Toward a Data-Centric Profiler for PGAS Applications

Hui Zhang, Jeffrey K. Hollingsworth  {hzhng86, 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.

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.

Method

With Augmented Chapel Compiler:

• Debug-info generation for LLVM backend
• User code instrumentation
• Multi-threads sampling

Intra-procedural Static Analysis

• Complex Types Analysis
• Data-flow Analysis

Node 1

Node 2

Node 3

Node 4

Node 5

Node 6

Node 7

Node 8

Node 9

Node 10

Intra-procedural Static Analysis

Static Analysis

Binary Execution

Analyzer (per node)

Post Processing

Aggregate and Display

Function Hierachy

Node 1

Node 2

Node 3

Node 4

Node 5

Node 6

Node 7

Node 8

Node 9

Node 10

Micro-benchmarks

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

1: int busy(int *x) {
  2:   if (*x) { // Consumes the most time
  3:     return *x;
  4:   }
  5:   else {
  6:     if (x>0) {
  7:       x = x+1;
  8:     }
  9:     return 0;
  10:   }
  11: }

1: proc grandChild(y_in:real, t_in:real) { // rise the bars
  2:   printf("Height: ", y_in);
  3:   for i in 1..LARGE { // много циклов
  4:     mid = (val1*3+11)/7 + mid1;
  5:     if (i>1) {
  6:       y_in = y_in+1;
  7:     }
  8:     return mid2;
  9:   }
  10: }

1: proc main() {
  2:   for i in 1..LARGE { // much latency
  3:     mid = mid + var1;
  4:   }
  5:   return 0;
  6: }

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Line numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>main</td>
<td>9, 10, 11, 18, 19</td>
</tr>
<tr>
<td>child</td>
<td></td>
<td>3, 4, 5, 10, 11, 12, 13</td>
</tr>
<tr>
<td>mid1</td>
<td></td>
<td>3, 4, 5, 10</td>
</tr>
<tr>
<td>mid2</td>
<td></td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>grandChild</td>
<td></td>
<td>3, 4, 5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Line numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActorA</td>
<td>main</td>
<td>12, 14, 18</td>
</tr>
<tr>
<td>ActorA</td>
<td>bd</td>
<td>12, 14, 18</td>
</tr>
<tr>
<td>ActorA</td>
<td>bd.month</td>
<td>12, 14, 18</td>
</tr>
<tr>
<td>ActorA</td>
<td>bd.day</td>
<td>12, 14, 16, 18</td>
</tr>
<tr>
<td>ActorB</td>
<td>main</td>
<td>13, 21, 22</td>
</tr>
<tr>
<td>ActorB</td>
<td>bd</td>
<td>13, 21, 22</td>
</tr>
<tr>
<td>ActorB</td>
<td>bd.year</td>
<td>13, 21, 22</td>
</tr>
<tr>
<td>mid</td>
<td></td>
<td>14, 15, 16</td>
</tr>
<tr>
<td>localVar</td>
<td></td>
<td>14, 15, 16</td>
</tr>
</tbody>
</table>

Results

Static Analysis

Analyzing the Context

Sensitive Samples

Compute Variable Profiles

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


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 DE-SC14043 and ER26054.

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

Figure 2. Our Profiling Framework

Figure 3. GUI output for Listing 1 and 2

Post-processing, GUI Display.

Code-centric Profiling

Data-centric Profiling

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

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

© 2015 PosterPresentations.com