Dynamic Analysis for Program Correctness and Optimizations: Minimizing Synchronization and Floating Point Precision

Koushik Sen, UC Berkeley & LBNL
James Demmel, UC Berkeley & LBNL
Costin Iancu, LBNL

How we build large programs today

- **Scientists depend on modularity**
  - Many independently written pieces glued together
  - Multi-language (Fortran, C++, UPC, …)
  - Multi-communication layers (OpenMP, MPI, GASnet, …)
  - Multi-library (BoxLib, CombBLAS, ScaLAPACK, …)
  - Heterogeneous hardware (CPU, GPU, …)
- **Advantages**
  - Easier to understand and build big apps from smaller components
  - Code reuse is efficient

How we build large programs today

- **Scientists depend on modularity**
  - Many independently written pieces glued together
  - Multi-language (Fortran, C++, UPC, …)
  - Multi-communication layers (OpenMP, MPI, GASnet, …)
  - Multi-library (BoxLib, CombBLAS, ScaLAPACK, …)
  - Heterogeneous hardware (CPU, GPU, …)
- **Disadvantages**
  - **Inaccuracies** from fast but careless data sharing ("race conditions")
    - Different answers from run to run ("nonreproduciblity")
  - **Inefficiencies** from overly conservative data sharing (too many "barriers")
  - **Inaccuracies** from using too little floating point precision
  - **Inefficiencies** from using high precision floating point everywhere
- **Goal – provide tools to address these problems**
- Common approach: "dynamic analysis"
  - Need to run program, gather data, learn from it, modify program (repeat)
  - Compilers ("static analysis") not enough

Practical Examples (so far ...)

- **Finding bugs**
  - Scalable data race detector for UPC found unknown bugs in established codes *(SC'11, ICS'13)*
  - Scaling replay of one-sided programs *(ICS'16)*
- **Performance Optimizations for NWChem**
  - Speculative elision of barriers *(PPoPP'15),* incorporated into NWChem
  - Energy optimizations *(Lavrijsen et al 2015)*
  - Converting blocking reads/writes to non-blocking ones (Saillard 2016)
- **Floating point reproducibility**
  - ReproBLAS (*Demmel et al, IEEE Trans Comp 2015*)
- **Floating point precision tuning**
  - Automatically lowered precision while maintaining final accuracy in GSL, NAS *(SC'13, ICSE'16)*
- **Open problems ...**

Part I: Race Detection and Optimization by Safely Removing Synchronizations

UPC Data Race Detection

Program monitoring and code generation infrastructure with low runtime overhead

Data race = two tasks access same memory location concurrently, one access is a write

1. Scaling Data Race Detection for Partitioned Global Address Space Programs. Park et al. ICS 2013
2. Efficient Data Race Detection for Distributed Memory Parallel Programs. Park et al. SC 2011

Design Requirements

- Efficiency and low overhead
  - Current commercial tools 10X-1000X on 16 cores
- Complete memory coverage: track every operation with high probability
- Precise: report only the “bugs” (no false alarms)
- Reproducible: identical behavior across executions
- Scalable in program size (LoCs), input size, concurrency
- Automated and guided detectors

UPC-Thrille: Data Race Detection Implementation

- For each load/store or communication operation

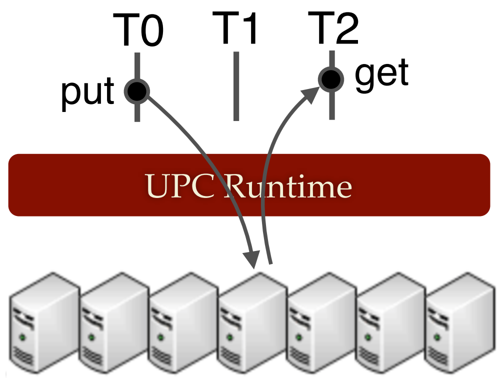
  Examine the address –> instrumentation overhead

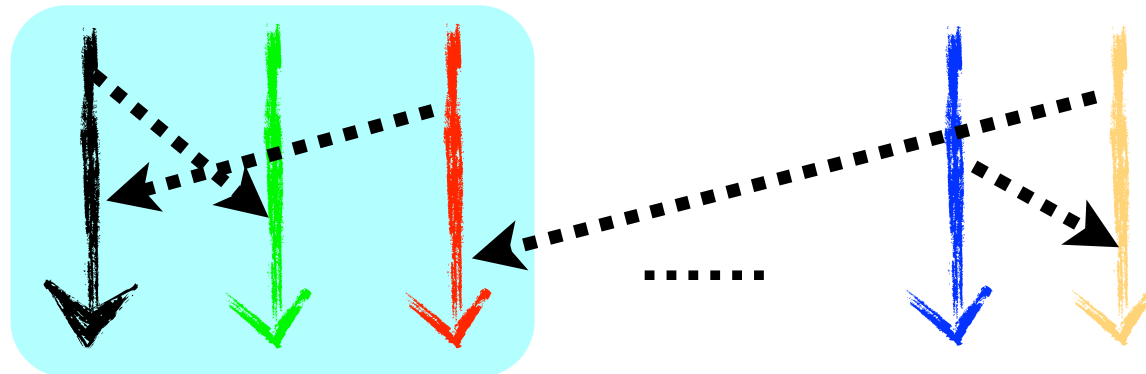
  Record the address –> data management overhead
- For each synchronization operation
  - Exchange information about all L/S and comms
  - Analyze for conflicts
- Instrumentation overhead is reduced by
  - Hybrid Sampling (instruction + function level sampling)
  - Further pruning using program analysis
- Data management overhead is reduced with better data structures
- **Our system: < 50% slowdown up to 2K cores on 20 applications**
  - Precise, more complete, scalable, reproducible with high probability

SReplay: reproduce nondeterministic bugs using a small set of threads

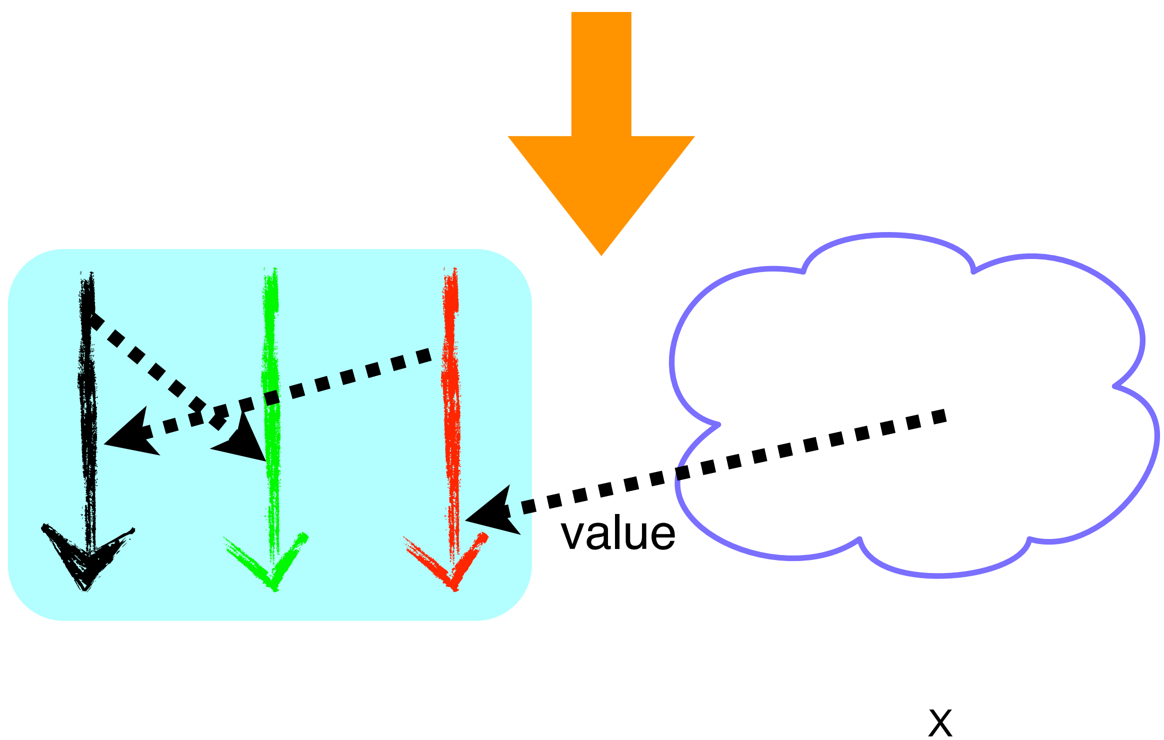
1. OPR: Deterministic Group Replay for One-Sided Communication, Qian et al., PPoPP 2016 (Poster)
2. SReplay: Deterministic Sub-Group Replay for One-Sided Communication, Qian et al., ICS 2016

Motivation



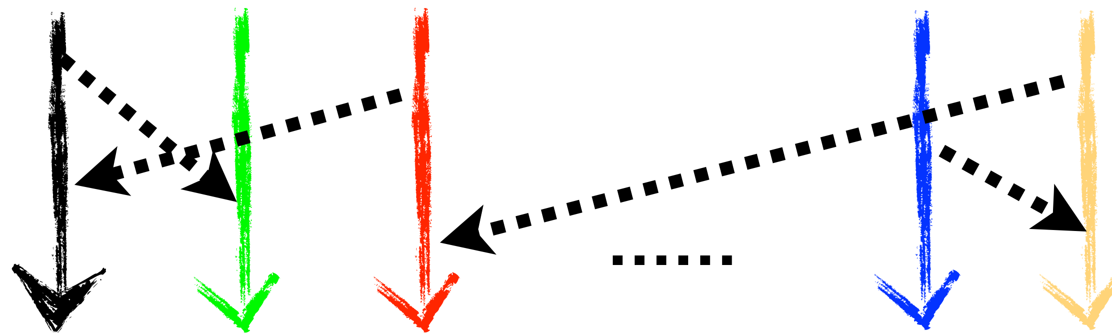


- Assumption: concurrency bugs typically exist among small set of threads

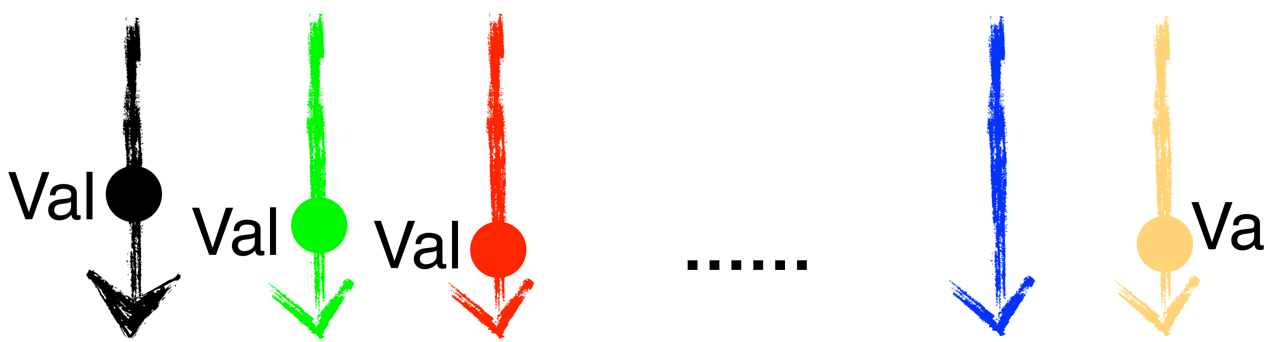


Two R&R Approaches

- **Order replay**
  - Log event order
  - Small log size
  - Same thread set in record and replay

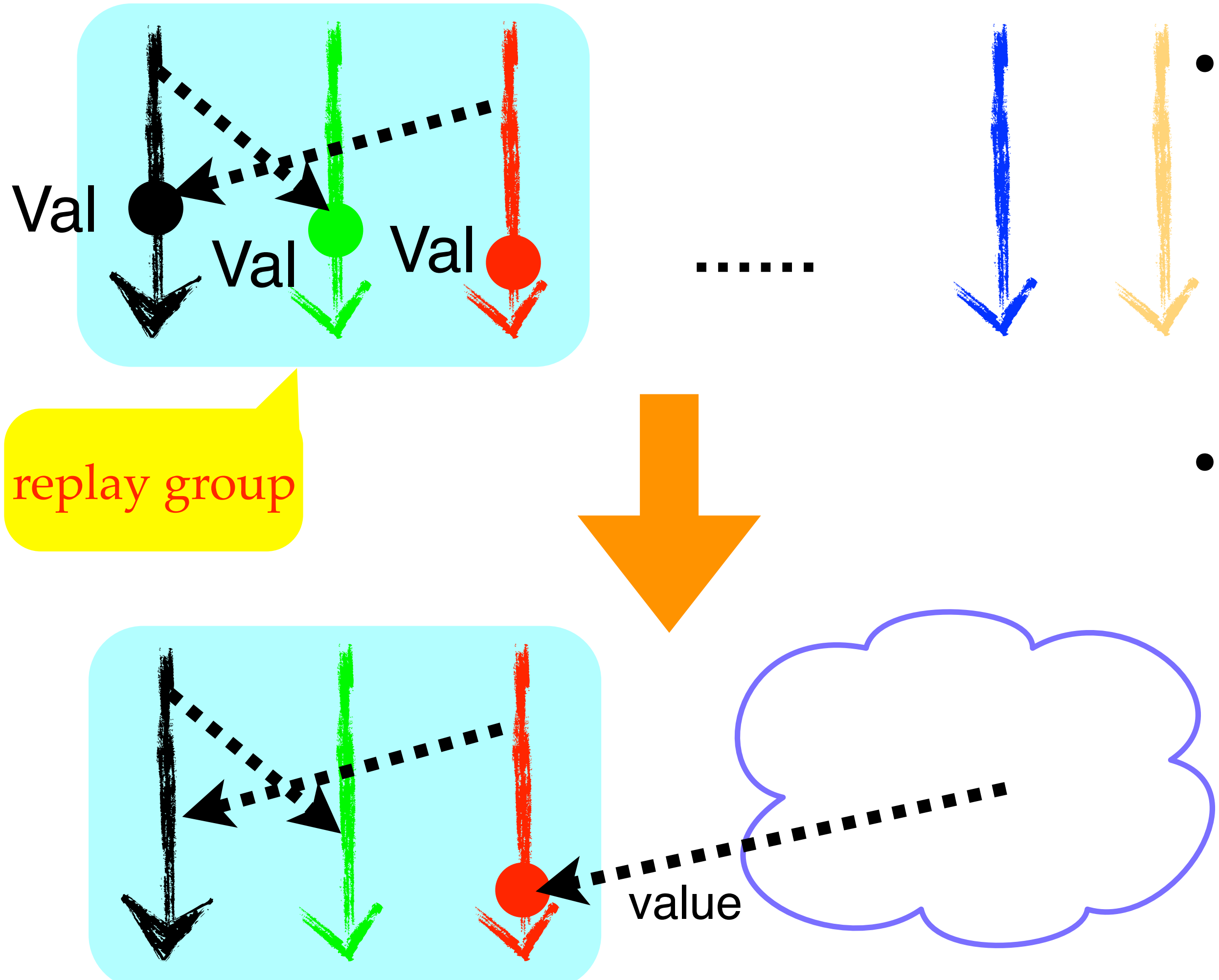


- **Data replay**
  - Record data input at the right time
  - Inject the values at same points in replay
  - Each thread could replay in isolation



X

SReplay: A hybrid approach



- **Record: in replay group**
  - Value inputs
  - Event order
- **Replay**
  - Data replay
  - Enforce event order
  - Infer communication

X

Result Highlights

- Used 15 UPC benchmarks to evaluate SReplay.
  - 8 NAS Parallel Benchmarks: BT, CG, EP, FT, IS, LU, MG, SP
  - 3 applications in the UPC test suite: guppie, laplace, cop
  - 2 applications in the UPC Task Library: fib, queens
  - Unbalance Tree Search (UTS)
  - Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous).
- 1.09x ~ 27.5x record overhead when recording 2 threads in 1024 threads
- Overhead does not increase significantly with the replay group size
- Low false positive/negative rates in event order detection

Safely Removing Barriers in NWChem

Using Data Race Detection for Performance Optimizations

1. Barrier Elision for Production Parallel Programs. Chabbi, Mellor-Crummey et al. PPoPP 2015

Barriers in NWChem

Example Runtime Overhead: 15% in QM-CC, 22% in QM-DFT

symmetry/sym_sym.F

```
next = nxtask(-nproc, 1)

NO GA OPS

call ga_copy(g_b, g_a)

call ga_destroy(g_b)

call ga_sync()
```

Do we need 9 barriers across 3 modules

(ga_,comex_, pnga_)

???

Programmer puts another Barrier()

Performance Improvements

Cores	Time (s)	Context	Total/skipped	Speedup
DCO-512	731	7959	138072/ 42%	**0.3% (0.7%)**
DCO-1024	1084	7959	138072/42%	**0.2% (7.6%)**
DCO-2048	1362	7959	138072/42%	**13.3%(13.9%)**
OCT-512	570	4702	72188/ 45%	**1.7% (3.4%)**
OCT-1024	586	4702	72188 / 45%	**4.4% (6.6%)**
OCT-2048	624	4702	72188/45%	**6.5% (6.0%)**

- Low overhead of instrumentation < 1% (most times)
- Offline analysis deletes 63% barriers, even higher speedup
- Feedback from analysis incorporated in NWChem (delete clearly redundant barriers)

Part II: Precision Tuning and Reproducibility of Floating-Point Programs

Precimonious: Automatic Tuning of Floating Point Precision

PRECIMONIOUS is open source, and can be found at
https://github.com/corvette-berkeley/precimonious

1. Precimonius: Tuning Assistant for Floating-Point Precision. Rubio-Gonzalez et al., SC'16
2. Floating-Point Precision Tuning Using Blame Analysis. Rubio-Gonzalez et al., ICSE'16

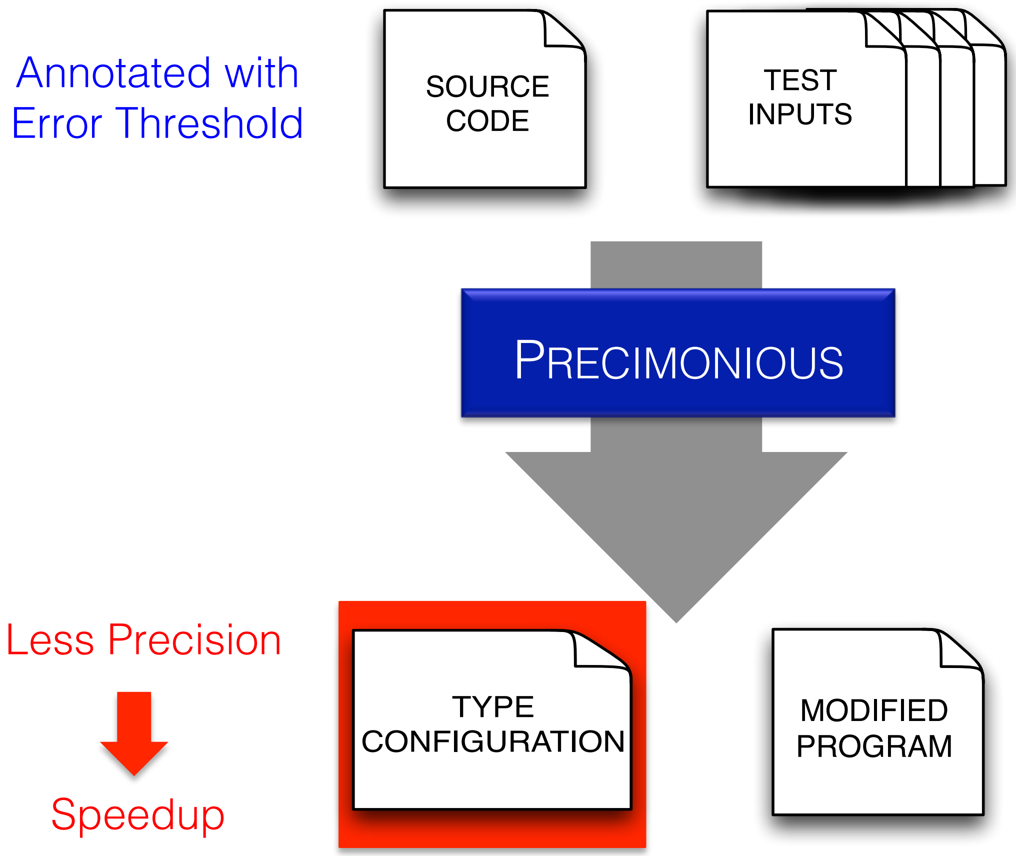
Motivation and Approach

- Conservative approach: use double precision everywhere
  - Pro: Usually most reliable, least effort required
  - Con: Uses more time, memory, energy than may be necessary
- Goal: Use as little precision as needed for any operation or variable, to get an "acceptable" final answer
  - Pro: Save time, memory, energy
  - Con: If done by hand, may require extensive numerical analysis, code rewriting, etc. (or if not done carefully, unexpected loss of accuracy)
- Automate analysis, using **Precimonious**
  - Short for "Parsimonious with Precision"
  - Uses "delta debugging", or bisection on code

PRECIMONIOUS [SC'13]

"Parsimonious with Precision"

Dynamic Program Analysis for Floating-Point Precision Tuning

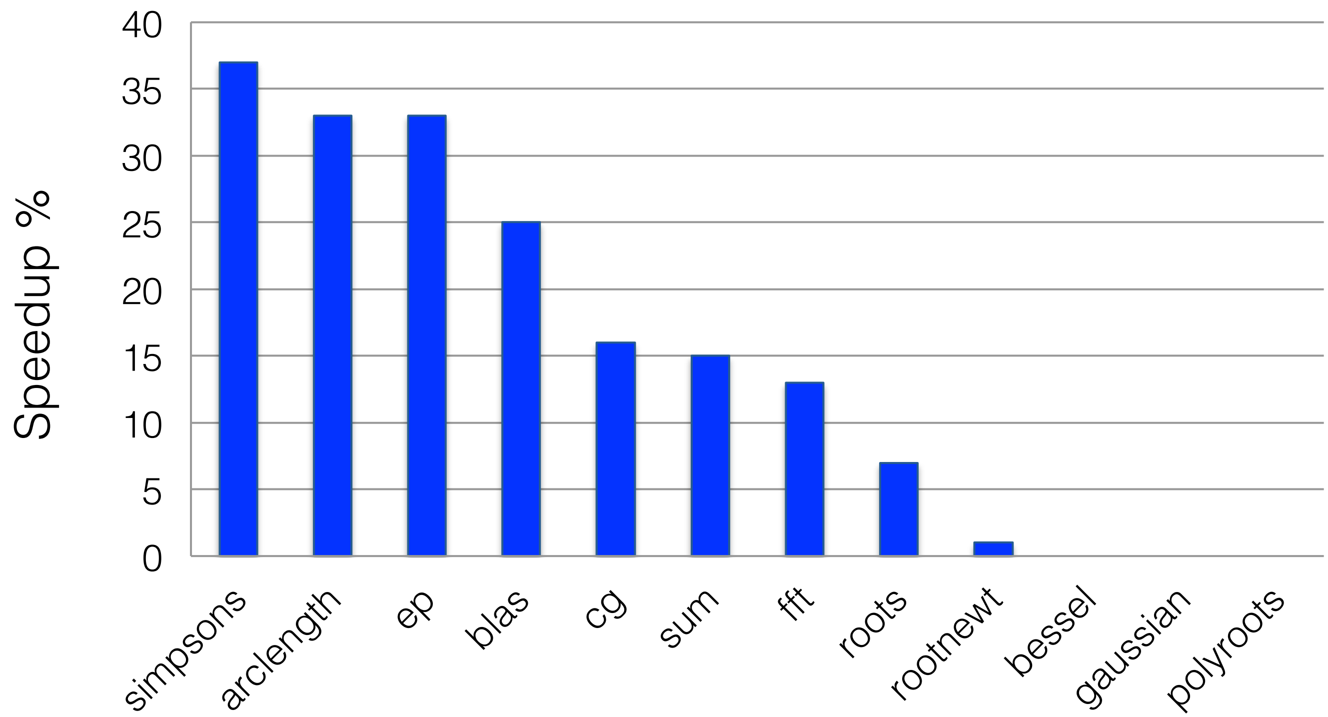


Dynamic analysis:
- Run program
- Gather data
- Learn from it
- Modify program
- Repeat

Experimental Results

	Original Type Configuration					Proposed Type Configuration (Error threshold: 10^{-4})					
	Program	L	D	F	Calls	L	D	F	Calls	# Config	mm:ss
GSL	bessel	0	18	0	0	0	18	0	0	130	37:11
GSL	gaussian	0	52	0	0	0	52	0	0	201	16:12
GSL	roots	0	19	0	0	0	0	19	0	3	1:03
GSL	polyroots	0	28	0	0	0	28	0	0	336	43:17
GSL	rootnewt	0	12	0	0	0	4	8	0	61	16:56
GSL	sum	0	31	0	0	0	9	22	0	325	28:14
GSL	fft	0	22	0	0	0	0	22	0	3	1:16
GSL	blas	0	17	0	0	0	0	17	0	3	1:06
NAS	EP	0	13	0	4	0	5	8	4	111	23:53
NAS	CG	0	32	0	3	0	2	30	3	44	0:57
	arclength	9	0	0	3	0	2	7	3	33	0:40
	simpsons	9	0	0	2	0	0	9	2	4	0:07

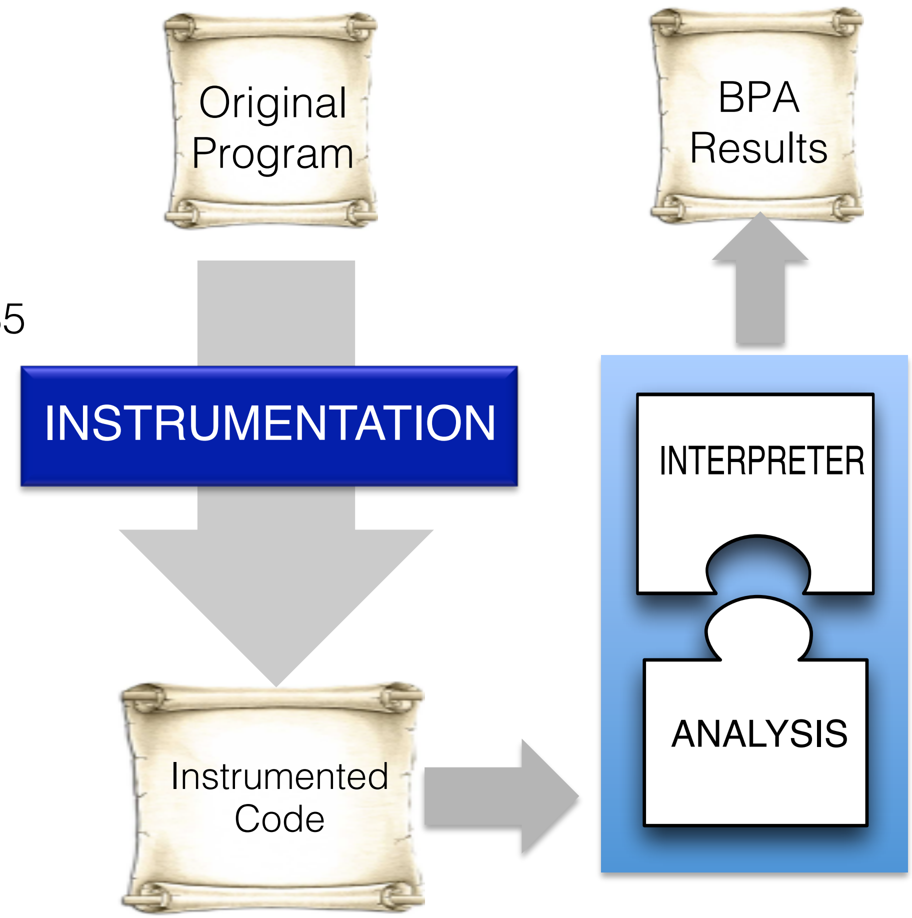
Speedup for Error Threshold 10^{-4}



Maximum speedup observed across all error thresholds: 41.7%

White-Box Approach: Blame Precision Analysis using Shadow Execution [ICSE'16]

- Scalability limitation of Precimonious
  - Too many runs for large programs
  - E.g., 52 FP variables → 1,435 configurations
- Goal: apply white-box approach to over-approximate set of variables that require higher precision (to be "blamed")



Results: Blame Precision Analysis

- Shadow execution performs FP operations in higher precision
  - Shadow object associated with each variable in the program
  - Shadow information includes value in different precisions
- Perform Blame Precision Analysis (BPA)
  - Collect a dynamic trace with shadow information
  - Construct a blame tree
    - Variables and operators that require higher precision given a precision requirement on the result
- Implemented using our general shadow execution framework for LLVM IR
- Combination of Blame Analysis with Precimonious
  - the optimized programs execute faster (in three cases, we observe as high as 39.9% program speedup) and
  - the combined analysis time is 9× faster on average, and
  - up to 38× faster than Precimonious alone.

Reproducible
Floating Point Computation

1. Jim Demmel, Hong Diep Nguyen, Peter Ahrens

Motivation (1/2)

- Since roundoff makes floating point addition nonassociative, different orders of summation often give different answers
- On a parallel machine, the order of summation can vary from run to run, or even subroutine-call to subroutine-call, depending on scheduling of available resources, so answers can change
- Why is this important?

Motivation (2/2)

- NA-Digest: Commercial FEM SW vendor wanted a parallel reproducible sparse linear equation solver, because his customers (civil engineers) had contractual obligations to their customers to get the same answer from run to run: “Will the bridge fall down or not?”
- Responses from ~100 UC Berkeley faculty to email query about the importance of reproducibility:
  - Most common: How will I debug without reproducibility?
  - How do I do fracture mechanics, where I do many random simulations looking for a very rare event, and when one occurs, I need to resimulate it exactly, while computing some side information?
  - What if my “illegal underground nuclear test detector” (funded by the United Nations) says “They did it!” and then “They didn’t do it”?
- Many workshops at recent Supercomputing conferences
  - Users, researchers, vendors (gcl.cis.udel.edu/sc15bof.php)
- Intel MKL with CNR (Conditional Numerical Reproducibility)
  - If user promises to use same number of cores, multicore code will perform operations in same order (with performance hit)
- Reproducible summation used in CCSM climate model (Pat Worley) for verification during development and reproducibility during production

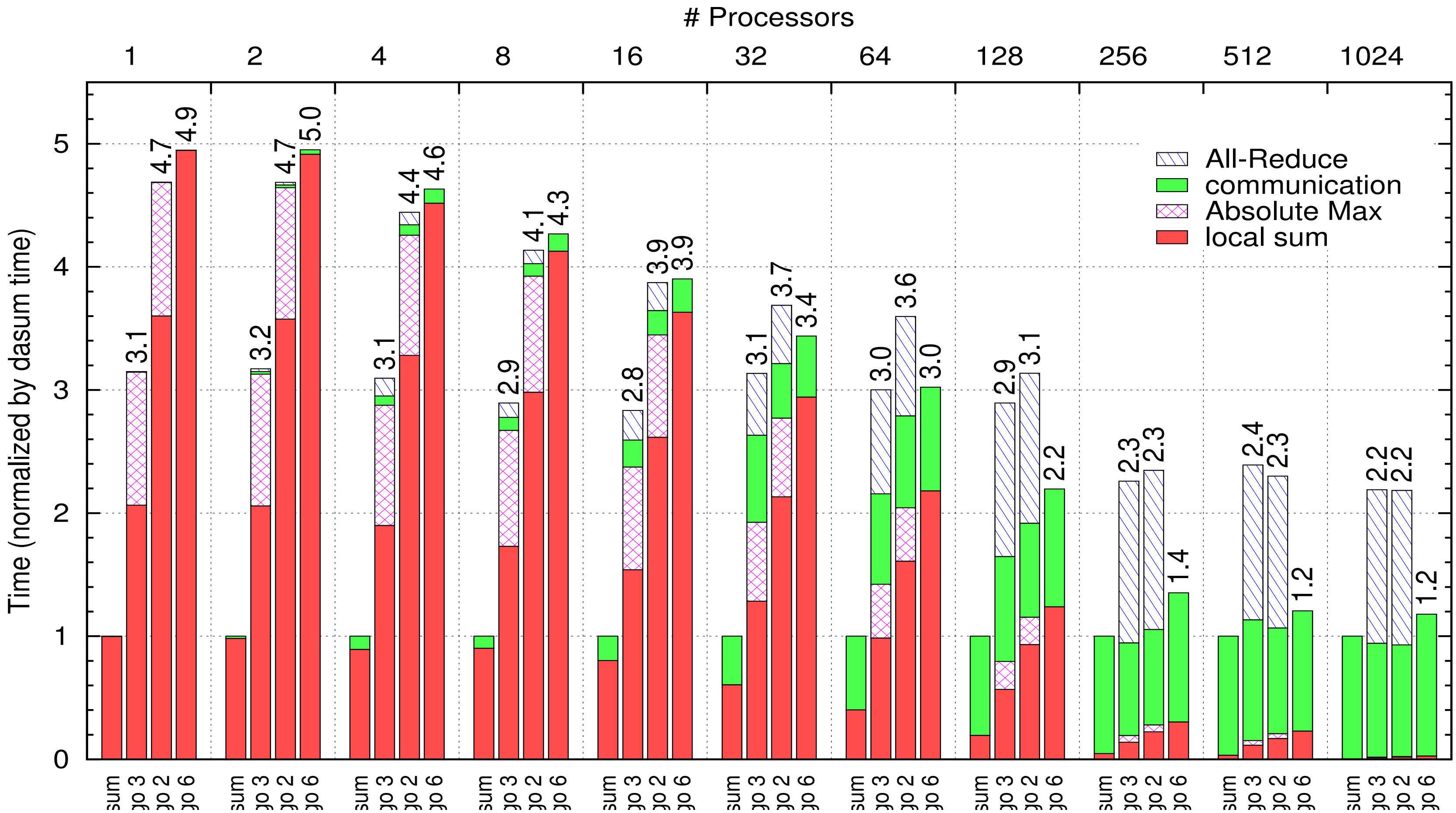
Reproducible BLAS

- First step in longer term goal of making Sca/LAPACK, other libraries/frameworks reproducible and still high performance
- Simplest algorithm for reproducible sum $s = \Sigma_i\ x(i)$
  1. Compute $M = \max_i |x(i)|$; exact and so reproducible
  2. Round all x(i) to 1 ulp (unit in last place) of M; error introduced no worse than usual error bound
  3. Add rounded x(i); they behave like fixed point numbers

     so summation exact and so reproducible
- Drawback: costs 2 or 3 passes over data sequentially, or 3 reduction/broadcast steps in parallel
- Better: can do it in 1 pass, or 1 reduction, by interleaving all 3 steps
- Industrial interest in hardware support

Performance results on 1024 proc Cray XC30

- **1.2x to 3.2x slowdown vs fastest (nonreproducible) code dasum**
- **Data for n=1M summands on up to p=1024 processors**
- **3 reproducible sum algorithms compared, best one depends on n, p**
- **Code and papers at bebop.cs.berkeley.edu/reproblas**



Future Work

- Correctness analysis and performance optimizations using static analysis has its limitations
- Dynamic program analysis is a must
  - Instrument, run, gather data, analyze and report, learn from data, modify code on the fly (JIT???)
  - In this project, we ended up implementing similar instrumentation framework over and over again
  - Tedious and error-prone
- We want to build a general, declarative, easy-to-use instrumentation, dynamic analysis, and code modification framework for an IR (e.g. LLVM)
  - possibly use it to support DSLs

Future Work

- Customizable Instrumentation and Code Transformation Framework (for LLVM and other IRs)
  - to detect and fix non-determinism and performance problems
- Configurable Instrumentation and Log Collection Framework at IR Level
  - Turn on and off instrumentation automatically or manually
  - Target LLVM
- Analyze collected logs
  - Infer bugs
  - Infer performance problems
- Recommend Dynamic Code Transformation at LLVM level
  - suggest addition or removal of instrumentation
  - suggest fix for non-deterministic bugs
  - suggest removal of synchronization and high-precision
  - suggest code transformation to replace blocking operations with non-blocking operations
- Apply fixes
  - after approval from user

Instrumentation and Code Transformation (for LLVM)

- STrex (coming soon ...)
  - an extension of regular expressions
  - works on tree-structured data (e.g. ASTs)
  - piggy-back on existing parser
  - little programming
- Conveniently specify
  - instrumentation rules (to collect data)
  - code rewriting rules (to fix bugs and performance problems)
    - JIT
  - query patterns over ASTs (to find common bugs)
  - transformation rules (DSL compiler)
- Implementation for C++ and C and other languages
- Work with application developers
  - and widely-used HPC applications (e.g. NWChem)

Future Work

- Complete implementation of sequential ReproBLAS
- Extend to PBLAS, other platforms, with autotuning
- Go up stack: Reproducible LAPACK, ScaLAPACK, many other libraries (depends on user demand)
- Collaborate with vendors on software and hardware implementations
- Under discussion with IEEE 754 Floating Point Standard and BLAS Standard Committees
- Use race detection tool to identify other sources of nonreproducibility, automate fixing them