CORVETTE: Program Correctness, Verification, and Testing for Exascale

Koushik Sen (**PI**) James Demmel, University of California at Berkeley
Costin Iancu, Lawrence Berkeley National Laboratory

http://crd.lbl.gov/organization/computer-and-data-sciences/future-technologies/projects/corvette/

The goal of this project is to provide tools to assess the correctness of parallel programs written using hybrid parallelism. There is a dire lack of both theoretical and engineering know-how in the area of finding bugs in hybrid or large scale parallel programs, which our research aims to change. As *intra-node* programming is likely to be the most disrupted by the transition to Exascale, we will emphasize support for a large spectrum of programming and execution models such as dynamic tasking, directive based programming, and data parallelism. For inter-node programming we plan to handle both one-sided (PGAS) and two-sided (MPI) communication paradigms.

We aim to provide tools that identify sources of non-determinism in program executions and make concurrency bugs (data races, atomicity violations, deadlocks) and floating-point behavior reproducible.

In order to increase the adoption of automatic program bug finding and exploration tools, novel techniques to increase **precision** and **scalability** are required. Precision implies that false alarms/positives are filtered and only the real problems are reported to users. During our research we will explore state-of-the-art methods that use dynamic program analysis. Since dynamic analysis monitors the program execution the resulting tools are precise, at the expense of scalability. Current approaches exhibit 10X-100X runtime overhead: it is our goal to provide precise tools with no more than 2X runtime overhead at large scale. We will also research techniques to maximize the tool efficacy on a time budget, e.g. no more than 10% overhead.

We will also research novel approaches to assist with program debugging using the minimal amount of concurrency needed to reproduce a bug and to support two-level debugging of high-level programming abstractions (DSLs). Furthermore, we plan to apply the combination of techniques developed for bug finding to provide an environment for exploratory programming. We will develop tools that allow developers to specify their intentions (not implementation) for code transformations and that are able to provide feedback about correctness. Besides code transformations, we plan to allow for automatic algorithmic tuning, i.e. transparently choosing at runtime the best implementation with respect to a metric from a collection of algorithms. As an initial case study, we will apply this methodology to determine program phases where double floating-point precision can be replaced by single precision.

1 Verification and Testing of Distributed Memory Parallel Programs

To achieve scalability and efficient utilization on future Peta and Exascale systems, high-performance scientific computing uses hybrid parallelism: the SPMD paradigm used for inter-node coordination is augmented with shared memory approaches for intra-node programming. Many studies showcase the advantages of MPI+X (X=OpenMP, Unified Parallel C), while others use Partitioned Global Address Space (PGAS) languages: UPC, Co-Array Fortran, Chapel and X10. In these approaches asynchrony is employed at the shared-memory node level as well as for inter-node coordination, resulting in a program where concurrency bugs are likely to occur and are hard to find.

We propose an active testing framework to find concurrency bugs in distributed memory programs with precision and scalability. Active testing works in two phases: in the first phase, it performs an imprecise dynamic analysis of an execution of the program and finds potential problems that could happen if the program is executed with a different thread schedule. In the second phase, active testing re-executes the program by actively controlling the thread schedule so that the behavior predicted in the first phase can be confirmed.

2 Testing and Debugging of Floating-Point Programs

Testing and debugging scientific applications is a difficult task, in particular for floating-point programs. Single and multi-threaded floating-point programs often suffer from a large variety of numerical exception problems (anomalies) that can have a significant impact on the results. Examples of common anomalies include: (1) rounding error accumulations, (2) conditional branches involving floating-point comparisons, (3) arithmetic underflow/overflow, and (4) benign and catastrophic cancellations.

Detecting such anomalies can be done using various techniques: altering rounding mode of floating-point arithmetic hardware and observing the sensitivity of the program to those changes, increasing the precision of the calculations on some floating-point operations (by using high precision or even arbitrary precision) and observing the impact on the final result, or also using interval arithmetic. These techniques vary in the costs of their application and in the kind of anomalies they detect. We propose to develop automatic techniques for testing and debugging anomalies in floating-point programs. Our techniques will help developers whose expertise does not necessarily extend to numerical error analysis by making the process of debugging both easier and faster, thus improving productivity.

This work consists of two parts: (1) *detecting* anomalies, and (2) *diagnosing* their cause. For detecting anomalies, we plan to explore existing techniques that combine concrete execution and symbolic evaluation to generate inputs that show a given anomaly. Applying these techniques will have its own challenges; they have not been used to generate floating-point input values.

For determining the cause of anomalies, we plan to adopt a Delta-Debugging like approach. The delta-debugging algorithm works as a binary search to isolate the failure-inducing part of a program and to determine a local minimal set of changes to be applied to the original program, so that the result remains within a given threshold of a known and more accurate result. We plan to use this algorithm to identify the parts of the program relevant for a given anomaly. This information will help developers in debugging the problem.