Automatic Detection of MPI Application Structure with Event Flow Graphs

Karl Fürlinger\textsuperscript{1}

joint work with
Xavier Aguilar\textsuperscript{2} and Erwin Laure\textsuperscript{2}

\textsuperscript{1} Ludwig-Maximilian-University (LMU)
Munich, Germany

\textsuperscript{2} KTH Royal Institute of Technology
Stockholm, Sweden
Trace

- Full temporal order of events is preserved
- A lot of data to store, process, analyze

Profile (summary)

- Temporal order is not preserved
- Far less data

Implementation in IPM

- Keep data in a hash table
- Keys: event (-signatures)
- Values: statistics (#calls, duration, ...)

<table>
<thead>
<tr>
<th>key</th>
<th>#calls</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>42</td>
<td>23.1</td>
</tr>
<tr>
<td>A</td>
<td>100</td>
<td>12.0</td>
</tr>
</tbody>
</table>

1 Integrated Performance Monitor
http://ipm-hpc.sourceforge.net/
Event Flow Graphs (EFGs)
- Keep a **history** of the previous event that happened
- Keep track of pairs of events \((\text{prev.}, \text{curr.})\) instead of single events

Similar to a control flow graph, but
- records transitions that have actually happened in an execution
- records how many times these transitions have happened

Implementation in IPM:
- Keep an **additional hash table**
- Keys: pairs of events \((\text{prev.}, \text{curr.})\)
- Values: statistics \((\#\text{transitions}, \text{duration}, \ldots)\)

<table>
<thead>
<tr>
<th>key</th>
<th>#trans.</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>7</td>
</tr>
</tbody>
</table>
```c
int main() {
    MPI_Init(...);
    for(i=1;i<=10;i++) {
        MPI_Send(...);
        MPI_Recv(...);
    }
    MPI_Finalize();
}
```

In this case, the EFG is a perfect representation of the trace.
In this case, the trace cannot be uniquely reconstructed from the EFG.

```c
int main() {
    MPI_Init(...);
    for(i=1;i<=10;i++) {
        MPI_Bcast(...);
        if(i%2) /* odd */
            MPI_Recv(...);
        else /* even */
            MPI_Send(...);
    }
    MPI_Finalize();
}
```
Temporal EFG (t-EFG):
- A modified version of an EFG that guarantees trace recovery

Ideas
- At each node, keep track of which outgoing edge to take next
- Represent this information in a compact way

t-EFG for the previous example:
- Edge label describes a partition of the iteration space

1,9,2,1: first, last, stride, blocksize

2,1: notation for simple case
Runtime data collection is still efficient

- Around 2% overhead in terms of execution time
- See: [EuroPar ’14]: Xavier Aguilar, et al. MPI Trace Compression using Event Flow Graphs

Compression results for some benchmarks [EuroPar ’14] (sequence of events only)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Ranks</th>
<th>Comp. Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td>96</td>
<td>1.76x</td>
</tr>
<tr>
<td>GTC</td>
<td>64</td>
<td>46.60x</td>
</tr>
<tr>
<td>MILC</td>
<td>96</td>
<td>39.03x</td>
</tr>
<tr>
<td>SNAP</td>
<td>96</td>
<td>119.23x</td>
</tr>
<tr>
<td>MiniDFT</td>
<td>40</td>
<td>4.33x</td>
</tr>
<tr>
<td>MiniFE</td>
<td>144</td>
<td>19.93x</td>
</tr>
<tr>
<td>MiniGhost</td>
<td>96</td>
<td>4.85x</td>
</tr>
</tbody>
</table>

Up to 120x Compression!
Compression ratio depends on the structure of the graphs
- Simple graphs with few nodes and edges correspond to high compression ratios

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td>1.76</td>
<td>9,384.94</td>
<td>10,586.47</td>
<td>4.59</td>
</tr>
<tr>
<td>MiniDFT</td>
<td>4.33</td>
<td>690.30</td>
<td>1,980.38</td>
<td>27.29</td>
</tr>
<tr>
<td>SNAP</td>
<td>119.23</td>
<td>28</td>
<td>1,120.26</td>
<td>14,149.22</td>
</tr>
<tr>
<td>GTC</td>
<td>46.60</td>
<td>114.5</td>
<td>121.20</td>
<td>109.10</td>
</tr>
</tbody>
</table>
Overview (1)

Event Flow Graph

start

end

EUROPAR ’14

Xavier Aguilar, Karl Fürlinger, and Erwin Laure.

MPI Trace Compression using Event Flow Graphs

EuroPar ’14:

Xavier Aguilar, Karl Fürlinger, and Erwin Laure.

MPI Trace Compression using Event Flow Graphs
MiniGhost example application
- 3160 events in the trace
- 87 nodes, 90 edges in the EFG

Compressing sequences (chains)
- 13 nodes, 16 edges
- Nested loops (cycles) visible
Detecting Application Structure Automatically

- **Application Structure**
  - Structure:= loops and their nesting
  - Folklore: “big outer loop hypothesis”: most scientific applications are dominated by a **big outer time-stepping loop**

- **Detecting Structure**
  - If a loop contains MPI calls, the loop will show up as a **cycle** in the Event Flow Graph

![Flow Graph Diagram]
Finding Cycles in the Graph

- Detecting cycles in flow graphs is a common requirement for (de-)compilers
  - Many algorithms exist
  - We used an efficient DFS-based algorithm by T. Wei et al., “A New Algorithm for Identifying Loops in Decompilation”, 2007

```c
for ( i = 0; ... ) {
    A( );
    for ( j = 0; ... ) {
        B( );
        C( );
    }
    D( );
}
```
Loop Detection Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Ranks</th>
<th>Total Runtime (sec)</th>
<th>Count</th>
<th>Time in all</th>
<th>Time in dominant</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniGhost</td>
<td>96</td>
<td>282.17</td>
<td>1</td>
<td>98.8%</td>
<td>98.8%</td>
</tr>
<tr>
<td>MiniFE</td>
<td>144</td>
<td>133.50</td>
<td>13</td>
<td>78.1%</td>
<td>77.7%</td>
</tr>
<tr>
<td>BT</td>
<td>144</td>
<td>370.59</td>
<td>7</td>
<td>99.4%</td>
<td>99.0%</td>
</tr>
<tr>
<td>LZ</td>
<td>128</td>
<td>347.53</td>
<td>3</td>
<td>99.2%</td>
<td>98.9%</td>
</tr>
</tbody>
</table>

“Big outer loop hypothesis” largely holds for these (and other) example benchmarks
Event Flow Graph

Trace

Temporal Event Flow Graph
Online Structure Detection

- So far: post-mortem operation

- Now: Online operation

**Steady state?**
- No → do nothing
- Yes → perform loop detection

**At main loop header?**
- No → do nothing
- Yes → collect trace for N iterations ("smart data collection")
Application structure can be detected online, while the application runs
- Reduce redundant data, change data granularity, etc

The event flow graph becomes stable once the application enters its iterative phase

Our mechanism checks the number of nodes in the graph to detect application stability to trigger the loop detection mechanism
EFG Stability

![Graph showing the execution time and number of nodes for LU, MiniFE, and MiniGhost.](image)

- **Execution time (seconds)**: The graph plots the execution time on the x-axis.
- **Num. nodes**: The graph plots the number of nodes on the y-axis.

- **LU**: A solid line representing the execution time and number of nodes for LU.
- **MiniFE**: A dotted line representing the execution time and number of nodes for MiniFE.
- **MiniGhost**: A dashed line representing the execution time and number of nodes for MiniGhost.
Six applications representing typical scientific codes
- MiniGhost
- MiniFE
- MiniMD
- GTC
- LU
- BT

Cray XE6 with 2 twelve-core AMD MagnyCours at 2.1 GHz
- 32 GB DDR3 memory per node
- Nodes interconnected with Cray Gemini network
## Smart Data Collection – Trace Size

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mini-Ghost</th>
<th>MiniFE</th>
<th>GTC</th>
<th>MiniMD</th>
<th>BT</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace size</td>
<td>26 MB</td>
<td>77 MB</td>
<td>48 MB</td>
<td>555 MB</td>
<td>717 MB</td>
<td>7.7 GB</td>
</tr>
<tr>
<td>10 iterations</td>
<td>4.4 MB</td>
<td>4.1 MB</td>
<td>1.3 MB</td>
<td>788 KB</td>
<td>29 MB</td>
<td>267 MB</td>
</tr>
<tr>
<td>trace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>% reduced</td>
<td>83%</td>
<td>94.7%</td>
<td>97.3%</td>
<td>99.8%</td>
<td>96%</td>
<td>96.53%</td>
</tr>
</tbody>
</table>

- Detect the application structure on-line to keep tracing information of only 10 iterations of the main loop
- If the application is regular, a few iterations will represent the overall performance behaviour
- Performance results (statistics) still representative
Overview (3)

Event Flow Graph

Start

A

B

C

D

End

Trace

A

B

A

D

C

D

Temporal Event Flow Graph

Start

A

B

C

D

End

(ongoing work)

Loop 1

A

Loop 2

D

B

C

SEQ

A

D

B

LOOP (100x)

SEQ

C

B

SEQ

C

B

SEQ

C

B

SEQ

C

B
Example: MiniGhost

- **ROOT**
  - **SEQUENCE**
    - MPI_Init
    - Seq 1 (length 9)
  - **LOOP** (60x)
    - **SEQUENCE**
      - Node A, Node B
    - **LOOP** (6x)
      - **SEQUENCE** [3,3,0,1]
        - Node C, Node G, Node F
      - **SEQUENCE** [1,1,0,1]
        - Node C, Node E, Node D
      - **SEQUENCE** [0,2,2,1 | 4,5,0,2]
        - Node C
  - **SEQUENCE**
    - Seq 3 (length 39)
    - Node H
  - **SEQUENCE**
    - Seq 2 (length 29)
    - MPI_Finalize

- **Predicate guards the activation of the node**
- Compact and clear representation of what the application does
- Code generation straightforward
Conclusions

- Event flow graphs together with graph cycle detection algorithms are able to detect MPI application structure

- No source instrumentation needed
  - Graphs captured through the PMPI interface

- Some use cases:
  - Map performance data to program structure
  - Reduce amount of data collected while application runs

- Converting t-EFGs to trees ongoing work
  - Exciting possibilities: analysis, modeling, code generation, ...