Abstract:

As applications are run on increasingly parallel and heterogeneous platforms, reproducibility of numerical results or code behaviors is becoming less and less obtainable. The same code can produce different results or occasional failures such as a crash on different hardware or even across different runs on the same hardware. This is due to many factors including non-determinism and the lack of numerical reproducibility with floating-point implementations. The target audience is application and tool developers interested in discussing case studies, tools, and techniques needed to address reproducibility on exascale systems. The format will be a panel of experts plus audience participation.

Moderator:

Miriam Leeser, Northeastern University

Panelists:

James Demmel, UC Berkeley
Ganesh Gopalakrishnan, Univ of Utah
Michael Heroux, Sandia
Walid Keyrouz, NIST
Kento Sato, LLNL

Report overview:

This report presents selected topics discussed at the SC15 BoF on Reproducibility of High Performance Codes and Simulations Tools, Techniques, Debugging by the panelists and the attendees on Thursday, November 19, 2015.

Slides of the Panelists can be found at: https://gcl.cis.udel.edu/sc15bof.php
Definitions of Reproducibility

Domain scientists need to be aware of the different “tiers” of reproducibility and select the right tiers based on their needs.

Reproducibility Schemes

The discussed reproducibility schemes are listed as follows (we will try to define the term “equivalent” later):

- Generating the “equivalent” program outputs.
- Generating the “equivalent” program performance.

The former reproducibility scheme is the focus on this BOF session. Term “reproducibility” in this article will refer to the first scheme listed above.

Tiers of Reproducibility

The tiers of reproducibility discussed in the BOF sessions are listed as follows:

- Bit-wise tier.
- Concurrency schedule determinism tier (e.g., thread and message matching schedule).
- Result quality tier: Reproducibility is claimed as long as the results satisfy the desired quality criteria. Image processing software is an example of this tier. Some machine learning algorithms also fall into this tier: for some classification algorithms, as long as the generated classifiers do accurate predictions, “subtle” reproducibility such as the bit-wise tier is not required. Another example belonging to this tier is that some geometry solvers only concern the residuals on the results (the results are acceptable as long as the residuals are small).
- Control-flow tier.
- Statistical tier.
- “Explainability” tier: Reproducibility is claimed as long as the sources of non-determinism are located (and accepted).

Sources of Non-reproducibility

Floating-point non-associativity is the major source of non-reproducibility. The associated non-reproducibility can be caused by:

- Non-deterministic concurrency scheduling.
- Compilation flags: Some compilers may not be aware of floating-point non-associativity and/or freely reassociate expressions based on ‘fast math’ flags. Others may choose a narrow range of rounding (e.g., ‘always to nearest’), not support gradual underflows, and truncate results when 80-bit internal values are stored into registers.
- Hardware Floating-point Arithmetic Unit (FAU) implementations: Different implementations of FAUs have different register width which may also affect floating-point reproducibility. Others may offer options such as Fused Multiply Add (FMA).

Defining Reproducibility by Answering Three Questions

1. What to compare between different versions (e.g., program output, thread interleaving)?
2. How do we define "deviation" (E.g., bit-wise or quality)?
3. Where do different program versions come from? Possible sources include:
   ○ Different versions of the same program
   ○ Same program, different compilers, and same platform
   ○ Same program, same compiler, and different platforms

**Scenarios of the Need for Reproducibility**

**Debugging**

Reproducibility is required for the purpose of debugging because without reproducibility, programmers may not be able to distinguish bugs from the non-reproducible program outputs.

Debugging MPI programs is an example of this problem. For example, floating-point arithmetic is not associative and MPI_Reduce can calculate the summation of a group of floating-point values with different associations in different runs. Such reduction schemes exist in many applications including sparse matrix solvers.

Porting applications among platforms is another example of the need for reproducibility and debuggability. For example, when porting a CPU program to a GPU, architectural differences, such as FAUs and register bit-width among processors, may result in non-determinism. Consequently, the results generated on a certain platform may not be reproducible on another platform. Without reproducibility, it may be difficult for the software developers to confirm the correctness of their application porting. An example of this was reported in [Meng et al. XSEDE’13].

Some branch conditionals are data-dependent. In this case, a subtle change of variables’ values (e.g., caused by lack of bit-wise reproducibility) may change the control flow and result in program errors such as segmentation faults. Without reproducibility, such error scenarios are hard to replay, and the root cause of the errors are difficult to resolve.

**Reproducibility Solutions**

**Record-and-replay**

Record-and-replay enforces floating-point associativities that helps programs generate the same result w.r.t. the same input. This technique can also help replaying control flows. However, record-and-replay requires large amount of storage for the execution logs. Compressing techniques are required to apply record-and-replay to large-scale programs [Sato et al. SC’15].

**Precise Computation**

Precise computation can be achieved by many approaches such as using high bit-width floating-point types, fixed-point numbers or using algorithms such as Kahan’s summation [Johnston et al. IEEE Cluster’15].
In this BoF session, an algorithm based reproducible computation library, ReproBlas\(^1\), was mentioned which allows programmers to use high reproducibility computation routines at the price of performance. Some preliminary results show that ReproBlas causes 3x slowdown on some 1-process computation and 1.2x slowdown on some 1024-process computation.

**Using Statistical Input Data**

Programmers should be educated to use statistical data for their programs’ inputs. E.g. taking averages of input values. Doing this can alleviate the effects caused by noisy inputs. This is especially important for applications whose inputs are acquired from imprecise sources such as sensors or image streams.

**Developing Reproducible Software**

Programmers should be aware of reproducibility if it is a concern for the software they develop. They should write reproducible code which is, for example, independent from specific thread schedules or floating-point associativity.

Compiler developers should provide options for generating reproducible code under the assumption that programmers have written reproducible programs. On the other hand, programmers should properly use the compiler flags in order to preserve reproducibility. A litmus test based method is currently developing in Prof. Gopalakrishnan’s group that helps programmers to properly apply compiler flags. [http://formalverification.cs.utah.edu/Repro/](http://formalverification.cs.utah.edu/Repro/)

In order to address the code sections causing non-reproducibility issues, a relative debugging method was suggested. To identify the problem causing code sections, the programmer should prepare a set of regression tests and different versions of the program should be generated during the development process. Then the programmer searches for the specific version which loses reproducibility compared to the previous version. The search can be done in a bisection manner. Once the specific version is identified, the updated code from the previous version should be suspected as the source of non-reproducibility.

**Performance-reproducibility Trade-offs**

Reproducibility is usually at the price of performance. This trade-off was discussed.

**Floating-point Bit-width Allocation**

Using low bit-width floating-point numbers unleashes performance but sacrifices reproducibility. Therefore, programmers can safely assign low bit-widths in the code sections which do not require reproducibility. Tools such as Precimonious [Rubio-González et al. SC’13] can automatically tune programs in assigning low bit-width floating point types.

**Determinism of Floating-point Associativity**

Multi-threaded programs may use non-deterministic reduction calls to improve performance. Using non-deterministic reductions inevitably causes non-reproducibility. Programmers should be aware of this issue.

---

\(^1\) ReproBlas site: [http://bebop.cs.berkeley.edu/reproblas/](http://bebop.cs.berkeley.edu/reproblas/)
Here is a research question: Is it possible to sacrifice non-determinism to a “small degree” and achieve “highly possible” reproducibility?

**Conclusions**

Programmers need to be educated on possible sources of non-reproducibility in their code and approaches that can help debug and identify the sources. Programmers also need to be able to identify where reproducibility is desired, and when performance can be sacrificed for reproducibility. Tools to help in this identification, as well as ways to help programmers identify causes and solutions to reproducibility are needed. More research into tools, techniques and solutions in general is needed.

**Acknowledgements**

Thank you to the students who contributed to this report: Ganesh Gopalkrishnan (University of Utah), Geof Sawaya (University of Utah), Janki Bhimani (Northeastern University), Dylan Chapp (University of Delaware), Simone Atzeni (University of Utah), Wei Fan Chiang (University of Utah).