Automated Analysis of Time Series Data to Understand Parallel Program Behaviors

LAI WEI, JOHN MELLOR-CRUMMEY
RICE UNIVERSITY
HOUSTON, TX, USA
Background

Parallel computers of increasing scale
- Support scientific simulations of increasing ambition

Performance of many applications fail to scale accordingly
- Load imbalance, serialization, network congestion, etc.

Performance tools to understand application behaviors
- Measure and present performance data
- Used by experts to manually identify performance inefficiencies
Profile

Breaks down application run time into sources of costs

### Calling context
- **main()**
- **init()**
- **solve()**
- **compute()**
- **sync()**

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
</tr>
<tr>
<td>init()</td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
</tr>
<tr>
<td>solve()</td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
</tr>
<tr>
<td>compute()</td>
<td>4s</td>
<td>4.1s</td>
<td>3.9s</td>
<td>4s</td>
</tr>
<tr>
<td>sync()</td>
<td>4s</td>
<td>3.9s</td>
<td>4.1s</td>
<td>4s</td>
</tr>
</tbody>
</table>
Profile

Breaks down application run time into sources of costs

**Calling context**

- **main()**
- **init()**
- **solve()**
- **compute()**
- **sync()**

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
</tr>
<tr>
<td></td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
</tr>
<tr>
<td></td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
</tr>
<tr>
<td></td>
<td>4s</td>
<td>4.1s</td>
<td>3.9s</td>
<td>4s</td>
</tr>
<tr>
<td><strong>compute()</strong></td>
<td><strong>4s</strong></td>
<td><strong>3.9s</strong></td>
<td><strong>4.1s</strong></td>
<td><strong>4s</strong></td>
</tr>
</tbody>
</table>
Profile

Breaks down application run time into sources of costs

<table>
<thead>
<tr>
<th>Calling context</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
<td>9s</td>
</tr>
<tr>
<td>init()</td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
<td>1s</td>
</tr>
<tr>
<td>solve()</td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
<td>8s</td>
</tr>
<tr>
<td>compute()</td>
<td>4s</td>
<td>4.1s</td>
<td>3.9s</td>
<td>4s</td>
</tr>
<tr>
<td>sync()</td>
<td>4s</td>
<td>3.9s</td>
<td>4.1s</td>
<td>4s</td>
</tr>
</tbody>
</table>

Performance loss, why?
Time series

Presents application behavior over time

Calling context

- main()
- init()
- solve()
- compute()
- sync()

Depth = 1

init() and solve() timelines for different processes (P0, P1, P2, P3)
Time series

Presents application behavior over time

Calling context

Depth = 2
Time series

Presents application behavior over time

Calling context

<table>
<thead>
<tr>
<th>Depth = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
</tr>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P3</td>
</tr>
</tbody>
</table>

init() → compute() → sync()

main() → init() → solve() → compute() → sync()
Time series

Presents application behavior over time

Calling context

main()
init()
solve()
compute()
sync()

Depth = 2

Load imbalance

P0
init() compute() sync()

P1
compute() sync()

P2
compute() sync()

P3
compute() sync()
Time series

Presents application behavior over time

Calling context
- main()
- init()
- solve()
- compute()
- sync()

Depth = 2

Load imbalance
Time series

Presents application behavior over time

Calling context

main()
init()
solve()
compute()
sync()

Depth = 2

Load imbalance

Depth = 2
Motivation

Experts manually examine time series
- Understand how and why performance inefficiencies arise

Time series of large scale parallel executions
- Vast in three dimensions
  - Process
  - Time
  - Call path depth

- Manual analysis is difficult if not impractical
Related work -- automated analysis

Analysis of profiles [Huck, SC’05] [Tallent, SC’10]
- Often insufficient for diagnosing how and why parallel inefficiencies arise

Analysis of execution traces
- Collecting instrumentation-based traces are costly in time and space
  - Fine-grained traces explode at large scale

- Analysis at coarse granularity [Gonzalez, IPDPS’09] [Llort, IPDPS’10]
  - Still needs lots of manual effort
- Analysis at fine granularity for short intervals [Geimer, CCPE’10] [Böhme, TOPC’16]
  - Requires prior knowledge for selective tracing
Our contribution

Automated analysis of sample-based time-series data

- Feasible for large-scale programs
  - Data volume is manageable

- Derive compact top-down summaries
  - Uncover patterns and variance
  - Direct attention to potential performance losses
  - Attribute losses to code regions where they originate
Approach

1. Collect and prepare sample-based time-series for further analysis
   ◦ Collect a time series of call paths with HPCToolkit
   ◦ Organize each time series as a tree of program calling contexts
   ◦ Identify iterative behaviors in the time series

2. Build clusters across threads and loop iterations

3. Quantify performance losses and attribute them to call paths
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
   C();
   for (int k = 0; k < 3; k++) {
      A();
      if (k==0) B();
      if (k==2) C();
   }
}
int main() {
   foo();
   return 0;
}
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}

int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Depth = 0
Depth = 1
Depth = 2
Depth = 3
Collect call path samples over time

```c
void A() {
    ...
}

void B() {
    ...
}

void C() {
    ...
}

void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}

int main() {
    foo();
    return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Depth = 0
Depth = 1
Depth = 2
Depth = 3
Collect call path samples over time

```c
void A() { ... }  
void B() { ... }  
void C() { ... }  
void foo() {
  C();
  for (int k = 0; k < 3; k++) {
    A();
    if (k==0) B();
    if (k==2) C();
  }
}
int main() {
  foo();
  return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Depth = 3
Depth = 2
Depth = 1
Depth = 0
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Diagram:
- `main()` (Depth = 0)
- `foo@13` (Depth = 1)
- `C@5` (Depth = 2)
- T1 (Depth = 3)
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

- **Depth = 0**
  - main()
  - C@5
  - T1

- **Depth = 1**
  - foo@13

- **Depth = 2**

- **Depth = 3**
Collect call path samples over time

```c
1 void A() { ... }
2 void B() { ... }
3 void C() { ... }
4 void foo() {
5    C();
6    for (int k = 0; k < 3; k++) {
7        A();
8        if (k==0) B();
9        if (k==2) C();
10    }
11 }
12 int main() {
13    foo();
14    return 0;
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3

T1
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Depth:
- `main()`: Depth = 3
- `foo@13`: Depth = 3
- `C@5`: Depth = 2
- `T1`, `T2`: Depth = 1
- `Depth = 0` (root)
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
1 void A() { ...
2 void B() { ...
3 void C() { ...
4 void foo() {
5     C();
6     for (int k = 0; k < 3; k++) {
7         A();
8         if (k==0) B();
9         if (k==2) C();
10     }
11 }
12 int main() {
13     foo();
14     return 0;
15 }
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
1  void A() { ... }
2  void B() { ... }
3  void C() { ... }
4  void foo() {
5       C();
6       for (int k = 0; k < 3; k++) {
7           A();
8           if (k==0) B();
9           if (k==2) C();
10      }
11 }
12 int main() {
13     foo();
14     return 0;
15 }
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Collect call path samples over time

```c
1 void A() { ... }
2 void B() { ... }
3 void C() { ... }
4 void foo() {
5     C();
6     for (int k = 0; k < 3; k++) {
7         A();
8         if (k==0) B();
9         if (k==2) C();
10     }
11 }
12 int main() {
13     foo();
14     return 0;
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Diagram:
- `main()`: Depth = 0
- `foo@13`: Depth = 1
- `loop@6`: Depth = 2
- `A@7`: Depth = 3

Paths:
- T1: Depth = 0
- T2: Depth = 3
Collect call path samples over time

```c
void A() { ...
void B() { ...
void C() { ...
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() {
    ...
}
void B() {
    ...
}
void C() {
    ...
}
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Diagram:
```
main() -> foo@13
        |          |
        |          |         |
        C@5      loop@6  A@7
        |          |          |
        |          |          |         |
        |          |          |          |         |
T1       T2       T3
        |          |          |          |
        |          |          |          |         |
        |          |          |          |         |
        |          |          |          |         |
Depth = 0 | Depth = 1 | Depth = 2 | Depth = 3
```
Collect call path samples over time

```c
1 void A() { ... }
2 void B() { ... }
3 void C() { ... }
4 void foo() {
5     C();
6     for (int k = 0; k < 3; k++) {
7         A();
8         if (k==0) B();
9         if (k==2) C();
10     }
11 }
12 int main() {
13     foo();
14     return 0;
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() { ...
  C();
  for (int k = 0; k < 3; k++) {
    A();
    if (k==0) B();
    if (k==2) C();
  }
int main() {
  foo();
  return 0;
}
```
Collect call path samples over time

```c
1 void A() { ... }
2 void B() { ... }
3 void C() { ... }
4 void foo() {
5     C();
6     for (int k = 0; k < 3; k++) {
7         A();
8         if (k==0) B();
9         if (k==2) C();
10     }
11 }
12 int main() {
13     foo();
14     return 0;
15 }
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Collect call path samples over time

1. void A() { ... }
2. void B() { ... }
3. void C() { ... }
4. void foo() {
   C();
   for (int k = 0; k < 3; k++) {
      A();
      if (k==0) B();
      if (k==2) C();
   }
}
5. int main() {
   foo();
   return 0;
}

Depth = 0
Depth = 1
Depth = 2
Depth = 3

T1
T2
T3
T4
T5
Collect call path samples over time

```c
void A() { ...
void B() { ...
void C() { ...
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3
Collect call path samples over time

1. `void A() { ... }
2. void B() { ... }
3. void C() { ... }
4. void foo() {
   5.   C();
   6.   for (int k = 0; k < 3; k++) {
       7.     A();
       8.     if (k==0) B();
       9.     if (k==2) C();
   10. }
11. }
12. int main() {
   13.   foo();
   14.   return 0;
Collect call path samples over time

```c
void A() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k == 0) B();
        if (k == 2) C();
    }
}

int main() {
    foo();
    return 0;
}
```

Depth = 0
Depth = 1
Depth = 2
Depth = 3

T1 T2 T3 T4

T5 T6 T7
Collect call path samples over time

```c
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
}
```
Collect call path samples over time

```c
void A() { ...
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}

int main() {
    foo();
    return 0;
}
```
Construct a **Temporal Context Tree**

![Temporal Context Tree Diagram]

- **main() T1-T8**
- **foo@13 T1-T8**
- **loop@6 T2-T8**

Depth Levels:
- **Depth = 0**
- **Depth = 1**
- **Depth = 2**
- **Depth = 3**
Construct a **Temporal Context Tree**
Construct a *Temporal Context Tree*

One tree per thread
Identify iterations

loop@6 T2-T8

A@7 T2, B@8 T3, A@7 T4-T5, A@7 T6-T7, C@9 T8
Identify iterations

```
for (int k = 0; k < 3; k++) {
    A();
    if (k==0) B();
    if (k==2) C();
}
```
Identify iterations

```c
for (int k = 0; k < 3; k++) {
    A();
    if (k==0) B();
    if (k==2) C();
}
```

Use ParseAPI to analyze binaries
Identify iterations

```
for (int k = 0; k < 3; k++) {
    A();
    if (k==0) B();
    if (k==2) C();
}
```

Use ParseAPI to analyze binaries

Insert a new iteration when a back edge must have been taken
Approach

1. Collect and prepare sample-based time-series for further analysis
   ◦ Collect a time series of call paths with HPCToolkit
   ◦ Organize each time series as a tree of program calling contexts
   ◦ Identify iterative behaviors in the time series

2. Build clusters across threads and loop iterations

3. Quantify performance losses and attribute them to call paths
Clustering

Objective
- Concisely summarize behaviors of a collection of threads executing many iterations
  - Represent a large set of instances with a few representatives
- Identify variations across threads and iterations
  - Variations may indicate performance bottlenecks
  - Is there any variation? How large is it? Where does variation arise?

Steps
- Quantify differences in Temporal Context Trees (TCTs)
- K-farthest clustering [Bahmani, BIG DATA’15]
  - Time complexity = \(O(NK^2G)\). \(N\) = number of instances; \(G\) = size of TCTs
  - Multi-level clustering ✔ Parallelization ✔
Approach

1. Collect and prepare sample-based time-series for further analysis
   ◦ Collect a time series of call paths with HPCToolkit
   ◦ Organize each time series as a tree of program calling contexts
   ◦ Identify iterative behaviors in the time series

2. Build clusters across threads and loop iterations

3. Quantify performance losses and attribute them to call paths
Quantify performance losses

Variation across threads provides clues to performance losses

P0: X = 2s, Y = 7s, Z = 1s
P1: X = 8s, Y = 7s, Z = 1s
P2: X = 2s, Y = 7s, Z = 1s
Quantify imbalance

Assume X as computation, Y & Z as synchronization

For a computation node C

- \( \text{imb}(C) = \) projected reduction in execution time if work in C is balanced across threads
Quantify imbalance

Assume X as computation, Y & Z as synchronization

For a computation node C

$imb(C) =$ projected reduction in execution time if work in C is balanced across threads
Quantify imbalance

Assume X as computation, Y & Z as synchronization

\[
\text{imb}(X) = \max(X) - \text{avg}(X) = 8s - 4s = 4s
\]

For a computation node C

\[\text{imb}(C) = \text{projected reduction in execution time if work in C is balanced across threads}\]
Quantify imbalance

Assume X as computation, Y & Z as synchronization

For a synchronization node S

- \( \text{imb}(S) = \) projected reduction in execution time if work between the prior synchronization and S is balanced.
Quantify imbalance

Assume X as computation, Y & Z as synchronization

\[ \text{imb}(Y) = \text{imb}(X) = 4s \]

For a synchronization node S

- \( \text{imb}(S) = \) projected reduction in execution time if work between the prior synchronization and S is balanced.
Attribute imbalance

Assume X as computation, Y & Z as synchronization

\[ \text{sumImb}(N) \text{ for each node } N \text{ in } TCT \]

\[ = \text{imb}(N) \text{ if } N \text{ is a leaf} \]

\[ = \text{Sum \{} \text{sumImb(every child of } N) \}\]
Attribute imbalance

Assume X as computation, Y & Z as synchronization

<table>
<thead>
<tr>
<th>P0</th>
<th>X = 2s</th>
<th>Y = 7s</th>
<th>imbalances</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>X = 8s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>X = 2s</td>
<td>Y = 7s</td>
<td></td>
</tr>
<tr>
<td>Optimized</td>
<td>X = 4s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ sumImb(N) \] for each node N in TCT

= \( imb(N) \) if N is a leaf

= Sum \{ sumImb(every child of N) \}

imb(N)
## Attribute imbalance

Assume X as computation, Y & Z as synchronization

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>2s</td>
<td>7s</td>
<td>1s</td>
</tr>
<tr>
<td>P1</td>
<td>8s</td>
<td>1s</td>
<td>1s</td>
</tr>
<tr>
<td>P2</td>
<td>2s</td>
<td>7s</td>
<td>1s</td>
</tr>
</tbody>
</table>

Optimized: X = 4s, Y = 4s, Z = 0s

\( sumImb(N) \) for each node N in TCT
- if \( N \) is a leaf
- Sum \( \{ sumImb(every \ child \ of \ N) \} \)
## Attribute imbalance

Assume X as computation, Y & Z as synchronization

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>2s</td>
<td>7s</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>8s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2s</td>
<td>7s</td>
<td></td>
</tr>
</tbody>
</table>

### Optimized

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4s</td>
<td></td>
</tr>
</tbody>
</table>

**sumImb(N)** for each node N in TCT

- \(imb(N)\) if N is a leaf
- \(\text{Sum}\ \{\text{sumImb(every child of N)}\}\)
Attribute imbalance

Assume X as computation, Y & Z as synchronization

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>2s</td>
<td>7s</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>8s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2s</td>
<td>7s</td>
<td></td>
</tr>
</tbody>
</table>

Optimized

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{sumImb}(N)\] for each node N in TCT

= \(\text{imb}(N)\) if N is a leaf

= Sum \{ \text{sumImb}(\text{every child of N}) \}
Highlight significant call paths

Pick call paths that contribute significantly to imbalance
- \( \text{sumImb}(N) / \text{RunTime} > \text{significanceRatio} = 1\% \)
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths
- \( \text{imb}(N) / \text{sumImb}(N) > \text{appropriateDepthRatio} = 70\% \)
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way
Highlight significant call paths

Pick call paths that contribute significantly to imbalance

- \( \frac{\text{sumImb}(N)}{\text{RunTime}} > \text{significanceRatio} \) (= 1%)
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths

- \( \frac{\text{imb}(N)}{\text{sumImb}(N)} > \text{appropriateDepthRatio} \) (= 70%)
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way
Highlight significant call paths

Pick call paths that contribute significantly to imbalance
- \( \frac{\text{sumImb}(N)}{\text{RunTime}} > \text{significanceRatio} (= 1\%) \)
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths
- \( \frac{\text{imb}(N)}{\text{sumImb}(N)} > \text{appropriateDepthRatio} (= 70\%) \)
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way
Highlight significant call paths

Pick call paths that contribute significantly to imbalance
- $\text{sumImb}(N) / \text{RunTime} > \text{significanceRatio} (= 1\%)$
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths
- $\text{imb}(N) / \text{sumImb}(N) > \text{appropriateDepthRatio} (= 70\%)$
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way
Experiments

Platform: Titan @ Oak Ridge National Laboratory
- One 16-core AMD Opteron per node; one thread per core
- Gemini interconnect (3D torus)

Applications:
- PFLOTRAN @ 512 MPI ranks, ~178 seconds
  - Simulation of subsurface flow and reactive transport
  - Chemical reactions; environmental assessment

- AMG2013 @ 512 MPI ranks, ~23 seconds
  - Parallel solver of structured/unstructured linear systems
Manual analysis of PFLOTRAN?

Run time = 178s
Manual analysis of PFLOTRAN?

Run time = 178s
Manual analysis of PFLOTRAN?

Run time = 178s

Need to zoom in to examine the execution at a higher resolution

Need to select appropriate call path depths to derive detailed insights
Visualization with automated insights

Run time = 178s

Cluster #1

P0
P4
P60
P1

Cluster #2

P511

Depth of callpath
Processes are ordered first by clusters and then by MPI rank
Height of each cluster is proportional to $\log_2(size+1)$
Visualization with automated insights

Run time = 178s

Processes are ordered first by clusters and then by MPI rank.
Height of each cluster is proportional to $\log_2(\text{size}+1)$.
Processes are ordered first by clusters and then by MPI rank.
Height of each cluster is proportional to $\log_2(\text{size}+1)$

Each pixel shows a procedure frame on the call path.
Depths are selected by automated analysis.
Visualization with automated insights

Run time = 178s

Processes are ordered first by clusters and then by MPI rank.

Height of each cluster is proportional to $\log_2(size+1)$.

Each pixel shows a procedure frame on the call path. Depths are selected by automated analysis.

Insignificant call paths are colored grey.

Execution cut into several segments.
Visualization with automated insights

Run time = 178s

Processes are ordered first by clusters and then by MPI rank.
Height of each cluster is proportional to $\log_2(size+1)$

Each pixel shows a procedure frame on the call path.
Depths are selected by automated analysis.

Insignificant call paths are colored grey.

Cluster #1
P0
P4
P60
P1

Cluster #2
P511

Execution cut into several segments

Zoom in
Understand behavior of PFLOTRAN
Understand behavior of PFLOTRAN

Cluster #1

P0

P4

P60

P1

Cluster #2

P511

51.97s

81.75s

Synchronization colored in purple
Understand behavior of PFLOTRAN

Cluster #1

P0

Cluster #2

P511

P60

P4

P1

51.97s

81.75s

Work colored in red, yellow, brown

Synchronization colored in purple
Understand behavior of PFLOTRAN

Cluster #1
- P0
- P4
- P60
- P1

Cluster #2
- P511

51.97s → I/O work → 81.75s

Work colored in red, yellow, brown

Synchronization colored in purple
Understand behavior of PFLOTRAN

51.97s  --------------------------  81.75s

Cluster #1

P0
P4
P60
P1

Cluster #2

P511

Work colored in red, yellow, brown
Wait colored in green and blue
Synchronization colored in purple
Understand behavior of PFLOTRAN

Cluster #1
- P0
- P4
- P60
- P1

Cluster #2
- P511

51.97s

81.75s

Work colored in red, yellow, brown

Wait colored in green and blue

Synchronization colored in purple

MPI_Send

MPI_Alltoall

I/O work
Understand behavior of PFLOTRAN

51.97s 81.75s

Cluster #1
P0
P4
P60
P1

Cluster #2
P511

MPIT Alltoal

Extreme load-imbalance, probably serialization

Wait colored in green and blue
Synchronization colored in purple

Work colored in red, yellow, brown

MPI_Send

Extreme load-imbalance, probably serialization

MPI_Alltoall

I/O work
Sketch of PFLOTRAN serialized I/O

If I’m P0
- Write global grid to visualization file

For -- (runs 3 iterations)
- MPI_Alltoall within local MPI group
- If I’m not P0
  - MPI_Send my data to P0
- Else
  - Write my data to visualization file
  - MPI_Recv data from P1 to P511 and write to visualization file
Sketch of PFLOTRAN serialized I/O

If I’m P0
- Write global grid to visualization file

For -- (runs 3 iterations)
- MPI_Alltoall within local MPI group
- If I’m not P0
  - MPI_Send my data to P0
- Else
  - Write my data to visualization file
  - MPI_Recv data from P1 to P511 and write to visualization file
Sketch of PFLOTRAN serialized I/O

If I’m P0
- Write global grid to visualization file

For -- (runs 3 iterations)
- MPI_Alltoall within local MPI group
- If I’m not P0
  - MPI_Send my data to P0
- Else
  - Write my data to visualization file
  - MPI_Recv data from P1 to P511 and write to visualization file

Symptom of serialization

Cause of serialization
Sketch of PFLOTRAN serialized I/O

If I’m P0
- Write global grid to visualization file

For -- (runs 3 iterations)
- MPI_Alltoall within local MPI group
- If I’m not P0
  - MPI_Send my data to P0
- Else
  - Write my data to visualization file
  - MPI_Recv data from P1 to P511 and write to visualization file

Cluster #1 (P4 - P60) in the same local group as P0

Symptom of serialization

Cause of serialization
Conclusion of PFLOTRAN

Serialized I/O is causing performance loss
- Automated analysis estimates run time improvement
  178s → 66s
- Replace serial I/O with parallel I/O
  178s → 70s

Presentation of automated insights of PFLOTRAN
- Reduces analysis complexity of time series in three dimensions
  - Process (group into clusters), Time (split into segments), Depth (automatic selection)
- Directs attention to potential performance losses
- Helps user understand the causes of such losses
Summary

Automated analysis of parallel time-series performance data
- Identifies potential inefficiencies in a large set of time series
  - Automation will be critical for analyzing performance on emerging exascale systems
  - Replace hours/days of manual effort with automated analysis

Future work
- Visualize summarized iterative behaviors over time
- Use semantic information for 1) MPMD applications; 2) more accurate diagnosis
- Provide automated hints on how to fix highlighted performance losses