Best Practices and Issues in Developing High-Productivity Software Tools for Supporting Computational and Data-Intensive Research

Ritu Arora, Robert McLay
Texas Advanced Computing Center, University of Texas at Austin

Abstract: High-productivity software tools are those tools that help their users in increasing their level of output while decreasing the investment of time and effort. Examples of such tools for Computational and Data-Intensive (CDI) research include, the Interactive Tool for Application-Level Checkpointing (ITALC) [1] that helps in reengineering existing code to make it checkpointable, Interactive Parallelization Tool (IPT) [2] for teaching parallel programming, and the XALT tool for helping system administrators in tracking user environments on large-scale systems, and detecting compile- or run-time problems [3]. In this paper, we briefly describe the best practices and issues in developing such high-productivity software tools.

Best Practices: Developing high-productivity software typically entails developing abstractions, and this can be an effort-intensive activity. As the hardware platforms on which the CDI applications typically run have a short life-span of 4-5 years, it is prudent to "future-proof" the applications so that they continue to be usable and scalable on the future hardware platforms and hence enjoy a long-life. There are multiple approaches for future-proofing the applications from the software design to the deployment stage. At the software design stage, one can consider adopting the software engineering principles such as, **the Open-Close Principle (OCP)**, as per which, the software is designed such that it is closed for modification but open for extension [4]. For building, deploying and distributing self-contained applications that can run on different but comparable hardware architectures, one can consider **containerization**. For scalability, the applications can be designed and developed to work in **distributed computing environment**s.

Issues: Designing "future-proof" software can be difficult. As one of our colleagues said: "one of the issues is **not thinking exponentially**". There are at least two instances where one of the authors on this paper missed handling exponentially large numbers. In the first case, a parallel program written in 1998 was assumed to never run on more than 2048 processors. In 1998, the largest number of processors available to the author was 128. Fifteen years later, when the program was run over 32000 processors by a user, the program failed. More recently, the author created a database and assumed that signed 32 bit integers would be sufficient to store all the data involved. Only later was it discovered that 2 billion records were too small for the problem that was being solved.

Conclusion: Besides the practice of "future-proofing" and the issue related to thinking exponentially, there are additional best practices and issues that the authors experienced while developing high-productivity tools (viz., generative programming techniques, disciplined agile delivery model, and cloud-enabling the tools for installation-free use). Such practices and issues will be discussed during the workshop.

References

1. Ritu Arora and Trung Nguyen Ba. 2017. ITALC: Interactive Tool for Application-Level Checkpointing. In Proceedings of the Fourth International Workshop on HPC User Support Tools (HUST'17). ACM, New York, NY, USA, Article 2, 11 pages. DOI: https://doi.org/10.1145/3152493.3152558
2. Ritu Arora, Julio Olaya, and Madhav Gupta. 2014. A Tool for Interactive Parallelization. In Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE '14). ACM, New York, NY, USA, Article 51, 8 pages. DOI: https://doi.org/10.1145/2616498.2616558
3. Kapil Agrawal, Mark R. Fahey, Robert McLay, and Doug James. 2014. User environment tracking and problem detection with XALT. In Proceedings of the First International Workshop on HPC User Support Tools (HUST '14). IEEE Press, Piscataway, NJ, USA, 32-40. DOI=http://dx.doi.org/10.1109/HUST.2014.6
4. Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall. ISBN 0-13-629049-3.