Toward a Data-Centric Profiler for PGAS Applications

Hui Zhang, Jeffrey K. Hollingsworth {hzhang86, hollings}@cs.umd.edu

Department of Computer Science, University of Maryland-College Park

Introduction

Performance profiling is a key tool for parallel computing. PGAS languages value performance as important as productivity, sometimes even more. Chapel [1], as a newer PGAS language, needs a fully supported profiler that can present performance information in an intuitive way, such as a data-centric view, that allows programmers to investigate opportunities of improving the performance of their programs.

We have presented an early snapshot of our on-going work that aims to provide Chapel programmers with a deeper understanding of the bottlenecks in their code. It also provides language developers with insights for future improvement.

Motivation

Finding performance problems is critical in any parallel program. However, the traditional code-centric view of performance data lacks the capability to find problems associated with how different variables are accessed by specific lines in the code. Data-centric approaches that relate performance to data structures rather than code are especially important for HPC/PGAS applications since memory allocation and data movement are often a bottleneck for overall performance. Therefore a profiling tool that can identify the inefficiencies by memory regions is highly desirable.

```
1: int busy(int  *x) {
2:     *x = complex();  //consumes the most time
3:     return *x;
3: }
4: int main() {
5:     for (i=0; i<n; i++) {
6:         A[i] = busy(&B[i]) + busy(&C[i]);
7:     }
8: }
```

Code-centric Profiling

main: 100% latency
busy: 100% latency
complex: 100% latency

Data-centric Profiling

Array A: 100% latency
  main
Array B: 50% latency
  main.busy.complex
Array C: 50% latency
  main.busy.complex

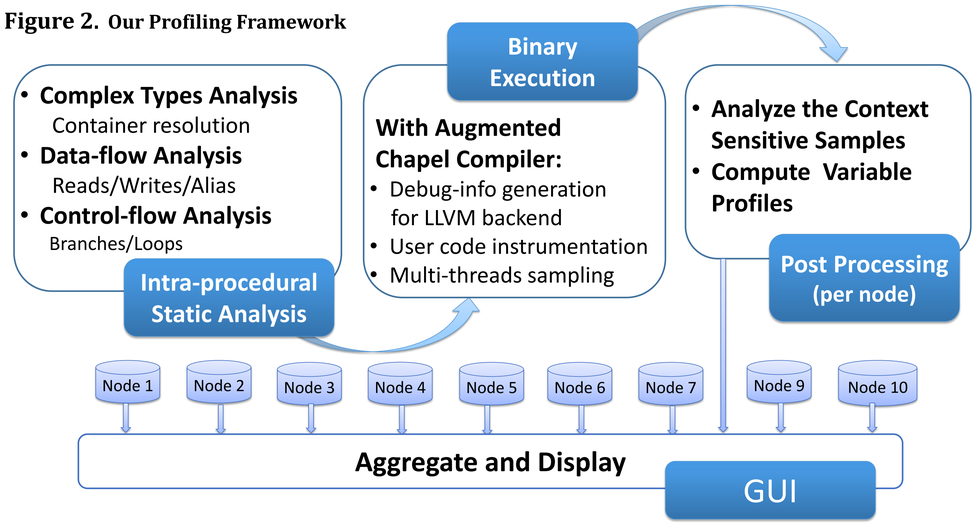
Figure 1. **Code-centric aggregates metrics to the different functions based on sampled lines, while data-centric can distinguish these metrics by different variables**

Method

This work is based on a previous data-centric profiler that focused on parallel programs running on clusters, called "Blame" [2]. Our tool can aggregate the performance data that is associated with the same variable or memory block from all the nodes in the system. The procedure of profiling Chapel programs consists of four steps: (see **Figure 2**)

Static Analysis, Execution Measurement, Post Processing, GUI Display.

Figure 2. **Our Profiling Framework**



Micro-benchmarks

Here are two representative programs that illustrate how the tool works for complex data types and multiple function calls.

```
1: proc grandChild(y_in:real) : real
2: {
3:     var mid2 = y_in;
4:     for i in 1..LARGE {
5:         mid2 = (mid2**2 + 10)/7;
6:     }
7:     return mid2;
8: }
```

Listing 1.
Function Hierarchy

```
9:  proc child(x_in: real) : real {
10:     var mid1 = grandChild(x_in),
11:     var val1;
12:     for i in 1..LARGE {
13:         val1 = (val1*3+11)/7 + mid1;
14:     }
15:     return val1;
16: }
17: proc main() {
18:     var bar = 1.1;
19:     var foo = child(bar);
20: }
```

```
1: record Birthday {
2:     var year: int;
3:     var month: int;
4:     var day: int;
5: }

7: record Actor {
8:     var name: string;
9:     var bd: Birthday;
10:}
```

Listing 2.
Data Hierarchy

```
11: proc main() {
12:     var ActorA = new Actor();
13:     var ActorB = new Actor();
14:     var localVar = 0.1, mid = 1;
15:     for i2 in 1..LARGE {
16:         mid = i%8;
17:         ActorA.bd.month = (mid**3+6)%12 + 1;
18:         ActorA.bd.day = (mid**2+8)%6;
19:         localVar = localVar**2;
20:     }
21:     for i in 1..LARGE/2 {
22:         ActorB.bd.year=(ActorB.bd.year**2+8)%6;
23:     }
24: }
```

Static Analysis

Table 1. Related source lines for each important local variable in **Listing 1**.

Name	Function	Line numbers
foo	main	3~5,10~13,18,19
val1	child	3,4,5,10,11,12,13
mid1	child	3,4,5,10
mid2	grandChild	3,4,5

Table 2. Related source lines for each important local variable in **Listing 2**.

Name	Function	Line numbers
ActorA	main	12,14~18
ActorA.bd	main	12,14~18
ActorA.bd.month	main	12,14~17
ActorA.bd.day	main	12,14~16,18
ActorB	main	13,21,22
ActorB.bd	main	13,21,22
ActorB.bd.year	main	13,21,22
mid	main	14,15,16
localVar	main	14,15,19

Results

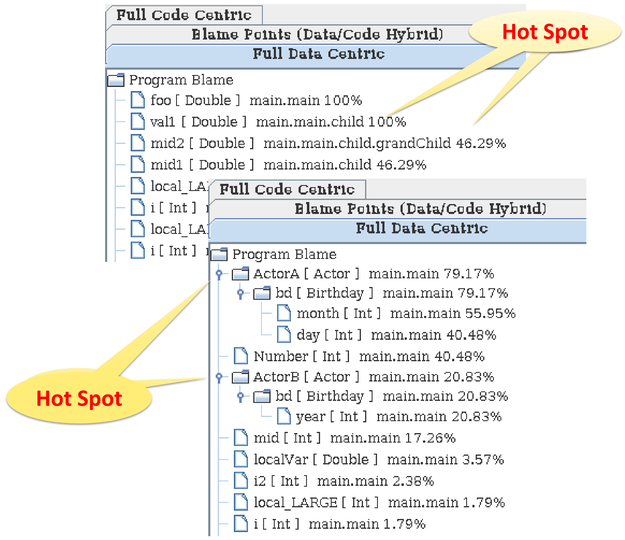


Figure 3. **GUI output for Listing 1 and 2**

Conclusion

We extended the Chapel compiler to make our tool's static analysis available to the LLVM backend, and added hooks to instrument the Chapel implicit multithreading feature. From the simple experiment result, we can see the unique insights that only data-centric profiling can provide. We believe that, through this tool and our future work on supporting the combination of data parallelism and task parallelism in multi-locale environment, developers and researchers will gain better ideas about Chapel or more general PGAS models.

References

[1] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online].Available: http://dx.doi.org/10.1177/1094342007078442.

[2] Rutar, Nick, and Jeffrey K. Hollingsworth. "Data centric techniques for mapping performance data to program variables." Parallel Computing 38.1 (2012): 2-14.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Numbers ER26143 and ER26054.

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND