have policies governing their usage. For example, the Stampede supercomputer at the Texas Advanced Computing Center (TACC) has several job queues, each with different characteristics. A job running in the "normal" queue can use up to a maximum of 256 nodes and can run for up to 48 hours [1]. However, many jobs need longer than the maximum allowed wall-clock time to complete. Although extending the maximum allowed queue-time for such jobs occasionally is an option (under emergency situations), it needs the intervention of system administrators. To overcome the limitations of the job queue policies, users are typically advised to use checkpointing. With checkpointing, the execution state of applications can be saved to the disk before they time-out from the job-queue. Using their saved state, the applications can resume their computation from the point where they stopped in the previous run instead of computing from scratch. When the checkpointing-and-restart mechanism is built within the application itself, it is called Application-Level Checkpointing (ALC) [2]. While we use ALC for pausing and resuming computational tasks, it is useful in other scenarios as well (*e.g.*, developing fault-tolerant applications).

ALC helps in periodically saving the state of only critical variables in an application. Critical variables are those variables in an application whose state or values are sufficient to recreate the execution state of the entire application. In the event of any interruption during its execution, the application can be restarted using the saved state of the critical variables in checkpointing files (instead of using their initial state).

As compared to system-level or library-level or kernel-level checkpointing, ALC can involve smaller memory footprint for writing checkpointing files (saves the state of critical variables only), and does not require any special system-level permissions. These features are especially important for users of open-science supercomputing resources because of storage quota limits and restricted permissions for system access.

Manual reengineering of large applications to take advantage of ALC (as described above) could be time-consuming and tedious for many users. None of the exisitng solutions for checkpointing was suitable for our purposes. We needed a solution that was light-weight, gave control about "what to checkpoint" to the user, did not require any special privileges, had small memory footprint of checkpointing files, and ensured high I/O throughput in case of writing checkpoints at a high frequency. Therefore, we are developing a new tool for assisting users to take advantage of ALC without compromising their productivity. The tool currently supports the ALC of applications written in C/C++ and will be extended to support Fortran and Python applications in future. We are currently implementing the support for writing distributed and centralized checkpoints from MPI [2] and OpenMP programs. Our tool is sensitive to the underlying filesystem. For example, when running on a Lustre filesystem, the tool will be able to set appropriate stripe count and stripe size for high throughput [3]. This will be useful when saving data in large arrays.

II. OVERVIEW OF THE TOOL

The tool is built on top of the ROSE source-to-source compiler [4]. The translator, which is the core of the tool, contains the code transformation rules and invokes the ROSE compiler to modify the abstract syntax tree of the input source code. The modifications made to the input code mainly include (1) insertion of code to embed the checkpointing-and-restart logic, (2) making changes to for-loops, (3) altering the order of function calls where required, and (4) doing consistency checks and showing warning messages to the users as needed. As the tool is semi-automatic, it interacts with the user to capture the specifications for checkpointing. We assume that the user is familiar with the code of their application. The user provides specifications about (1) what to checkpoint, (2) where to checkpoint, and (3) the frequency of checkpointing. On the

basis of the specifications provided by the user, and the results of its static code analysis, the tool modifies the input source code provided by the user. For checkpointing MPI programs, the user is prompted for additional input. They specify whether they prefer a centralized or distributed checkpoint, and provide information about the filesystem.

The tool is launched from the command-line and the user provides the name of the application file that should be checkpointed. The tool then prompts the user for further information as required, and produces a checkpointed version of the application as its output. The screen-shot of tool in action is shown in Figure 1. The tool presents a list of critical variables for checkpointing to the user. The user can chose to decline checkpointing the variables short-listed by the tool.

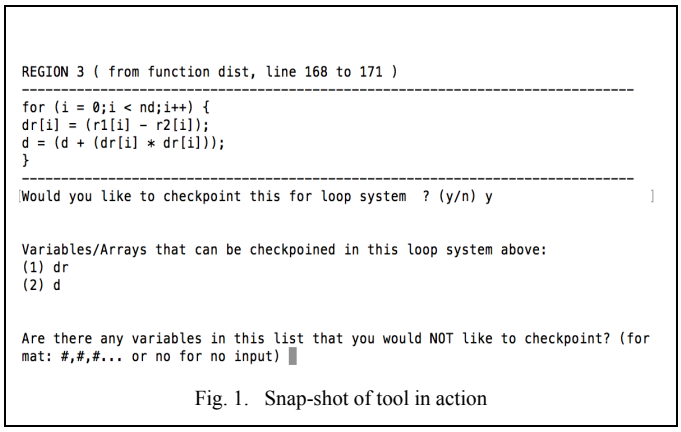

Fig. 1. Snap-shot of tool in action

III. Results and Discussion

A serial version of a Molecular Dynamics (MD) simulation application is used to demonstrate the functionality and results of our tool for ALC. The application was checkpointed manually and semi-automatically (using our tool). It was run on the Stampede supercomputer at TACC using a single core. As can be noticed from the results shown in Table 1, there is no significant difference in the performance of the two versions of the checkpointed code. There is no difference in accuracy of the results in the non-checkpointed and the checkpointed versions of the program. It took under two minutes to checkpoint this MD application.

Table 1. Run-time Comparison of Checkpointed MD Code

Versions	100 iterations	400 iterations	700 iterations	1000 iterations
Manually checkpointed	8.22161 (s)	32.057214 (s)	55.680952 (s)	79.488127 (s)
Semi-automatically checkpointed	8.145265 (s)	31.541982 (s)	56.271514 (s)	78.4087 (s)

IV. Related Work

During the intial phase of the research related to our tool, a Domain-Sepcific Language (DSL) for ALC was developed [2]. Though the DSL-based approach for ALC was effective and easy to use, it was based upon a commercial source-to-source compiler. However, we needed a non-commercial solution.

Distributed Multi-Threaded Checkpointing (DMTCP) [5] saves the state of a distributed program spread across multiple machines. It saves the state of the entire program. As compared to it, our tool checkpoints an application as per the specification provided by the user, and thus provides a fine-grained control to the user on what to checkpoint. Our tool saves the state of only the critical application variables and hence, entails less amount of time in doing I/O for reading and writing checkpointing data.

Bronovetsky *et al* [6,7] use a compiler technique to analyze the source code for checkpointing. Unlike our approach of checkpointing only critical variables, they checkpoint the entire state of the program. The users of their program need to manually modify their applications to specify the checkpoint, while in our approach, users can select the positions where checkpoint blocks should be inserted while working with the interactive interface.

V. Current and Future Work

We are currently implementing the features to support the checkpointing MPI and OpenMP applications. Currently, the tool cannot automatically analyze the user-defined header files included in the program. This limitation comes from the ROSE compiler and we have found a work-around for it. In future, we will support checkpointing Fortran and Python applications, and will simplify the mechanism to specify checkpoints.

Acknowledgment

We are grateful to TACC and XSEDE for providing the resources for this project. XSEDE is funded by the National Science Foundation (NSF) through award ACI-1053575.

References

[1] Stampede User-Guide, website accessed on July 22, 2016: https://portal.tacc.utexas.edu/user-guides/stampede

[2] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. 2011. A technique for non-invasive application-level checkpointing. J. Supercomput. 57, 3 (September 2011), 227-255. DOI=http://dx.doi.org/10.1007/s11227-010-0383-5

[3] Introduction to Parallel I/O, Ritu Arora, and Si Liu, website accessed on July 22, 2016: https://sea.ucar.edu/sites/default/files/PIO-SEA2015.pdf

[4] ROSE compiler infrastructure, website accessed on July 22, 1016: http://rosecompiler.org/

[5] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-12. DOI=10.1109/IPDPS.2009.5161063 http://dx.doi.org/10.1109/IPDPS.2009.5161063

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. 2004. Application-level checkpointing for shared memory programs. SIGOPS Oper. Syst. Rev. 38, 5 (October 2004), 235-247. DOI=http://dx.doi.org/10.1145/1037949.1024421

[7] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Radu Rugina. 2007. Compiler-Enhanced Incremental Checkpointing. In Languages and Compilers for Parallel Computing, Vikram Adve, María Jesús Garzarán, and Paul Petersen (Eds.). Lecture Notes In Computer Science, Vol. 5234. Springer-Verlag, Berlin, Heidelberg 1-15. DOI=http://dx.doi.org/10.1007/978-3-540-85261-2_1