ChplBlamer: A Data-centric and Code-centric Combined Profiler for Multi-locale Chapel Programs

Hui Zhang, Jeffrey K. Hollingsworth
{hzhang86, hollings}@cs.umd.edu
Department of Computer Science, University of Maryland- College Park
Multi-locale Chapel Environment
Motivation

• **Why PGAS** (Partitioned Global Address Space)
  ▪ Parallel programming is too hard
  ▪ Unified solution for mixed mode parallelism

• **Why Chapel**
  ▪ Chapel is an emerging PGAS language with productive parallel programming features
  ▪ Potential for performance improvement (especially in multi-locale) and few Chapel profilers for its users
  ▪ Insights for evolving the language in the future and the same idea can be applied to other parallel programming paradigms through generic approaches
Data-centric Profiling

```c
int busy(int *x) {
    // hotspot function
    *x = complex();
    return *x;
}

int main() {
    for (i=0; i<n; i++) {
        A[i] = busy(&B[i]) +
        busy(&C[i-1]) +
        busy(&C[i+1]);
    }
}
```

Code-centric Profiling
- main: 100%
- busy: 100%
- complex: 100%

Data-centric Profiling
- A: 100%
- B: 33.3%
- C: 66.7%
What is “ChplBBlamer”?

"MISS HARPER—GET ME SOMEBODY TO BLAME."
Properly Assign Blame

“I didn’t say you were to blame... I said I am blaming you.”
Blame Definition

1) \( \text{BlameSet}(v) = \bigcup_{w \in W} \text{BackwardSlice}(w) \)

2) \( \text{isBlamed}(v, s) = \{ \text{if } (s \in \text{BlameSet}(v)) \text{ then } 1 \text{ else } 0 \} \)

3) \( \text{BlamePercentage}(v, S) = \frac{\sum_{s \in S} \text{isBlamed}(v, s)}{|S|} \)

- \( v \): a certain variable
- \( w \): a write statement to \( v \)’s memory region
- \( W \): a set of \( w \) (all write statements to \( v \)’s memory region)
- \( s \): a sample
- \( S \): a set of samples
### Blame Calculation

1. \( a = 8; \) //Sample 1
2. \( b = a * a; \) //Sample 2,3
3. \( \text{for} \ (i = 0; i < N; i++) \) //Sample 4
4. \( b = b + i; \)
5. \( c = a + b; \) //Sample 5

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>a</th>
<th>b</th>
<th>( i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result Type</td>
<td>inc</td>
<td>exc</td>
<td>inc</td>
</tr>
<tr>
<td>BlameSet</td>
<td>1</td>
<td>1</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>Blame Samples</td>
<td>S1</td>
<td>S1</td>
<td>S1,2,3,4</td>
</tr>
<tr>
<td>Blame</td>
<td>20%</td>
<td>20%</td>
<td>80%</td>
</tr>
</tbody>
</table>
ChplBlamer Framework

1: Intraprocedural Static Analysis
- **Module:** Global Variables, Type Analysis (class, record)
- **Function:** Local Variables, Parameters, Return Values

2: Monitored Execution
- Run the Program with Sampling and Instrumentation Enabled

3: Post Processing
- Reconstruct calling context for samples and combine static info
- Variable Profiles (Per Node)

4: GUI Presentation
- Aggregate Data from All Nodes and Display

Multi-locale Challenges

• 1st Challenge:
Aggregate blame of many temporary variables that point/refer to the distributed variables through remote data accesses.

• Solution:
  ▪ Link variable PvID (privatized id) with different objects accessed through specific Chapel runtime functions: `chpl_getPrivatizedCopy`, and `chpl_getPrivatizedClass`. 
Multi-locale Challenges

• **2\textsuperscript{nd} Challenge:**
  - Recover the **hidden** data-flow information from Chapel internal module calls, e.g., `chpl_gen_comm_get`
  - Recover the **interrupted** data-flow information from Chapel runtime calls, e.g., `chpl_taskListAddBegin`

• **Solution:**
  - Conduct simplified blame analysis for Chapel module functions to get data-dependencies between parameters
  - Resolve actual wrapper task function statically through function pointers that were passed to certain Chapel runtime functions
Multi-locale Challenges

• 3rd Challenge:

Reconstruct the full calling context for each sample and handle asynchronous&remote tasking

• Solution:

  ▪ Instrument Chapel tasking and communication layer
  ▪ Log “task function ID”, “task sender’s locale ID”, and “task receiver’s locale ID” for each remote task
  ▪ Iteratively glue stacktraces to the current calling context until having the user “main” frame
New Tool Feature
Load Imbalance Check

Node information for $Ab$ of HPL on 32 locales
## Experiment – ISx

### Data-centric

<table>
<thead>
<tr>
<th>Name</th>
<th>2-loc</th>
<th>8-loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>myBucketedKeys</td>
<td>41.1%</td>
<td>22.9%</td>
</tr>
<tr>
<td>myKeys</td>
<td>36.9%</td>
<td>20.9%</td>
</tr>
<tr>
<td>sendOffsets</td>
<td>27.3%</td>
<td>15.4%</td>
</tr>
<tr>
<td>bucketOffsets</td>
<td>26.9%</td>
<td>15.2%</td>
</tr>
<tr>
<td>barrier</td>
<td>10.3%</td>
<td>20.8%</td>
</tr>
</tbody>
</table>

### Code-centric

<table>
<thead>
<tr>
<th>Name</th>
<th>2-loc</th>
<th>8-loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucketSort</td>
<td>80.9%</td>
<td>64.2%</td>
</tr>
<tr>
<td>bucketizeLocalKeys</td>
<td>40.2%</td>
<td>22.3%</td>
</tr>
<tr>
<td>countLocalKeys</td>
<td>11.4%</td>
<td>6.4%</td>
</tr>
<tr>
<td>pthread_spin_lock</td>
<td>16.7%</td>
<td>29.3%</td>
</tr>
<tr>
<td>chpl_comm_barrier</td>
<td>0</td>
<td>3.46%</td>
</tr>
</tbody>
</table>

### OPTIMIZATION:

1. Optimize “Barrier” module
2. Apply “local” clause

### Table

<table>
<thead>
<tr>
<th>Name</th>
<th>original</th>
<th>localization</th>
</tr>
</thead>
<tbody>
<tr>
<td>myBucketedKeys</td>
<td>41.11%</td>
<td>17.78%</td>
</tr>
<tr>
<td>sendOffsets</td>
<td>27.28%</td>
<td>6.02%</td>
</tr>
<tr>
<td>bucketOffsets</td>
<td>26.85%</td>
<td>5.46%</td>
</tr>
<tr>
<td>bucketizeLocalKeys</td>
<td>40.24%</td>
<td>24.54%</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Blame</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>Elems</td>
<td>Struct</td>
<td>74.3%</td>
</tr>
<tr>
<td>elemToNode</td>
<td>Struct</td>
<td>60.4%</td>
</tr>
<tr>
<td>xd/yd/zd</td>
<td>Struct</td>
<td>48.0%</td>
</tr>
<tr>
<td>x/y/z</td>
<td>Struct</td>
<td>37.0%</td>
</tr>
<tr>
<td>fx/fy/fz</td>
<td>Struct</td>
<td>35.6%</td>
</tr>
<tr>
<td>dvdx/dvdy/dvdz</td>
<td>Struct</td>
<td>33.4%</td>
</tr>
<tr>
<td>x8n/y8n/z8n</td>
<td>Struct</td>
<td>33.3%</td>
</tr>
<tr>
<td>elemMass</td>
<td>Struct</td>
<td>29.5%</td>
</tr>
<tr>
<td>hgfx/hgfy/hgfz</td>
<td>Array</td>
<td>26.7%</td>
</tr>
<tr>
<td>shx/shy/shz</td>
<td>Double</td>
<td>26.7%</td>
</tr>
<tr>
<td>hx/hy/hz</td>
<td>Array</td>
<td>26.6%</td>
</tr>
<tr>
<td>dxx/dyy/dzz</td>
<td>Struct</td>
<td>12.2%</td>
</tr>
</tbody>
</table>
### Problem:

```plaintext
proc CalcHourglassControlForElems (determ) {
    var dvdx, dvdy, dydz, x8n, y8n, z8n: [Elems] 8*real;
    ...
}
```

### Solution:

Hoisting distributed local variables to the global space so that they won’t be dynamically allocated frequently.

### Result:

![Graph showing execution time comparison between Original and Globalization for different #nodes]
LULESH Optimization: Replication

### Problem:
Frequent calls to "\texttt{localizeNeighborNodes}" on these variables which incurs sequential remote data accesses.

```latex
\textbf{for} i \ \textbf{in} 1..\texttt{nodesPerElem}
\{ \\
\texttt{const} \ \texttt{noi} = \texttt{elemToNode[eli][i]}; \\
x\_local[i] = x[\texttt{noi}]; \\
y\_local[i] = y[\texttt{noi}]; \\
z\_local[i] = z[\texttt{noi}]; \\
\}
```

### Solution:
Allocate global maps to prestore neighboring nodes for each element using the same domain: \texttt{var x\_map: [Elems] nodesPerElem*real}
Conclusion

- Data-centric Profiling and Blame Analysis
- Multi-locale Support and New Features
- Benchmark Profiling and Optimization

move from having slowdown as more locales were added to having speedups!