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 Automated Bug Finding

Summary: During the review period we have completed the first publicly available implementation of a data race detector for distributed memory programs that tracks all memory references. The goal of our implementation is to provide low overhead with good program coverage when running at scale. We propose two techniques to improve the scalability of data race detection in UPC programs: 1) hierarchical function and instruction level sampling; and 2) exploiting the runtime persistence of aliasing and locality in UPC applications. The results indicate that both techniques are required in practice and we analyze programs running on 2048 cores with less than 50% additional overhead.

				Overhead				
Bench	LoC	Time(s)	#Races	NL	HA.5	IA	FA0	I
guppie	271	19.070	2 + 0	54.9%	54.2%	53.7%	DNF	74.9%
psearch	803	0.697	3 + 2	2.48%	10.8%	666%	8.01%	6490%
BT 3.3	9698	189.48	7 + 3	0.574%	1.16%	77.6%	DNF	-
CG 2.4	1654	39.573	0 + 1	1.09%	27.6%	57.6%	DNF	2579%
EP 2.4	678	54.453	0	-0.618%	0.805%	2.09%	4.74%	111%
FT 2.4	2289	62.663	2 + 0	0.601%	30.1%	121%	DNF	2744%
IS 2.4	1R36	5.130	0	0.376%	119%	159%	DNF	1201%
LU 3.3	6348	155.997	0 + 44	-0.425%	-	75.7%	DNF	-
MG 2.4	2229	18.687	2 + 4	0.336%	176%	632%	DNF	2020%
SP 3.3	5740	247.937	10 + 3	0.160%	0.861%	29.1%	DNF	-

Table 1: *Statistics for the NAS Parallel Benchmarks class C, guppie and psearch running on 16 cores. We report the races found as A + B, where A represents the number of races detected by the original UPC-Thrille tool (column NL: No-Local) and B represents the additional number of races detected with our extensions. Some execution overheads are omitted (-), due to configuration errors.*

Attaining good performance and efficacy on contemporary and future large scale High Performance Computing systems requires using hybrid programming models: OpenMP+MPI, UPC+MPI, Intel TBB + MPI or OpenMP+UPC. With multiple levels, *intra-node* parallelism is usually exploited using shared memory programming models, while *inter-node* parallelism is exploited using message passing or shared memory abstractions. Bugs due to non-deterministic execution and conflicting memory accesses are fairly common and notoriously hard to detect in a parallelism rich environment. Previous work demonstrates the ability of dynamic program analyses to find concurrency bugs (data race, atomicity violations, deadlock) in shared memory programs. Dynamic program analyses have been also used to find *heisenbugs* in distributed memory programs: DAMPI [9] for MPI wildcard receives and UPC-Thrille [8] for data races in Unified Parallel C [6].

Data race detectors for shared memory programming trace individual memory references (load/store instructions) and reason about program semantics using a centralized analysis. The implementations are heavily optimized to reduce the instrumentation overhead and reportedly function with overhead lower than $10\times$. Bug finding for distributed memory programming models is made scalable by using a distributed analysis, but the current approaches illustrated by DAMPI and UPC-Thrille track only the calls into communication libraries. Thus, distributed memory tools need to be extended with tracking of memory references in order to handle hybrid programming. Furthermore, while acceptable when testing programs on workstations, the current overhead of dynamic program analyses is hard to stomach at the contemporary HPC concurrencies of tens of thousands of cores. Large scale analyses face the additional challenge to provide the lowest achievable overhead while still providing good coverage. While the adoption criteria for shared memory tools is "acceptable overhead", more stringent optimality criteria are desired at scale.

We have completed the first complete dynamic analysis for distributed memory programs able to track both memory references and communication calls. We extend the UPC-Thrille data race detection tool with tracking of individual memory references and discuss techniques to achieve low overhead for scientific applications running at scale. The results are validated for implementations of the NAS Parallel Benchmarks [5], as well as other fine-grained dynamic programming and tree search applications. We believe that our findings are widely applicable to any tool for data race detection in Partitioned Global Address Space languages: Chapel, Titanium, Co-Array Fortran, X10.

1.1 Scalable Data Race Detection

UPC-Thrille implements a dynamic program analysis running in two phases. In the first phase the program is executed with additional instrumentation and data about memory accesses, communication and synchronization operations is gathered and analyzed. For the purposes of this paper we distinguish three

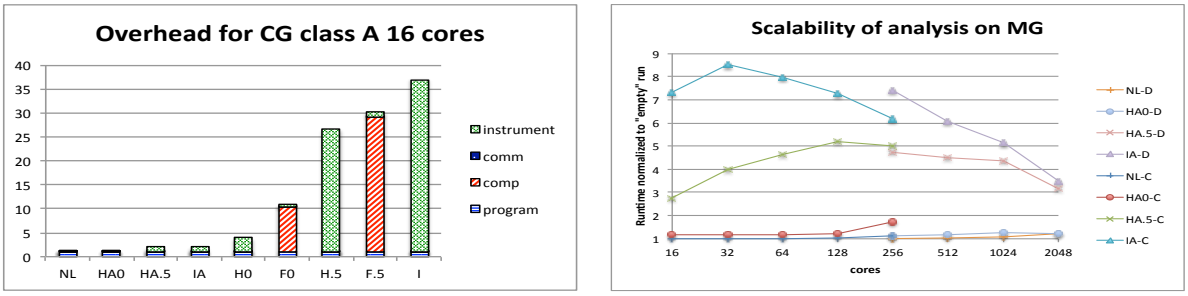

Figure 1: *Breakdown of data race detection overhead running on 16 cores (left). Scalability of the different sampling methods on NPB 2.4 MG, classes C and D (right).*

types of overhead: 1) *instrumentation overhead* is introduced by the checks to prune the non-interesting data accesses; 2) *computation overhead*, by the operations on internal data structures to manage the accesses and compute conflicting accesses; and 3) *communication overhead*, by the exchange of conflicting accesses between threads.

Analysis Overhead: The most widely used technique to reduce overhead is sampling of the program execution. Tools for shared memory use instruction level sampling while the distributed memory tools [8, 9] implement its equivalent by sampling the communication operations. For shared memory, Marino et al [7] recently introduced LiteRace which coarsens the granularity of the sampling at function boundaries: functions are compiled in two versions, instrumented and uninstrumented, each version being selected at runtime using heuristics. LiteRace showed better scalability and coverage than instruction level sampling when applied on several Microsoft programs, as well as Apache and Firefox.

We have experimented with both instruction level sampling and function level sampling on a Cray XE6 system composed of nodes containing two twelve-core AMD MagnyCours 2.1 GHz processors. The results in Table 1 indicate that instruction level sampling (IA) performs better than (FA) function level sampling for scientific programs. Instruction level sampling adds runtime overhead as high as $65\times$ while many runs using function level sampling did not terminate, even when instrumenting only the first execution of a function (FA0). This result contradicts the trends reported for LiteRace and it is caused by a combination of two factors: 1) determining the locality of a reference is expensive in PGAS programs; and 2) scientific programs have long running loops, with billions of memory accesses per invocation in our benchmarks. Our results also indicate that in most settings instrumentation overhead dominates the computation and communication overhead during the analysis. The typical behavior is illustrated in Figure 1 (left). Note that with function sampling (F.5, F0) the computation overhead increases due to the very large number of memory locations accessed in loops.

Reducing Overhead: For every memory reference there are two sources of runtime overhead. Instrumentation overhead is introduced to decide whether the reference should be recorded and computation overhead is introduced when recording the reference in the tool internal data structures. We employ a combination of techniques to improve the analysis performance: 1) we use program semantic information such as aliasing to prune un-interesting memory accesses; and 2) we use a hierarchical sampling approach where instrumentation is dynamically controlled both at the function level and at the instruction level.

The first optimization reduces the overhead of instrumentation by exploiting the insight that aliases are persistent in PGAS programs: once one is created it will point in the same memory region (private or global) for a long period of time. Using this we can eliminate the overhead introduced by looking up the

physical memory layout inside the language runtime. Adding the aliasing heuristics to any of the tool methods greatly improves performance. For example, the overhead of instruction sampling (I) is reduced from 3600% to 105% with (IA) for CG class A running on 16 cores. The overhead of hierarchical sampling (H) is reduced from 2550% with (H.5) to 99% with (HA.5) and from 294% with (H0) to 17% with (HA0). The lowest overhead of data race detection is obtained by the HA approach.

Function sampling ((F) or (FA)) is faster than instruction sampling ((I) or (IA), respectively) for problems using small datasets, such as class A of the NAS Parallel Benchmarks. When increasing the data set size to B, C and D, function sampling in any flavor does not terminate, while the highest overhead observed for instruction sampling is $65\times$. From all benchmarks considered, the only exception happens for *psearch* and EP where (F) is roughly twice as fast as (I). *psearch* is a tree search benchmark which performs a constant and small amount of work per function, independent of the problem size: this is a common characteristic to many commercial applications. EP is an "Embarrassingly Parallel" benchmark where no global memory accesses are made and thus none need to be tracked. The performance reversal observed for most benchmarks contradicts the common intuition that function sampling performs better than instruction sampling.

Hierarchical sampling (H) performs better than both instruction sampling (I) and function sampling (F) as it reduces all three type of overhead: instrumentation, computation and communication. With hierarchical sampling we observe slowdowns as high as $20\times$ which is still unacceptable when running at scale. Applying the aliasing heuristic reduces the overhead of data race detection for both instruction level and hierarchical sampling. The maximum slowdown observed by (IA) is $10\times$ while the maximum slowdown for (I) is $65\times$. Similar results are observed for (HA) when compared to (H).

1.2 Results

Figure 1 (right) shows the performance of our approach when performing strong scaling experiments for the classes C and D of the MG NAS Parallel Benchmark. For all experiments, the lowest overhead is introduced by the (HA) configuration and we are able to find all the races with less than 50% runtime overhead when running up to 2048 cores. In the case of the NAS Parallel Benchmarks class C on 16 cores, the weighted average overhead for all the benchmarks with (HA.5) was 11.9%.

For scalable data race detection, we needed to combine the two techniques: hierarchical sampling and aliasing heuristics. In the scalable versions of (IA) and (HA), the computation overhead is small. At large scale the communication overhead is also small due to the techniques presented in [8]. Overall, instrumentation overhead contributes the most to the slowdown caused by data race detection.

Races Found: In [8] we present a detailed discussion of the races found in the current program workload. Our extended implementation finds all these and, in addition, uncovers several other races. For a summary please see Table 1. For example, we detect a previously unknown race in NAS CG introduced by the presence of aliasing: memory is initialized using "local" pointers and distributed without synchronization to other threads using global pointers. In NAS BT, LU, and SP we uncover 50 additional races. Four of these races are real and confirmed by the tool; they occur when executing custom synchronization code similar to:

$$\texttt{signal(v = 1);} \quad \| \quad \texttt{wait(while(v == 0););}.$$

The remaining new data races are caused by data references separated by custom synchronization code. Identifying races in the presence of custom synchronization code is a common limitation of data race detection tools.

2 Automated Precision Tuning of Floating-Point Programs

Summary: During the review period we have started the development of an infrastructure to automatically adjust the floating point precision required during application execution. We have implemented an analysis pass in LLVM that given as input a floating point program it will propose modifications to reduce the precision of the program variables, i.e. from `double` to `float`. Preliminary experients applying the tool to the GNU Scientific Library (GSL) indicate up to 10% execution time improvements.

The use of floating-point applications has been growing rapidly over the past few years. Unfortunately, testing and debugging floating-point programs is a difficult task given the large variety of numerical errors that can occur in these programs, including extreme sensitivity to roundoff, incorrectly handled exceptions, and nonreproducibility across machines or even across runs on the same machine. Furthermore, most programmers are not experts in floating point, which makes testing and debugging these applications particularly challenging. One common practice in floating-point programs is to use the highest available precision. Using the highest available precision has its own disadvantages: it is more expensive in terms of running time, storage, and energy consumption.

We are developing automated testing and debugging techniques to recommend the lowest precision that can safely be used in each part of a program. We expect the non-expert developer to write code in the highest precision and to optionally specify accuracy requirements. Our tool then recommends a safe reduction in precision, automatically determining how little precision each part of the computation requires in order to produce an accurate enough answer without exceptions. At a high level, our approach consists of creating multiple variants of the program, each using different precision. We use the delta-debugging algorithm [4] to find a variant that uses less precision and complies with the developer's accuracy requirements.

2.1 Framework Components

We have designed and implemented a tool that automatically tunes the precision of a program given an input set. Our tool consists of four different components (see square boxes in Figure 2):

Creating Search File The first component creates the *search file* for the program whose precision is to be tuned. The search file contains a listing of the floating-point variables (and operators) in the program along with the set of floating-point types to be explored (e.g., `float`, `double`, and `long double`). The input to this component is the program under analysis. If the program contains any functions whose precision should not be modified, then the name of those functions can also be provided as input. Our framework is built using the LLVM compiler infrastructure [3], thus our tool requires the program to be compiled into LLVM bitcode form in order to apply our analysis.

Delta Debugging Algorithm The second component uses the delta debugging approach [4] to systematically search the space of possible program changes and produce a configuration set of changes to apply. Program changes include: (1) changing the precision of floating-point program variables, (2) changing the precision in which floating-point arithmetic operations are performed, and (3) choosing between more/less precise implementations of a given function. This component produces a *configuration file* that gives a type assignment for all floating-point variables and operators (if any) in the program.

Program Transformations We have identified a set of program transformations that need to be performed when changing the type of variables and operators in a program at the LLVM bitcode level. In order to apply these transformations, we require the LLVM bitcode for the program under analysis and a configuration file produced by the delta debugging algorithm. A modified bitcode file is produced, which reflects the type assignment given in the configuration file. For example, let variables `x` and `y` be of type `long double` in an original program *P*. A type configuration file could assign type `double` to variable `x` and type `float` to variable `y`. In this case, a program *P'* is produced where variables `x` and `y` have the types `double` and `float`, respectively.

Determining Type Assignment Validity To determine whether a type configuration is a valid configuration, we run the modified program and compare the result against the expected result. The expected result is the value (or values) obtained by running the original program in a given input. If a threshold is provided by the programmer, then it is taken into account when comparing the results. The outcome of the comparison is provided as feedback to the delta debugging algorithm, which will decide whether a new type configuration should be produced. If so, the process repeats. Eventually, the delta debugging algorithm will choose a valid type configuration. We accurately log and compare floating-point values.

2.2 Preliminary Results

We have completed an end-to-end implementation of the tool. The tool has been implemented in C, C++ and Python. We use LLVM 3.0 to implement our program transformations.

We initially applied the analysis to a set of small programs (50-500 LOC) in which experts had observed that a less precise program would still produce the expected results while improving performance significantly. These programs were originally written in C, Fortran and Matlab. We manually translated Fortran and Matlab programs into C. For each of these programs, experts provided the *ideal* type configuration, which was found manually. Figure 3 shows one of these programs, which calculates the arc length of a function. The program is originally written in the highest available precision (`long double`). Figure 4 shows the less precise program suggested by our tool. Note that this configuration matched the ideal configuration provided by experts. Our tool took only 4 seconds to find this type configuration automatically, which lead to a 10% speedup with respect to the original program. The experiments were run on a 2.9 GHz Intel processor machine with 16 GB RAM.

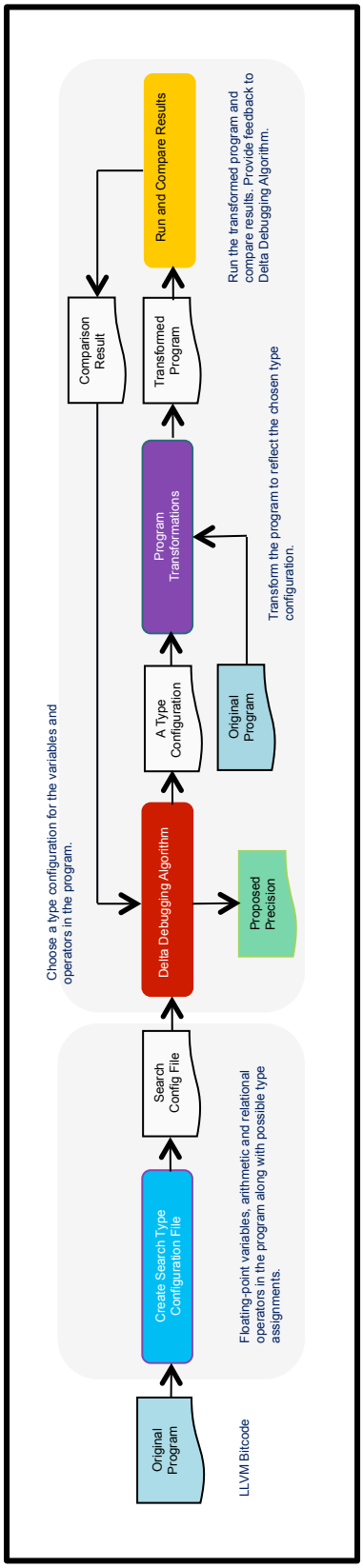

Figure 2: High-Level Framework Components

```
#include <math.h>
#include <stdio.h>

long double fun( long double x ) {
  int k, n = 5;
  long double t1, d1 = 1.0L;


  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}


int main( int argc, char **argv) {
  int i, j, k, n = 1000000;
  long double h, t1, t2, dppi, ans = 5.795776322412856L;
  long double s1, threshold = 1e-14L;



  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }

  // final answer is stored in variable s1
  return 0;
}
```

Figure 3: Original Program

```
#include <math.h>
#include <stdio.h>

double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f; // double before

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}


int main( int argc, char **argv) {
  int i, j, k, n = 1000000;
  double h, t1, t2, dppi;
  float ans = 5.795776322412856f; //double before
  long double s1;
  float threshold = 1e-14f; // long double before

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }

  // final answer is stored in variable s1
  return 0;
}
```

Figure 4: Precision-Tuned Program

Table 2: Number of Load, Store, and Arithmetic instructions executed and speedup for GSL programs

	Loads			Stores			Arith Ops			
GSL Program	F	D	LD	F	D	LD	F	D	LD	Speedup %
intro original	0	560	13	0	218	7	0	359	1	-
intro new	63	497	11	54	164	7	1	358	1	5.34 (1.05x)
cdf original	0	275	353	0	129	333	0	152	4	-
cdf new	83	192	353	30	99	333	6	146	4	84.49 (6.05x)
roots original	0	721	35	0	352	30	0	190	1	-
roots new	122	599	35	62	290	30	19	171	1	8.47 (1.09x)

We have recently compiled the entire GNU Scientific Library (GSL) [2] into a single LLVM bitcode file. Table 2 shows preliminary results for three of the test programs included with the library (intro.c, cdf.c and roots.c). Running our analysis on these programs shows speedup from 5.34% to 84.49% when tuning the precision of the library. Our goal is to complete a detailed study that includes more GSL test programs, and also to apply our analysis to real-world clients of the GSL library to assess the impact of precision tuning in larger and more complex floating-point programs. After we finish these experiments, we are planning to compile CLAPACK [1] into LLVM bitcode to apply our precision-tuning analysis to its extensive regression test suite. We also plan to apply our analysis to the Gyrokinetic Toroidal Code (GTC) code base from the Lawrence Berkeley National Laboratory.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[2] GSL Project Contributors. GSL - GNU scientific library - GNU project - free software foundation (FSF). http://www.gnu.org/software/gsl/, 2010.

[3] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[5] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. *Technical Report NAS-95-010, NASA Ames Research Center*, 1995.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, and K. W. E. Brooks. Introduction to UPC and Language Specification, 1999.

[7] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *PLDI*, 2009.

[8] C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient Data Race Detection for Distributed Memory Parallel Programs. In *Supercomputing (SC11)*, 2011.

[9] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Supercomputing (SC10)*, 2010.