E Progress Report

In the first phase of the project we have demonstrated novel approaches in several areas:

i. Low overhead automated and precise detection of concurrency bugs at scale.
ii. Using low overhead bug detection tools to guide speculative program transformations for performance.
iii. Techniques to reduce the concurrency required to reproduce a bug using partial program restart/replay.
iv. Techniques to provide reproducible execution of floating point programs.
v. Techniques for tuning the floating point precision used in codes.

E.1 Detecting Concurrency Bugs

Our research in this area has resulted in the UPC-Thrille tool released with the Berkeley UPC compiler, available at `http://upc.lbl.gov`.

Efficient Data Race Detection for Distributed Memory Parallel Programs Chang-Seo Park, Koushik Sen, Paul Hargrove, Costin Iancu. SC 2012. *In this paper we present a precise data race detection technique for distributed memory parallel programs. Our technique, which we call Active Testing, builds on our previous work on race detection for shared memory Java and C programs and it handles programs written using shared memory approaches as well as bulk communication. 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 data races 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 data races reported in the first phase can be confirmed. A key highlight of our technique is that it can scalably handle distributed programs with bulk communication and single- and split-phase barriers. Another key feature of our technique is that it is precise—a data race confirmed by active testing is an actual data race present in the program; however, being a testing approach, our technique can miss actual data races. We implement the framework for the UPC programming language and demonstrate scalability up to a thousand cores for programs with both fine-grained and bulk (MPI style) communication. The tool confirms previously known bugs and uncovers several unknown ones. Our extensions capture constructs proposed in several modern programming languages for High Performance Computing, most notably non-blocking barriers and collectives.*

Scaling Data Race Detection for Partitioned Global Address Space Programs Chang-Seo Park, Koushik Sen, Costin Iancu. ICS 2013.

Contemporary and future programming languages for HPC promote hybrid parallelism and shared memory abstractions using a global address space. In this programming style, data races occur easily and are notoriously hard to find. Existing state-of-the-art data race detectors exhibit $10\times - 100\times$ *performance degradation and do not handle hybrid parallelism. In this paper we present the first complete implementation of data race detection at scale for UPC programs. Our implementation tracks local and global memory references in the program and it uses two techniques to reduce the overhead: 1) hierarchical function and instruction level sampling; and 2) exploiting the runtime persistence of aliasing and locality specific to Partitioned Global Address Space applications. The results indicate that both techniques are required in practice: well optimized instruction sampling introduces overheads as high as 6500% (*$65\times$ *slowdown), while each technique in separation is able to reduce it only to 1000% (*$10\times$ *slowdown). When applying the optimizations in conjunction our tool finds all previously known data races in our benchmark programs with at most 50% overhead when running on 2048 cores. Furthermore, while previous results illustrate the benefits of function level sampling, our experiences show that this technique does not work for scientific programs: instruction sampling or a hybrid approach is required.*

E.2 Speculative Transformations

Our research in this area has resulted in software available with NWChem.

Barrier Elision for Production Parallel Programs Milind Chabbi, Wim Lavrijsen, Wibe de Jong, Koushik Sen, John Mellor-Crummey, Costin Iancu. PPoPP 2015.

Large scientific code bases are often composed of several layers of runtime libraries, implemented in multiple programming languages. In such situation, programmers often choose conservative synchronization patterns leading to suboptimal performance. In this paper, we present context-sensitive dynamic optimizations that elide barriers redundant during the program execution. In our technique, we perform data race detection alongside the program to identify redundant barriers in their calling contexts; after an initial learning, we start eliding all future instances of barriers occurring in the same calling context. We present an automatic on-the-fly optimization and a multi-pass guided optimization. We apply our techniques to NWChem—a 6 million line computational chemistry code written in C/C++/Fortran that uses several runtime libraries such as Global Arrays, ComEx, DMAPP, and MPI. Our technique elides a surprisingly high fraction of barriers (as many as 63%) in production runs. This redundancy elimination translates to application speedups as high as 14% on 2048 cores. Our techniques also provided valuable insight about the application behavior, later used by NWChem developers. Overall, we demonstrate the value of holistic context-sensitive analyses that consider the domain science in conjunction with the associated runtime software stack.

E.3 Concurrency Reduction for Debugging

OCR: Partial Deterministic Record and Replay of One-Sided Communication Xuehai Qian, Paul Hargrove, Koushik Sen, Costin Iancu. In preparation.

Debugging large-scale distributed HPC applications is challenging. One-sided communication is widely used in Partitioned Global Address Space (PGAS) programming models. Despite its potential performance advantages, the inherent non-determinism makes debugging more difficult. The essential challenge is that the readers of updated shared data do not have any information on which remote threads produced the updates. This paper presents OPR (One-sided communication Partial Record and Replay (R&R)), a general deterministic R&R scheme that could partially replay one-sided communications. Using OPR, the user specifies a subset of threads (R_Set) to record, threads in R_Set could be partially replayed without executing threads not in R_Set. Due to the lack of producer information, data replay is used to ensure replay correctness. In record phase with all threads, OPR logs input values for read operations to shared addresses and generates a value log for each thread in R_Set. In replay, each thread in R_Set reproduces the same execution in isolation deterministically based on value log. To reduce value log size, in record phase, each thread maintains a shadow memory for the values that have seen by a thread. We only log the values for reads when they are either initial (first read) or has been changed. To infer the communication pattern, OPR runs a simplified vector clock algorithm during record and imprecisely generates an event order log. The even order information is used together with value log to match the producer and consumer in replay phase. Therefore, OPR makes it possible to debug part of a large execution on small machines.

E.4 Floating Point Reproducibility

Our research has resulted in the ReproBLAS released software.

Reproducible Tall-Skinny QR Factorization H.D. Nguyen and J. Demmel. 22st IEEE Symposium on Computer Arithmetic 2015.

Reproducibility is the ability to obtain bitwise identical results from different runs of the same program on the same input data, regardless of the available computing resources. Recently, techniques have been proposed to attain reproducibility for BLAS operations, all of which rely on reproducibly computing the floating-point summation and dot product. Nonetheless, a reproducible BLAS library does not automatically translate into a reproducible LAPACK library, especially when communications are taken into account. For instance, for the QR factorization, conventional algorithms such as Householder transformation or Gram-Schmidt process can be used to reproducibly factorize a floating-point matrix by fixing the order of computation, for example column-by-column from left to right, and by using reproducible versions of level-1 BLAS operations such as dot product and 2-norm. In a massively parallel environment, those algorithms have high communication cost due to the need for synchronization after each step. The Tall-Skinny QR algorithm obtains much better performance in massively parallel environments by reducing the number of messages to O(logP) where P is

the processor count, and reducing the number of reduction operations to O(1). Those reduction operations however are highly dependent on the network topology, in particular the number of computing nodes, and therefore are difficult to implement reproducibly and with reasonable performance. In this paper we present a new technique to reproducibly compute a QR factorization for a tall skinny matrix, which is based on the Cholesky QR algorithm to attain reproducibility as well as to improve communication cost, and the iterative refinement technique to guarantee the accuracy of the computed results. Our technique exhibits strong scalability in massively parallel environments, and at the same time can provide results of almost the same accuracy as the conventional Householder QR algorithm unless the matrix is extremely badly conditioned. Initial experimental results in Matlab show that for not too ill-conditioned matrices whose condition number is smaller than sqrt(1/e) where e is the machine epsilon, our technique runs less than 4 times slower than the built-in Matlab qr() function, and always computes numerically stable results in terms of column-wise relative error.

Reproducible Parallel Summation H.D. Nguyen and J. Demmel. IEEE Transactions on Computers, 2014.

Reproducibility, i.e. getting bitwise identical floating point results from multiple runs of the same program, is a property that many users depend on either for debugging or correctness checking in many codes. However, the combination of dynamic scheduling of parallel computing resources, and floating point nonassociativity, makes attaining reproducibility a challenge even for simple reduction operations like computing the sum of a vector of numbers in parallel. We propose a technique for floating point summation that is reproducible independent of the order of summation. Our technique uses Rumps algorithm for error-free vector transformation, and is much more efficient than using (possibly very) high precision arithmetic. Our algorithm reproducibly computes highly accurate results with an absolute error bound of n 228 macheps maxi jvij at a cost of 7n FLOPs and a small constant amount of extra memory usage. Higher accuracies are also possible by increasing the number of error-free transformations. As long as all operations are performed in to-nearest rounding mode, results computed by the proposed algorithms are reproducible for any run on any platform. In particular, our algorithm requires the minimum number of reductions, i.e. 1 reduction of an array of 6 double precision floating point numbers per sum, and hence is well suited for massively parallel environments.

Numerical Accuracy and Reproducibility at ExaScale H.D. Nguyen and J. Demmel. 21st IEEE Symposium on Computer Arithmetic, 2014.

Fast Reproducible Floating-Point Summation H.D. Nguyen and J. Demmel. 21st IEEE Symposium on Computer Arithmetic, 2014.

*Reproducibility, i.e. getting the bitwise identical floating point results from multiple runs of the same program, is a property that many users depend on either for debugging or correctness checking in many codes. However, the combination of dynamic scheduling of parallel computing resources, and floating point nonassociativity, make attaining reproducibility a challenge even for simple reduction operations like computing the sum of a vector of numbers in parallel. We propose a technique for floating point summation that is reproducible independent of the order of summation. Our technique uses Rumps algorithm for error-free vector transformation, and is much more efficient than using (possibly very) high precision arithmetic. Our algorithm trades off efficiency and accuracy: we reproducibly attain reasonably accurate results (with an absolute error bound c*n2*macheps*max(v[i]) for a small constant c with just 2n + O(1) floating-point operations, and quite accurate results (with an absolute error bound c*n3*macheps2*max(v[i])) with 5n + O(1) floating point operations, both with just two reduction operations. Higher accuracies are also possible by increasing the number of error-free transformations. As long as the same rounding mode is used, results computed by the proposed algorithms are reproducible for any run on any platform.*

E.5 Floating Point Precision Tuning

This research has resulted in the PRECIMONIOUS software publicly available.

Precimonius: Tuning Assistant for Floating-Point Precision Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, David Hough. SC13

Given the variety of numerical errors that can occur, floating-point programs are difficult to write, test and debug. One common practice employed by developers without an advanced background in numerical analysis is using the highest available precision. While more robust, this can affect program performance significantly. In this paper we describe a dynamic program analysis technique to find a lower floating-point precision that can be used in any part of a program. PRECIMONIOUS *performs a search on the program variables trying to lower their precision subject to accuracy constraints and performance goals. The tool then recommends a type instantiation for these variables using less precision while producing an accurate enough answer without causing exceptions. We evaluate* PRECIMONIOUS *on a few widely used functions from the GNU Scientific Library. For most of the programs tested,* PRECIMONIOUS *is able to reduce precision, which results in performance improvements as high as 25%.*

Floating Point Precision Tuning Using Blame Analysis Cindy Rubio-González, Cuong Nguyen, James Demmel, William Kahan, Koushik Sen, Wim Lavrijsen, Costin Iancu. In preparation.

While tremendously useful, automated techniques for tuning the precision of floating-point programs have to search over the program space and face scalability challenges. We present BLAME ANALYSIS*, a novel dynamic approach to improve precision tuning.* BLAME ANALYSIS *performs floating-point operations in a program with different levels of accuracy for the operands side-by-side with concrete execution. For each instruction,* BLAME ANALYSIS *determines the types of all operands in all other instructions required for a given precision in the target instruction. It then computes a solution based on a "merge" of all type assignments associated with any instruction. We implemented* BLAME ANALYSIS *in LLVM and evaluated it on ten scientific programs. When used standalone,* BLAME ANALYSIS *is successful in lowering the precision for all tests. The biggest benefits are observed when using* BLAME ANALYSIS *to filter the inputs to a search-based tool for floating-point tuning. Our experiments show that combining* BLAME ANALYSIS *with* PRECIMONIOUS *leads to finding better results faster. Combined analysis times are 9× faster on average, and up to 38× faster in comparison to* PRECIMONIOUS *alone. The solution computed is superior to the search-based tool and in three cases we observe as much as 40% program execution speedup.*