Scaling Data Race Detection for Partitioned Global Address Space Programs

Chang-Seo Park Koushik Sen
University of California Berkeley
{parkcs,ksen}@eecs.berkeley.edu

Costin Iancu
Lawrence Berkeley National Laboratory
cciancu@lbl.gov

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.2.5 [*Testing and Debugging*]

1. Introduction

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 [5] for MPI wildcard receives and UPC-Thrille [4] for data races in Unified Parallel C [2].

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 present 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 [1], 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.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
ACM 978-1-4503-1922/13/02.

				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	-

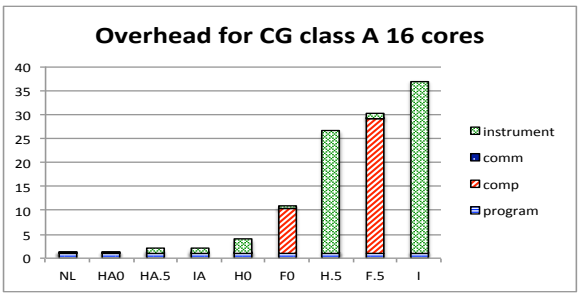
Table 1: *Statistics for the NAS Parallel Benchmarks class C, guppie and psearch running on 16 cores. We report the races found as A + B, where A represents the number of races detected by the original UPC-Thrille tool (column NL: No-Local) and B represents the additional number of races detected with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

2. 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 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 [4, 5] implement its equivalent by sampling the communication operations. For shared memory, Marino et al [3] 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 func-



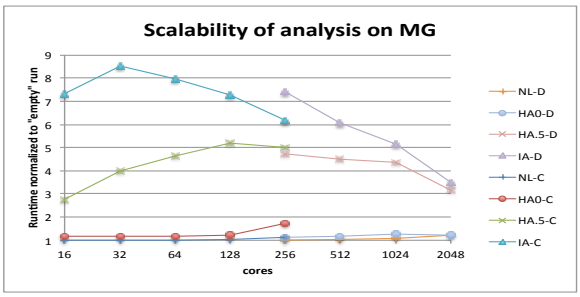


Figure 1: *Breakdown of data race detection overhead running on 16 cores (left). Scalability of the different sampling methods on NPB 2.4 MG, classes C and D (right).*

tion 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).

3. 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 [4]. Overall, instrumentation overhead contributes the most to the slowdown caused by data race detection.

Races Found: In [4] 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.

4. Conclusion

We present the first 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.

References

[1] 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.

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

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

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

[5] 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.