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	P0	P1	P2	P3
main()	9s	9s	9s	9s
→ init()	1s	1s	1s	1s
→ solve()	8s	8s	8s	8s
→ → compute()	4s	4.1s	3.9s	4s
→ → sync()	4s	3.9s	4.1s	4s

Profile

Breaks down application run time into sources of costs

Calling context	P0	P1	P2	P3
main()	9s	9s	9s	9s
init()	1s	1s	1s	1s
solve()	8s	8s	8s	8s
compute()	4s	4.1s	3.9s	4s
sync()	4s	3.9s	4.1s	4s

Profile

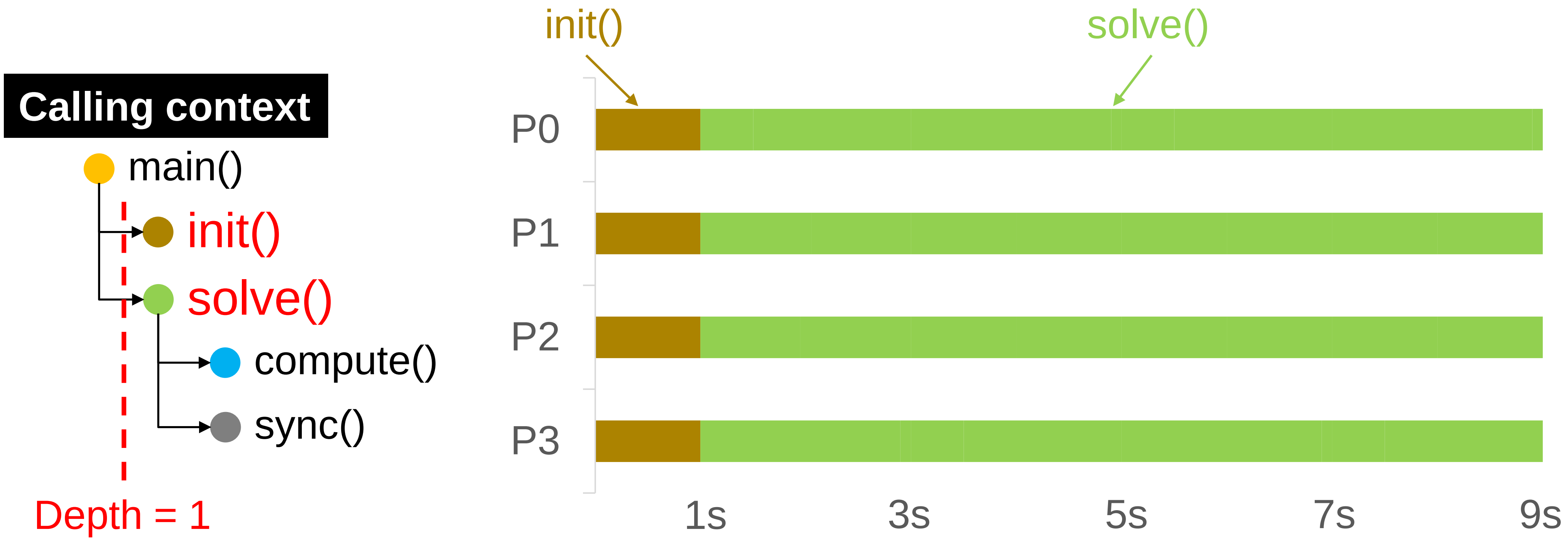
Breaks down application run time into sources of costs

Calling context	P0	P1	P2	P3
main()	9s	9s	9s	9s
→ init()	1s	1s	1s	1s
→ solve()	8s	8s	8s	8s
→ → compute()	4s	4.1s	3.9s	4s
→ → sync()	4s	3.9s	4.1s	4s

Performance loss, why?

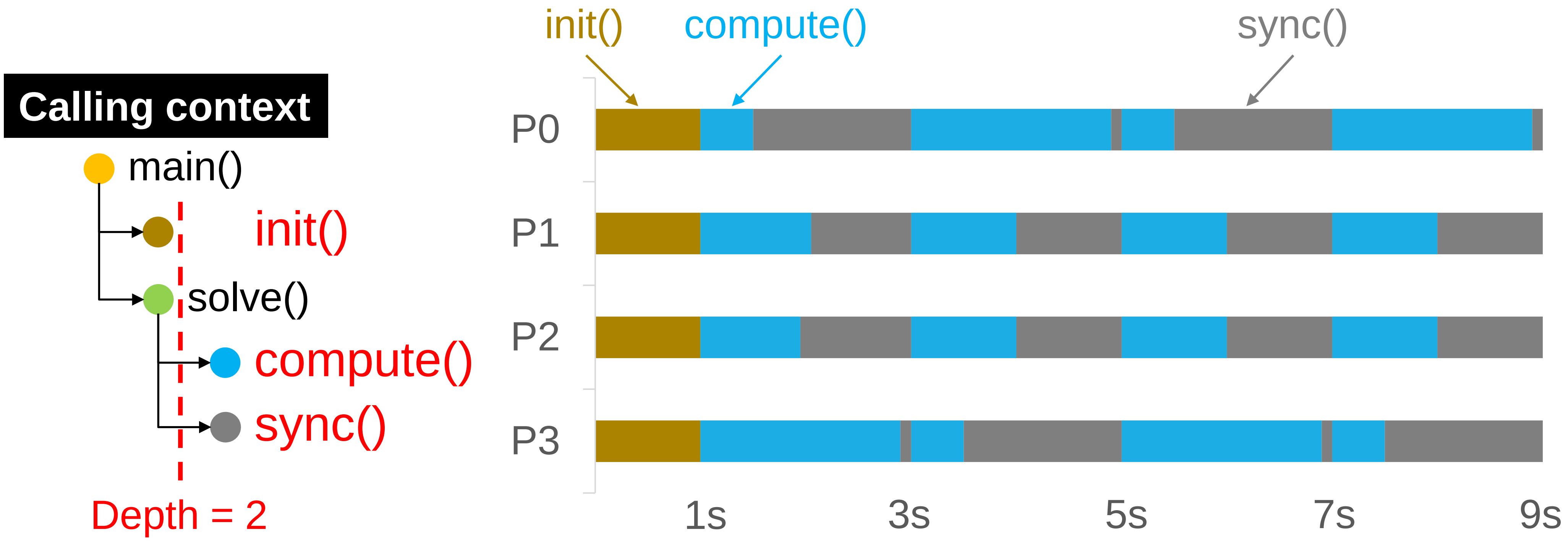
Time series

Presents application behavior over time



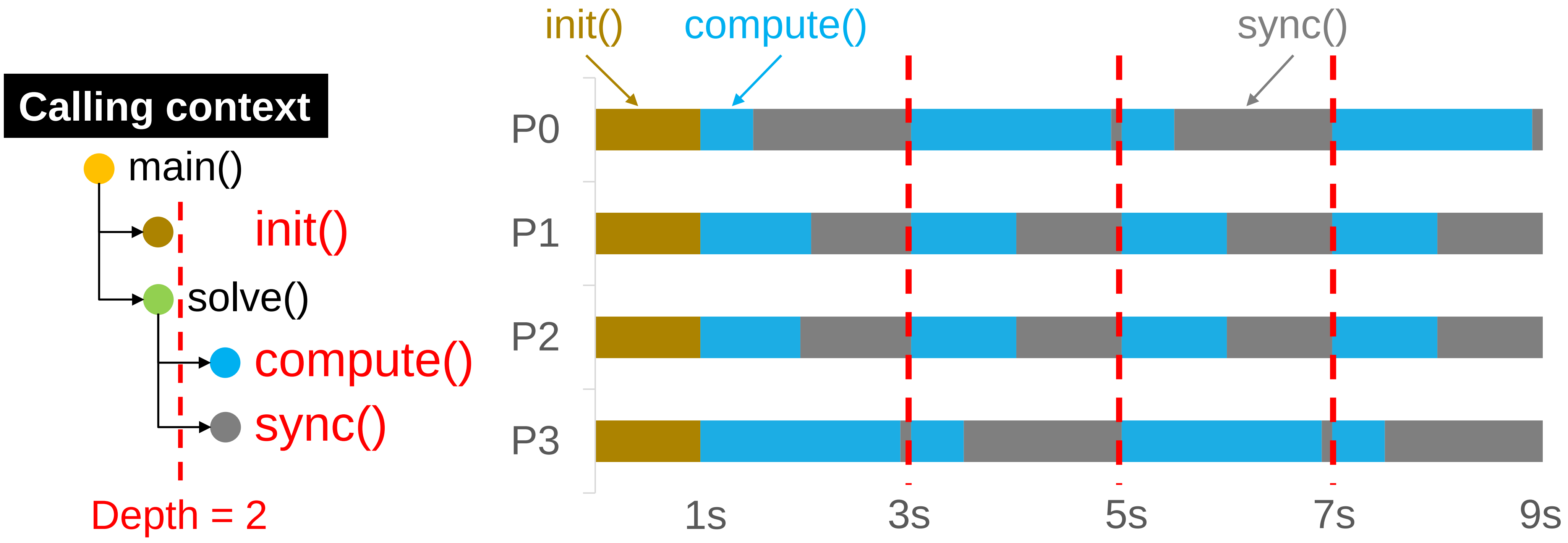
Time series

Presents application behavior over time



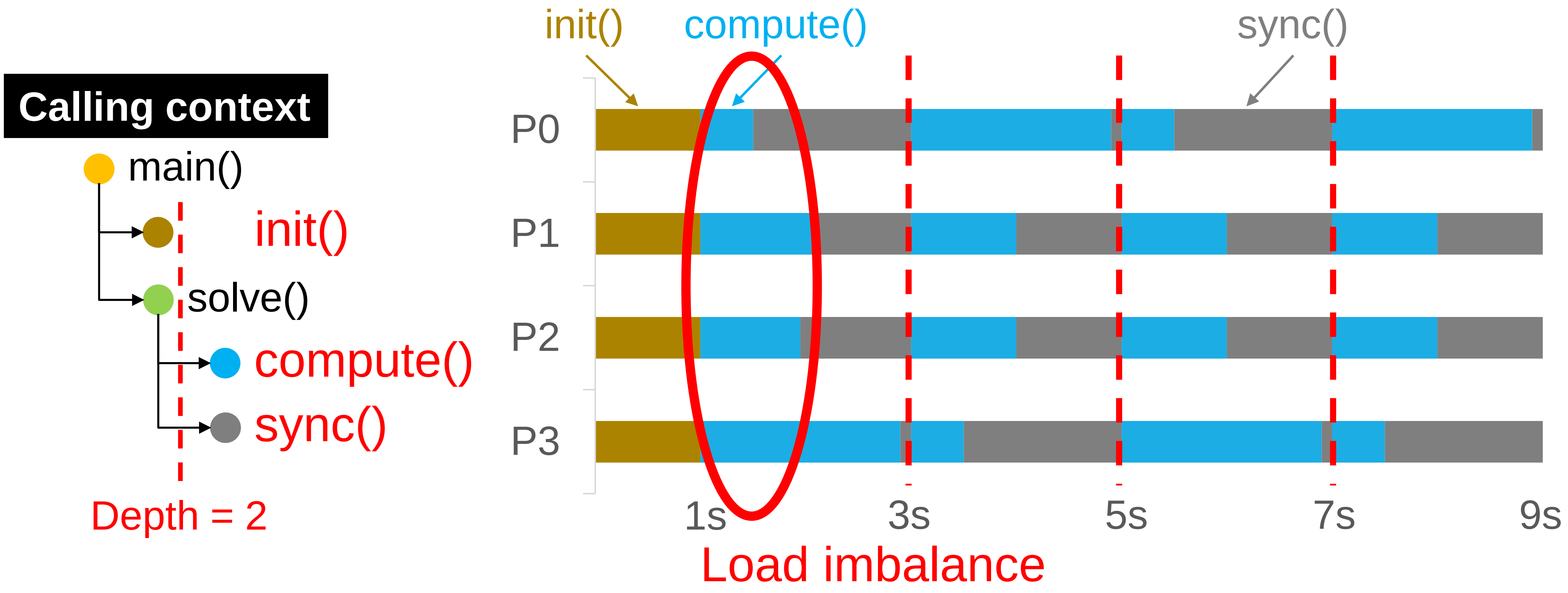
Time series

Presents application behavior over time



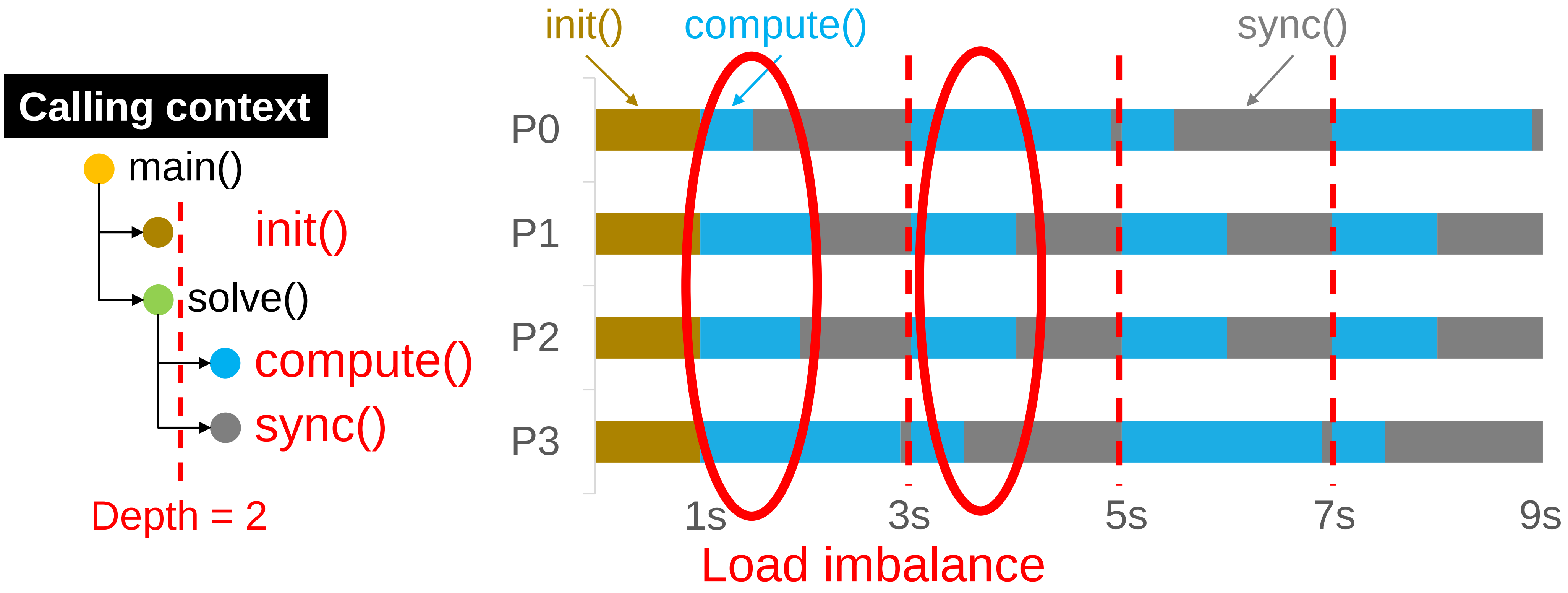
Time series

Presents application behavior over time



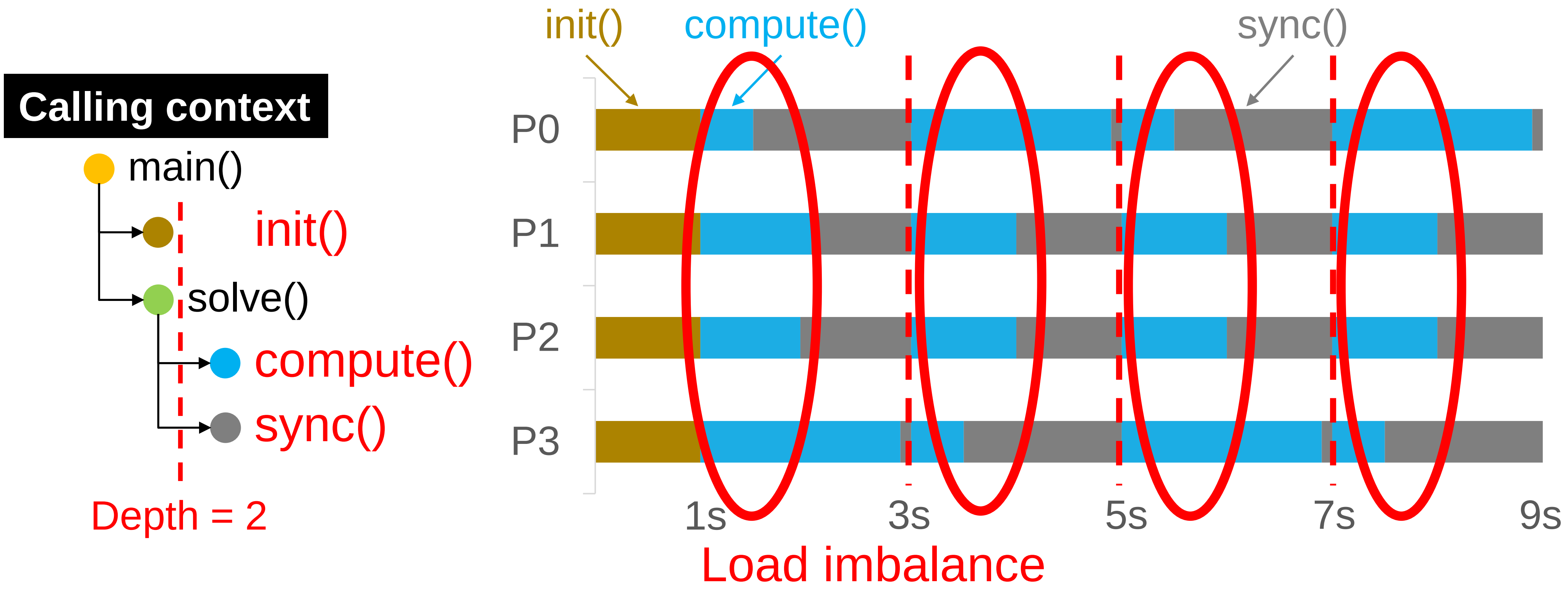
Time series

Presents application behavior over time



Time series

Presents application behavior over time



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

```
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

```
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

```
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

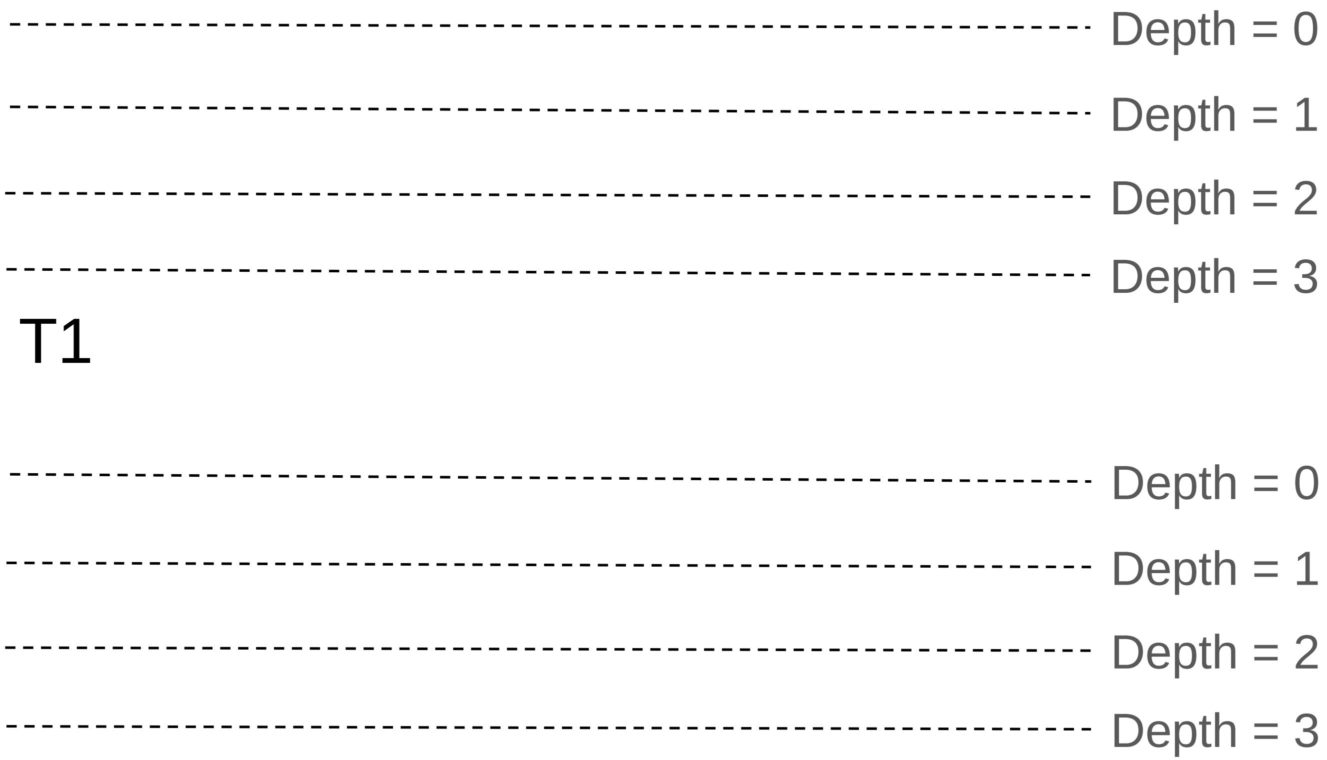
```
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
      C();
      for (int k = 0; k < 3; k++) {
            A();
            if (k==0) B();
            if (k==2) C();
      }
}
int main() {
      foo();
      return 0;
```

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

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

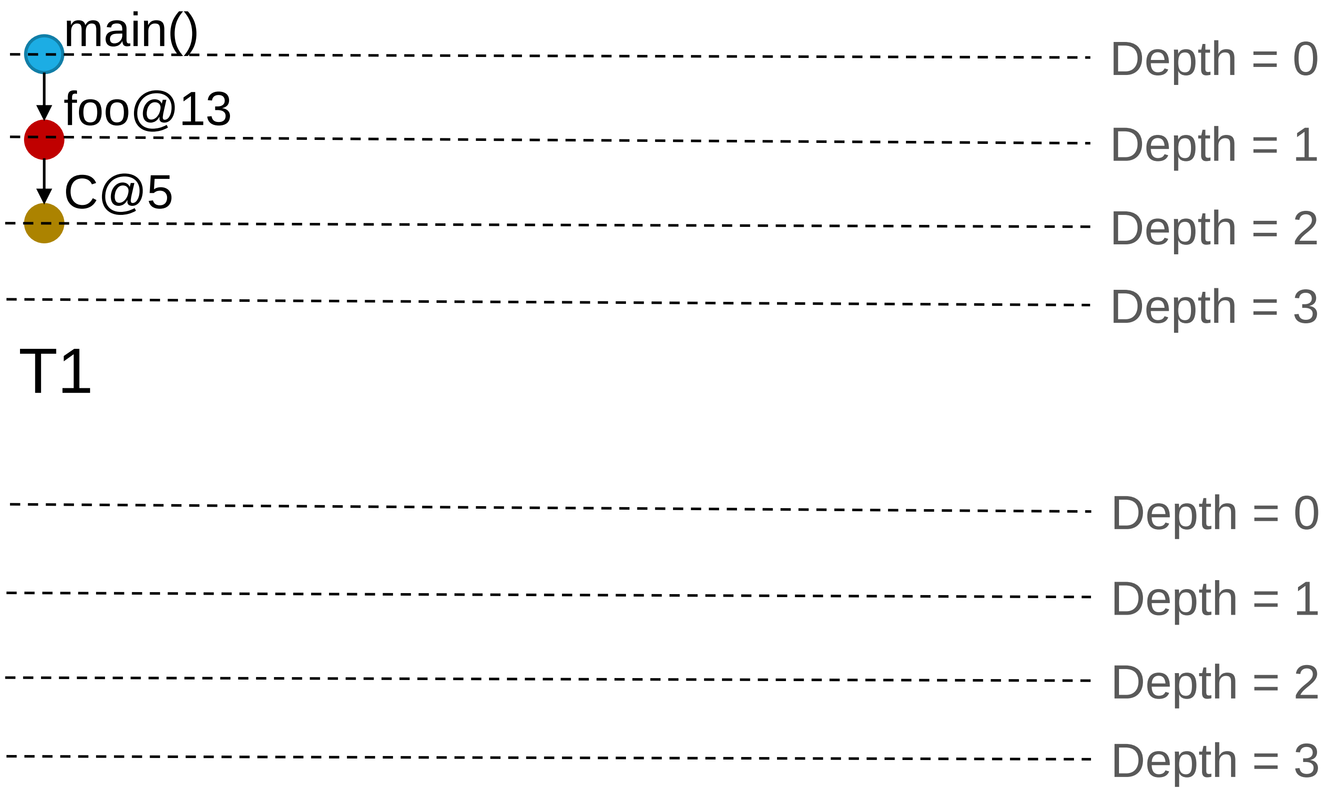
Collect call path samples over time

```
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;
```



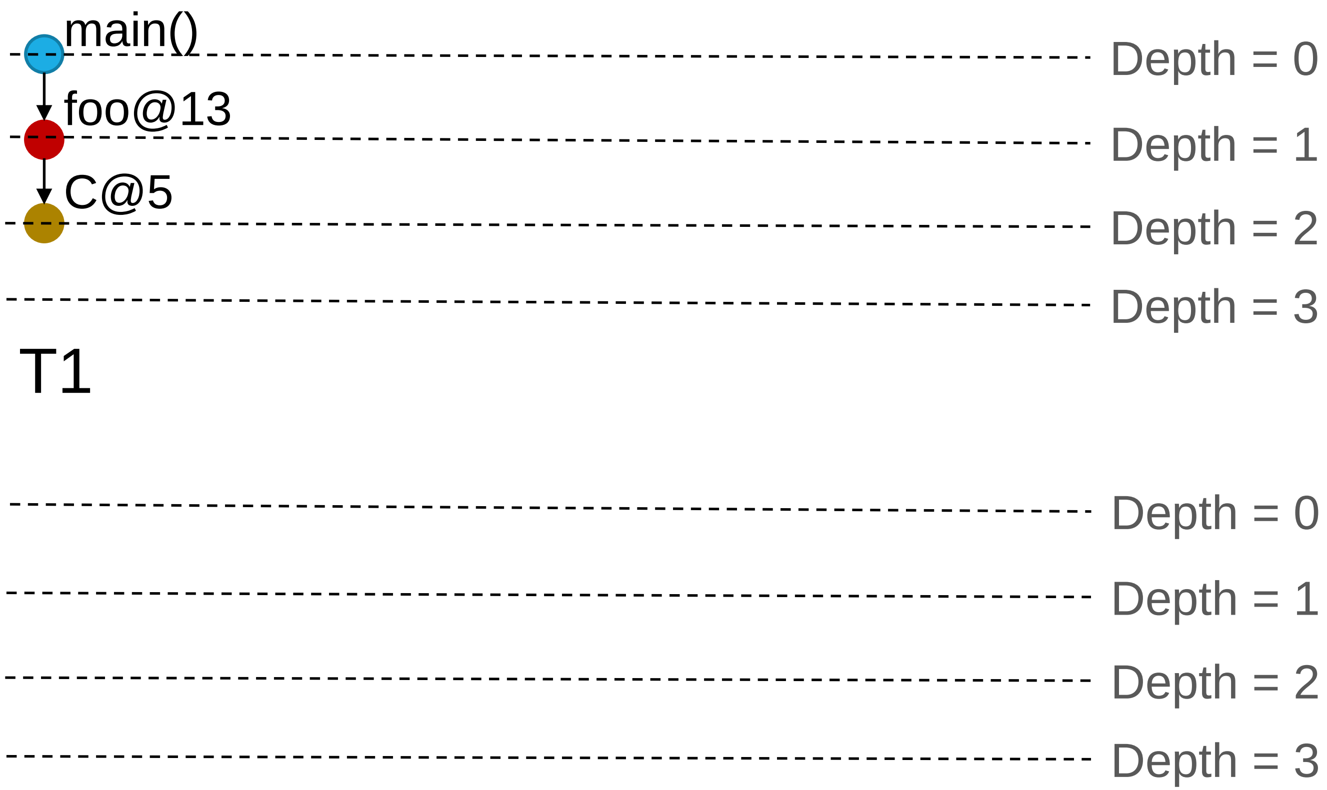
Collect call path samples over time

```
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;
```



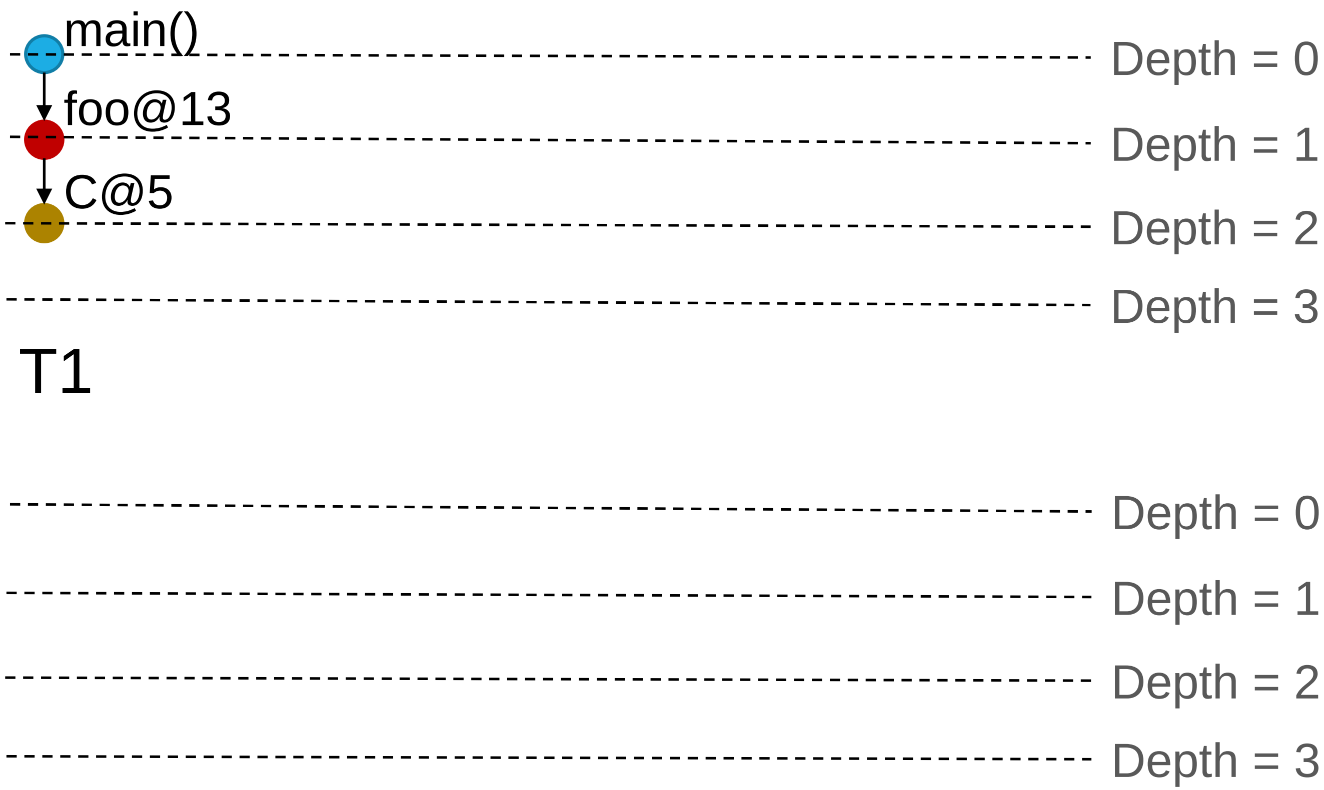
Collect call path samples over time

```
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;
```



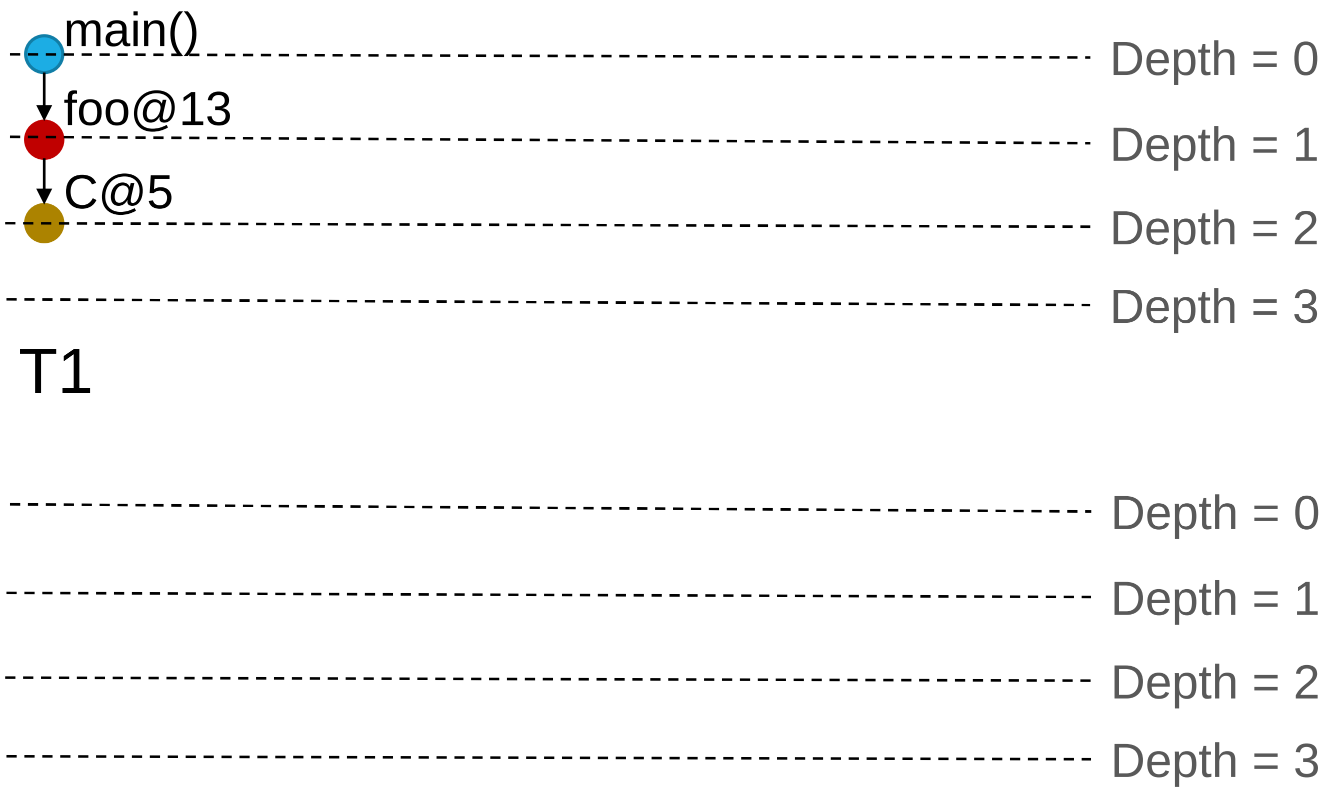
Collect call path samples over time

```
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;
```



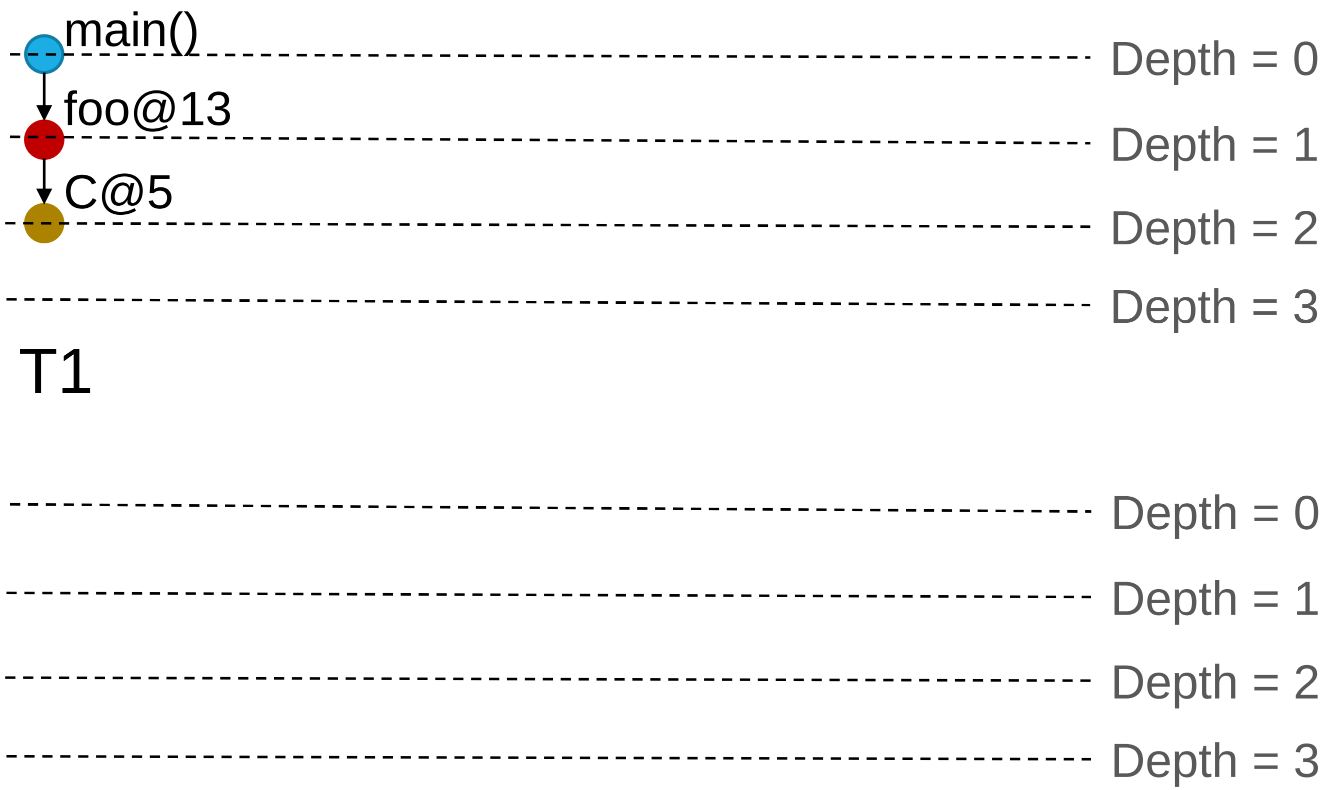
Collect call path samples over time

```
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;
```



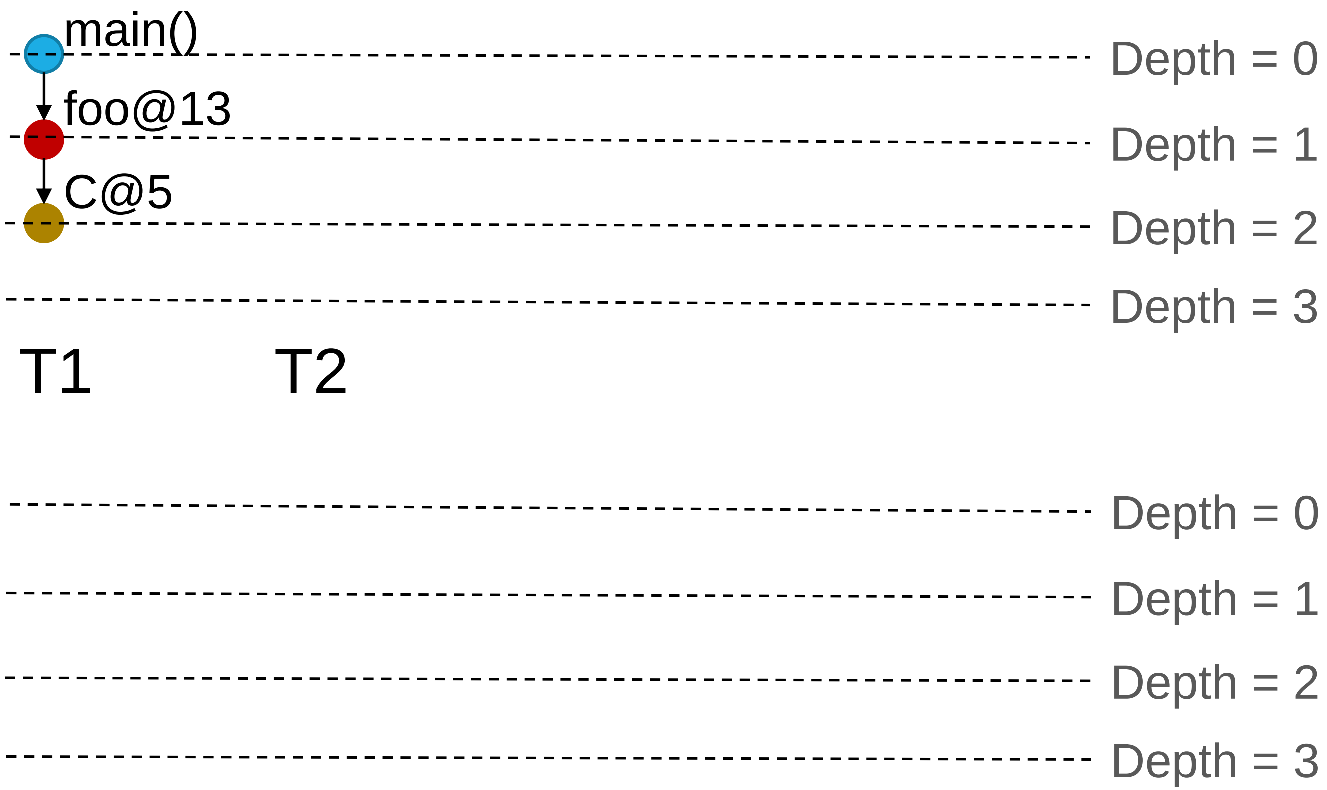
Collect call path samples over time

```
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

```
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

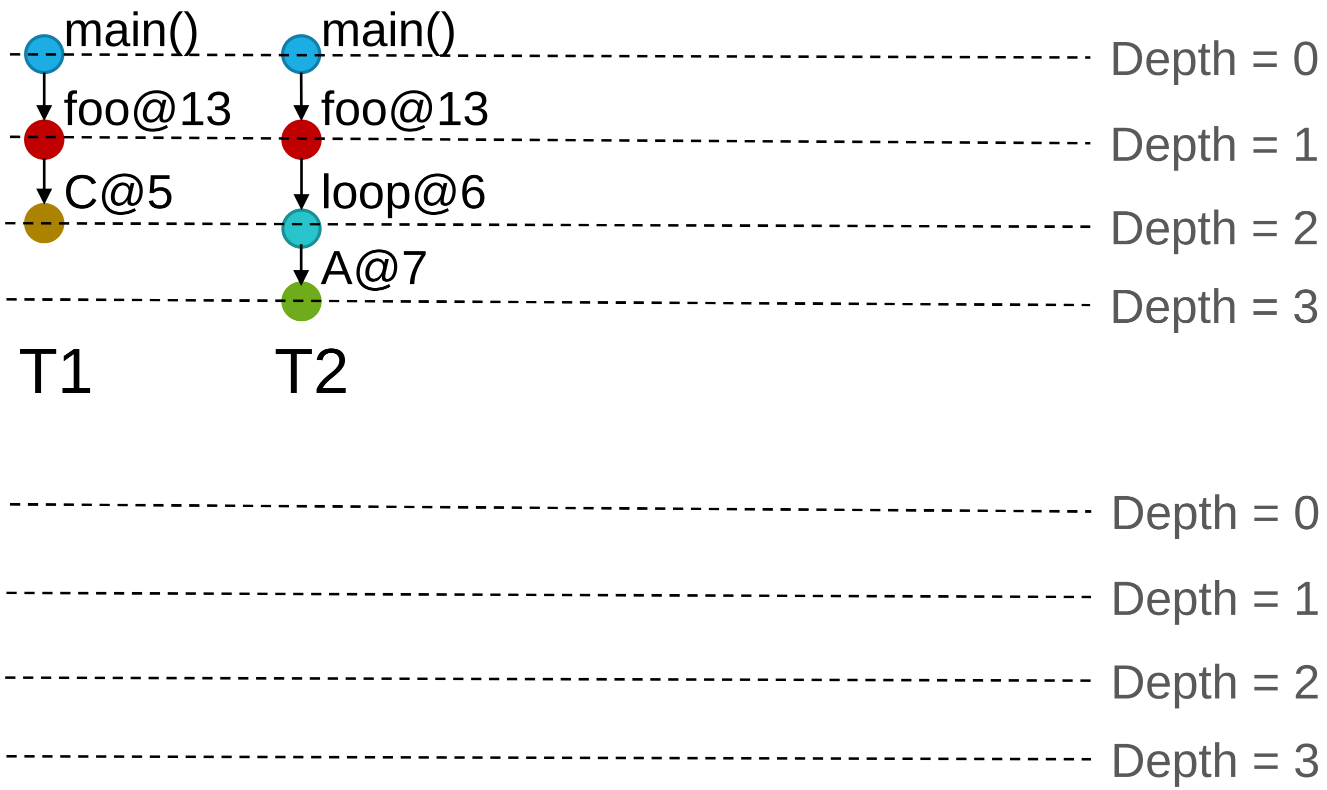
```
void A() { ... }
void B() { ... }
void C() { ... }
void foo() {
    C();
    for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
    }
}
int main() {
    foo();
    return 0;
```

Collect call path samples over time

```
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;
```

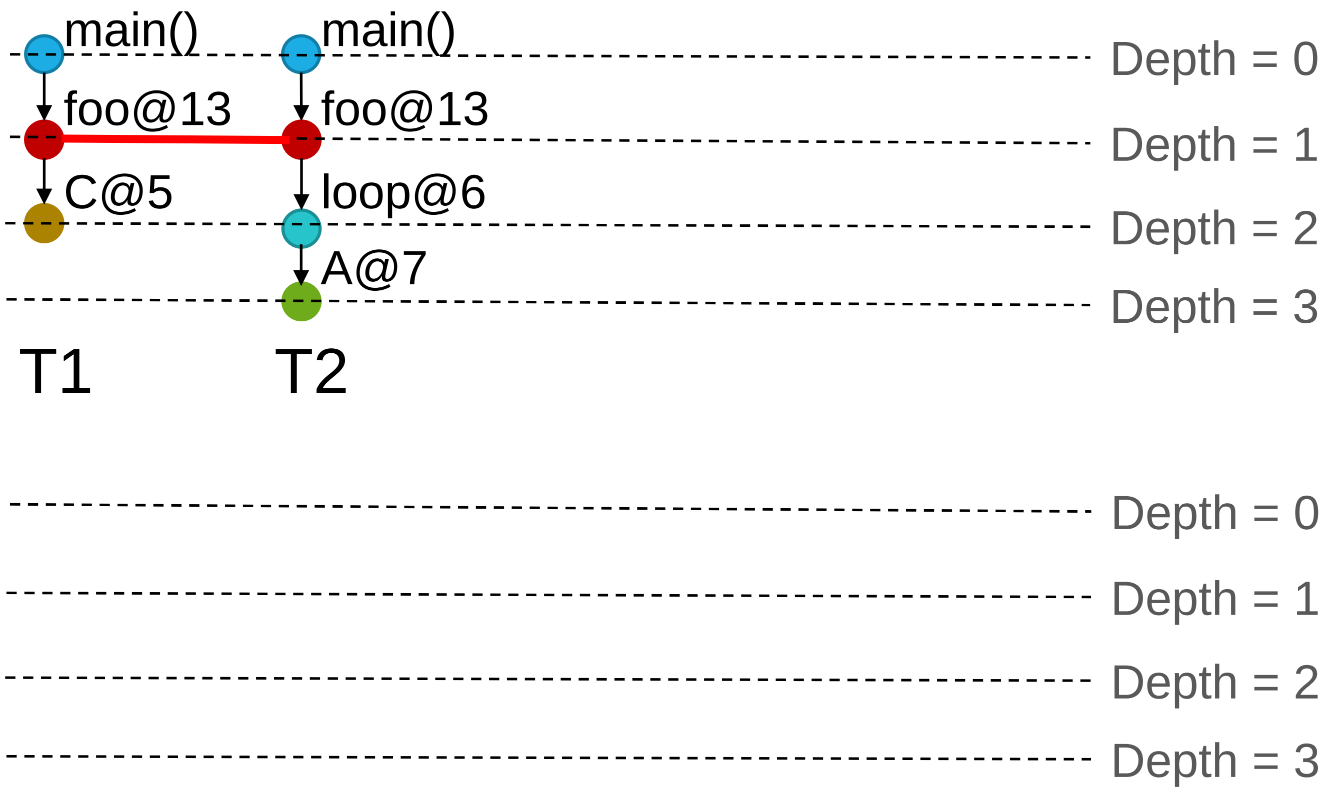
Collect call path samples over time

```
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;
```



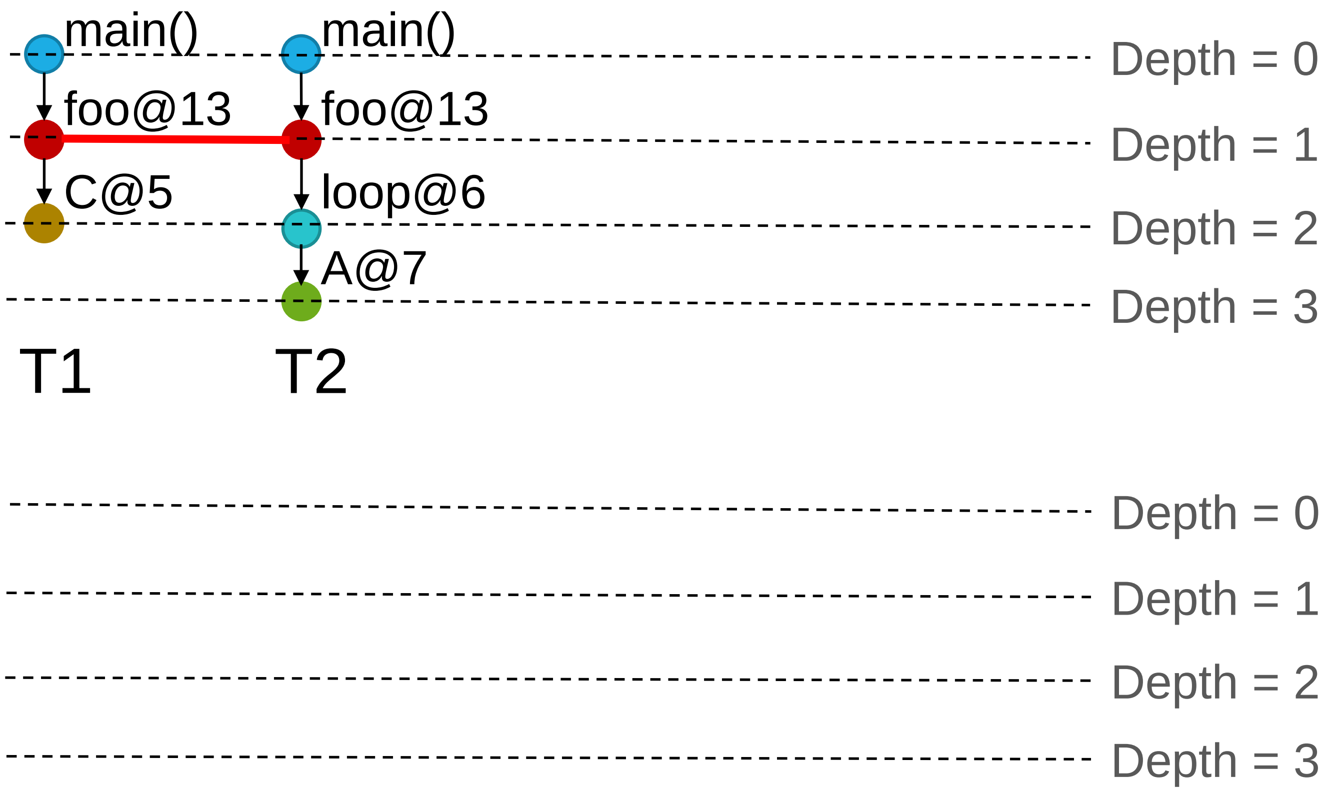
Collect call path samples over time

```
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;
```



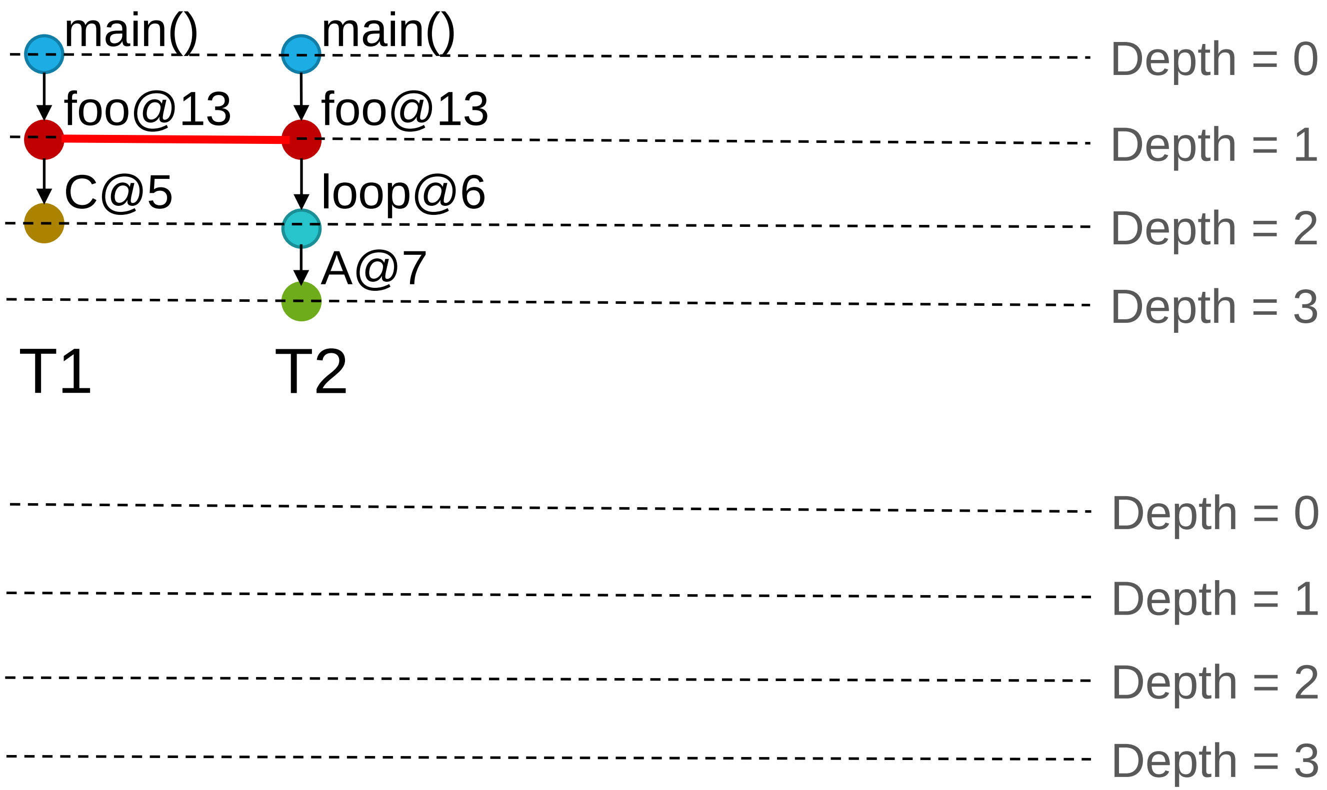
Collect call path samples over time

```
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;
```



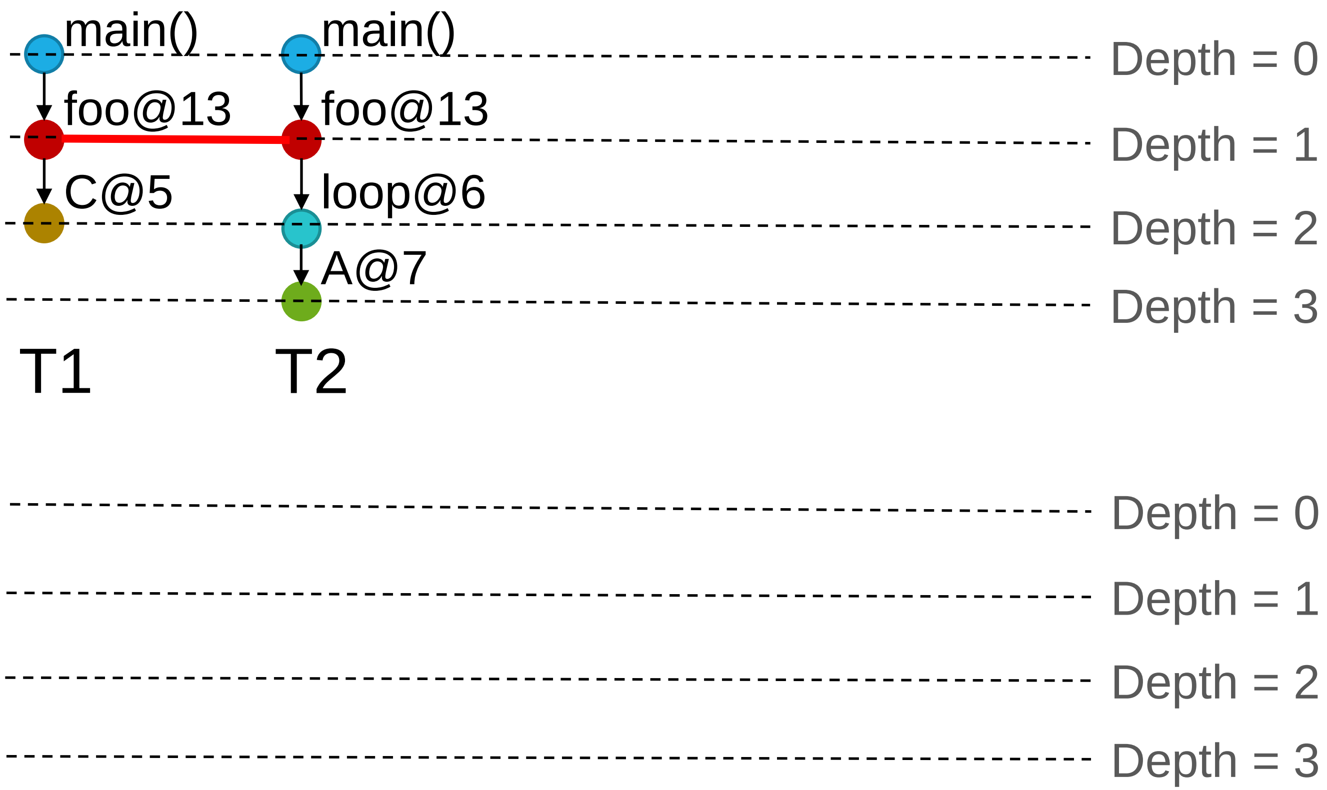
Collect call path samples over time

```
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;
```



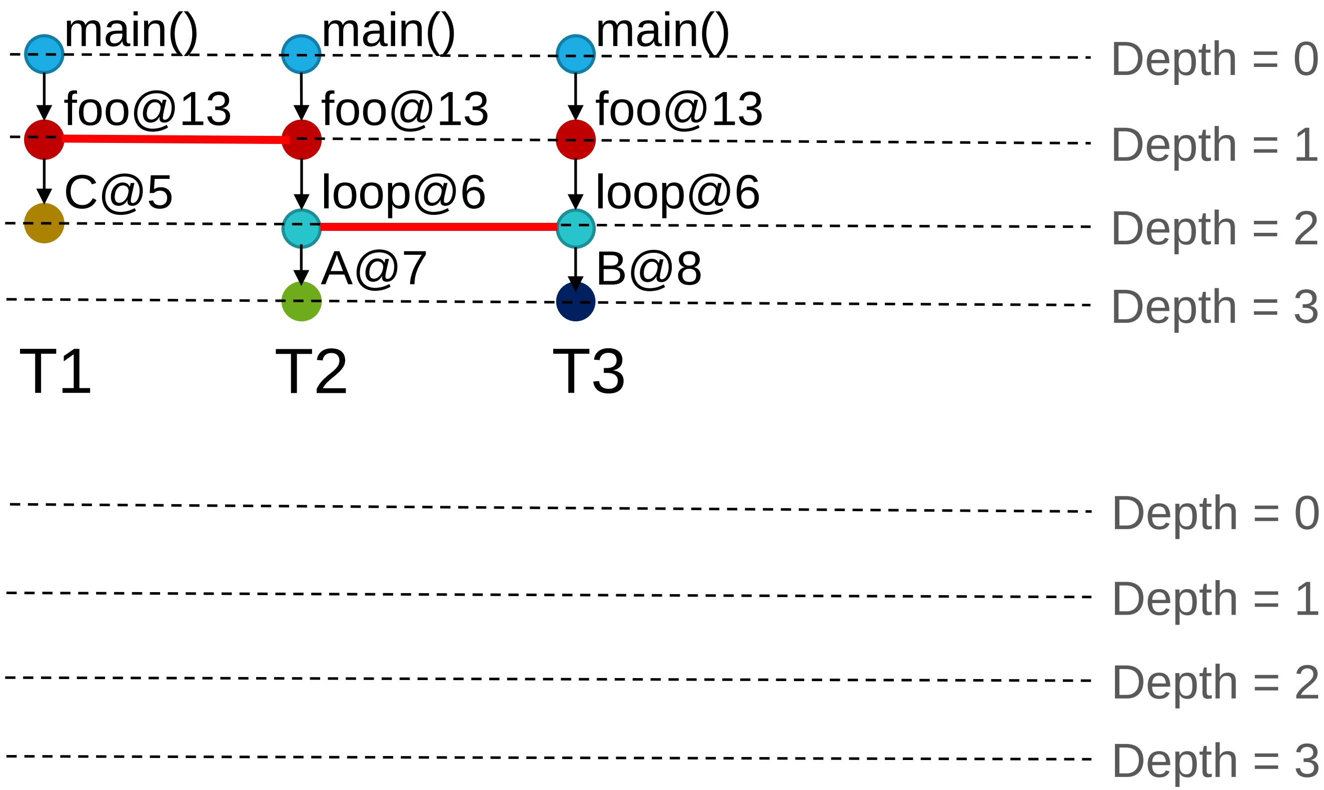
Collect call path samples over time

```
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;
```



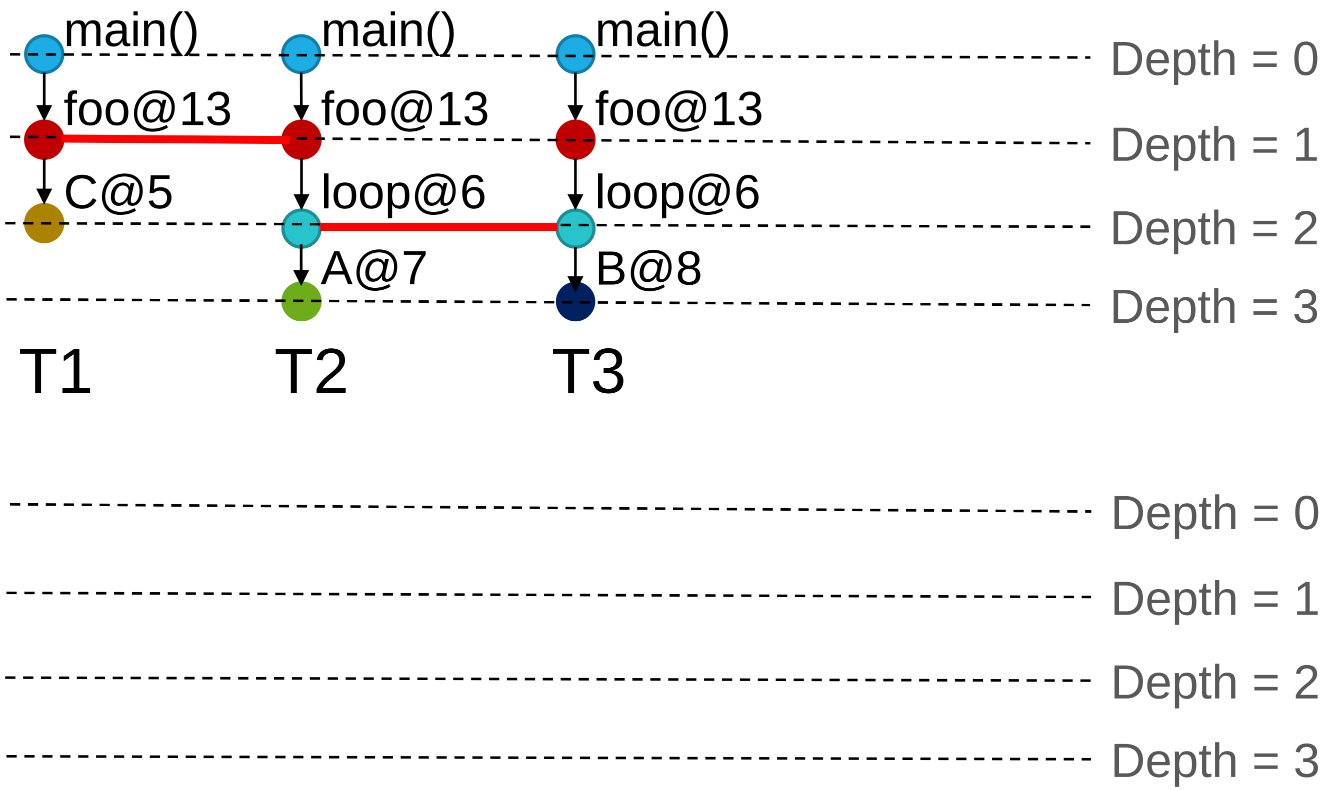
Collect call path samples over time

```
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;
```



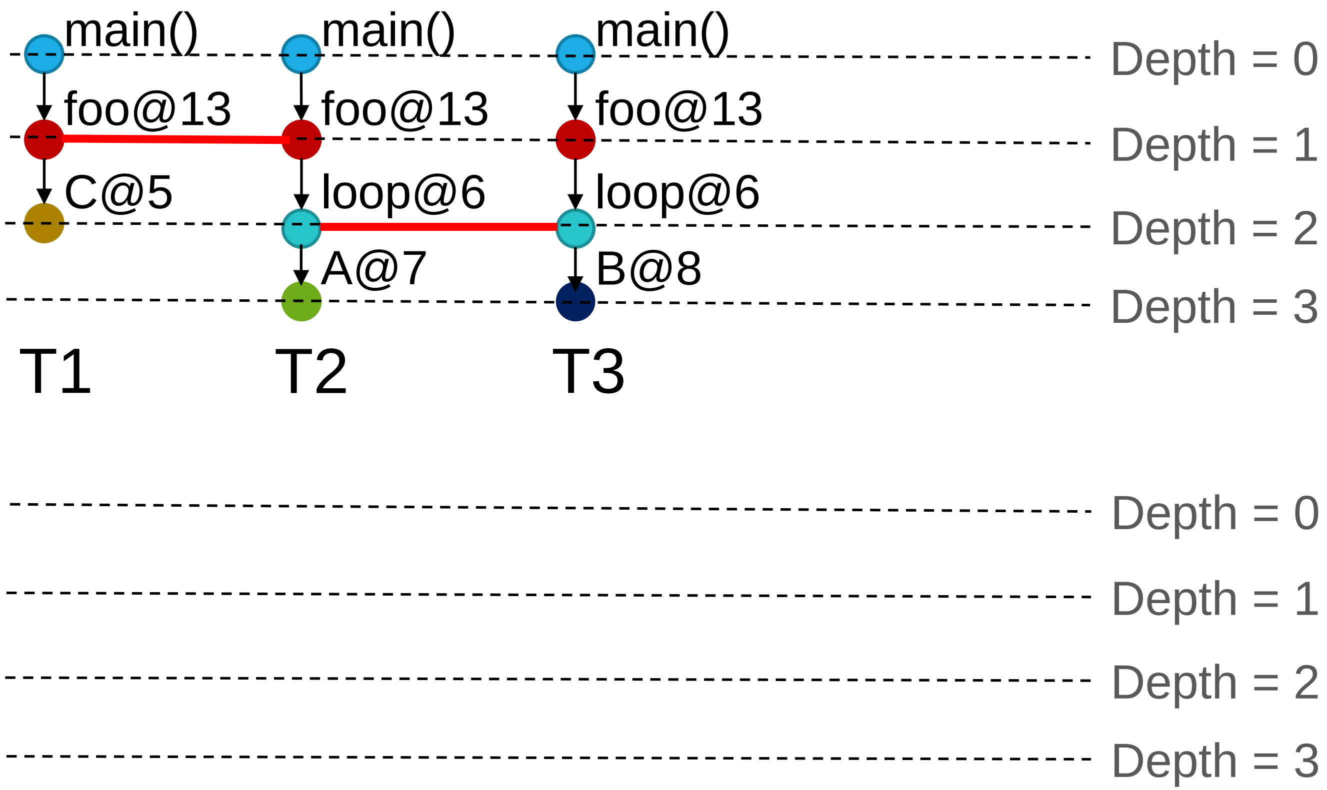
Collect call path samples over time

```
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;
```



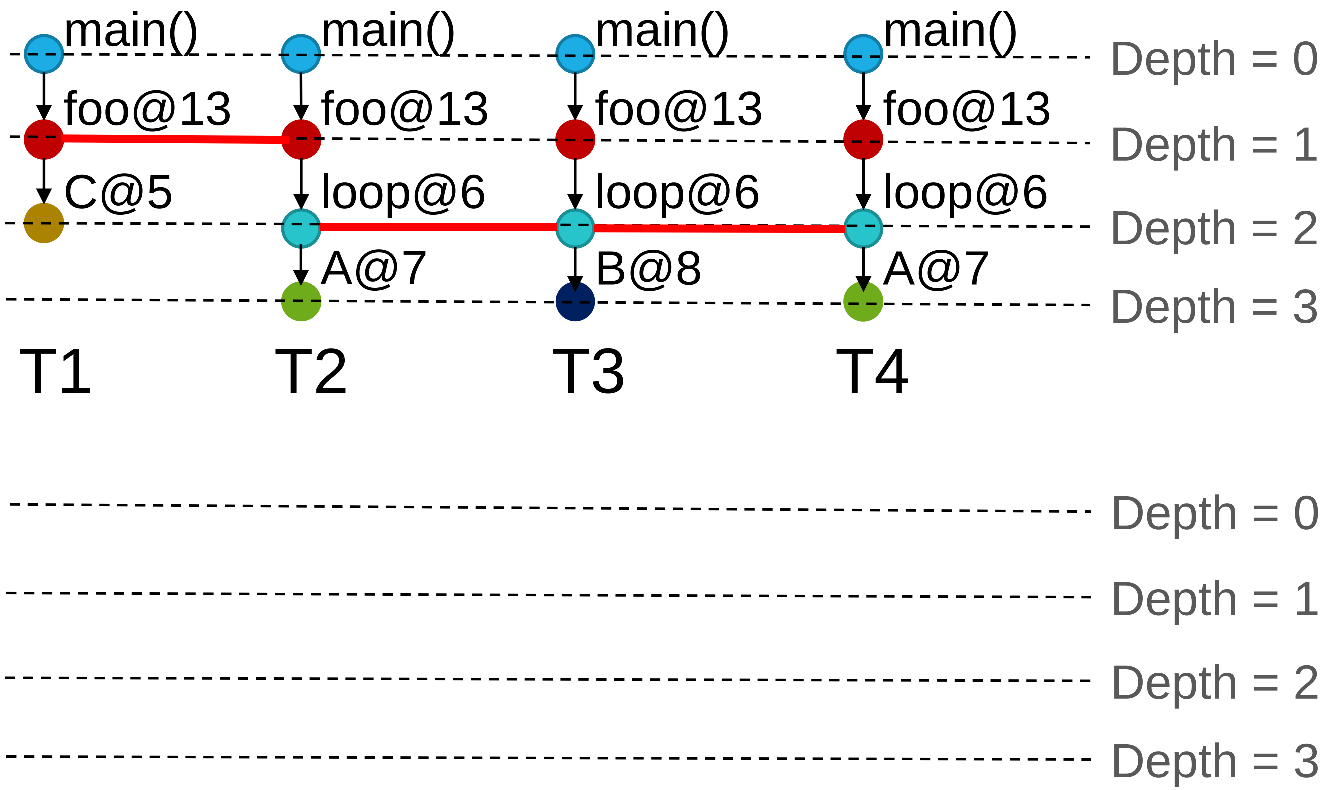
Collect call path samples over time

```
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;
```



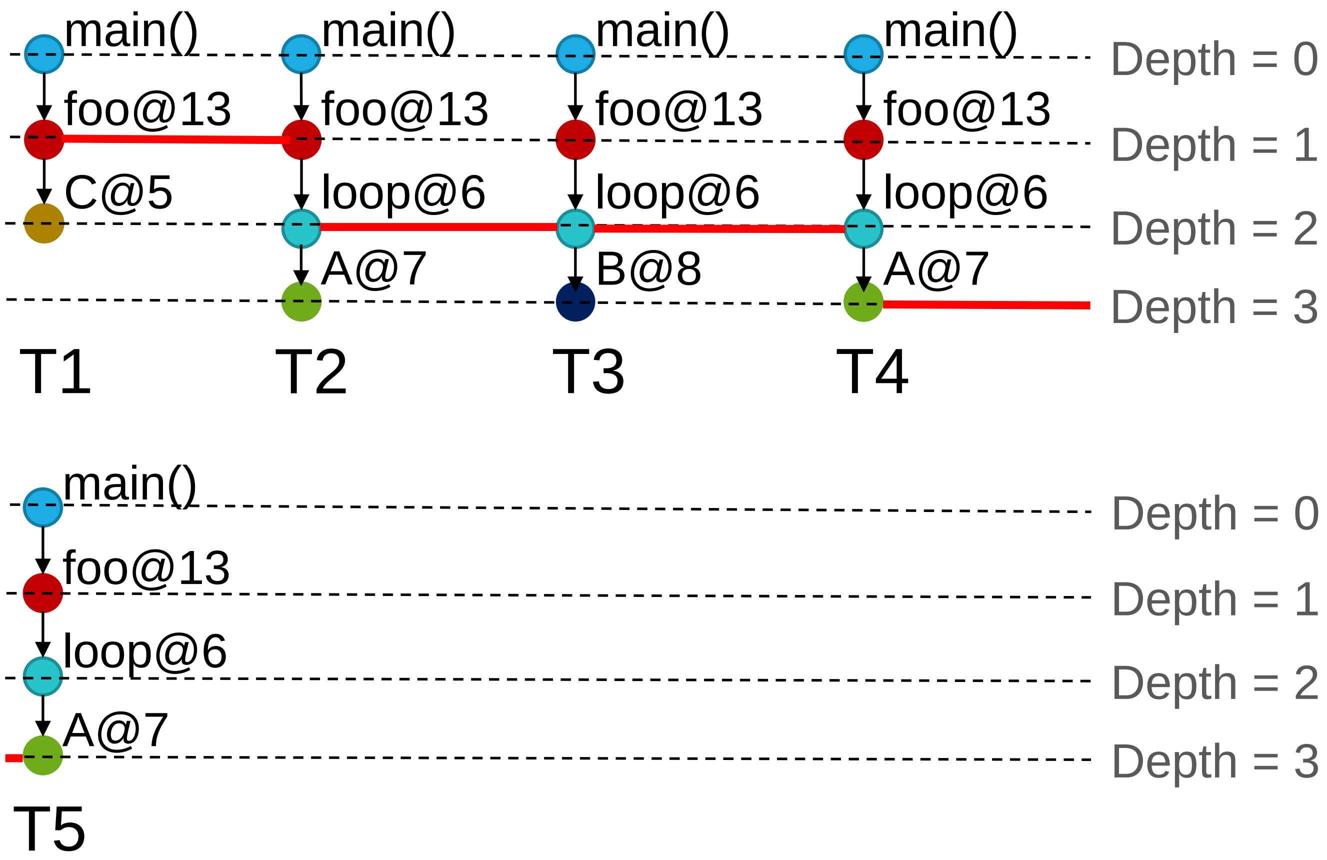
Collect call path samples over time

```
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;
```



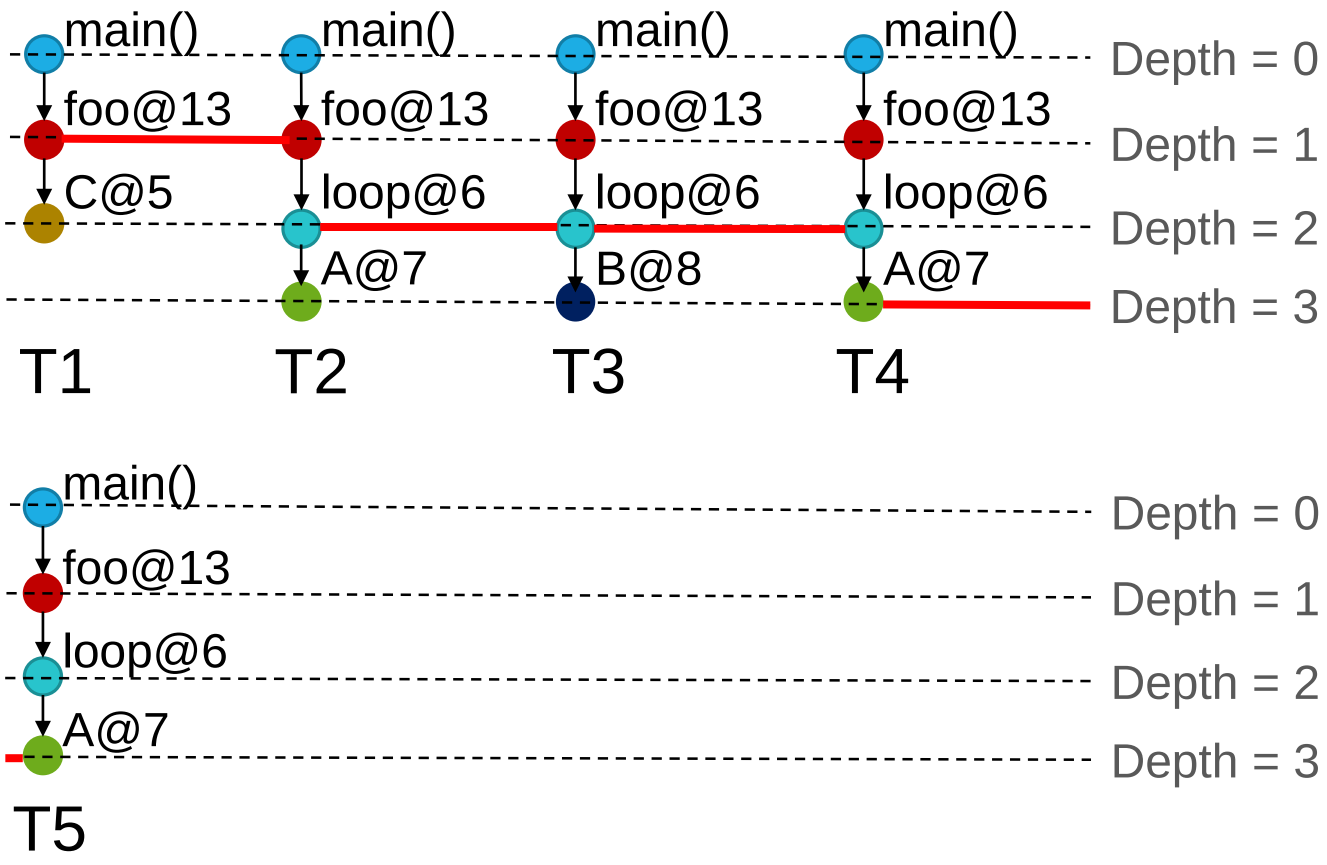
Collect call path samples over time

```
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;
```



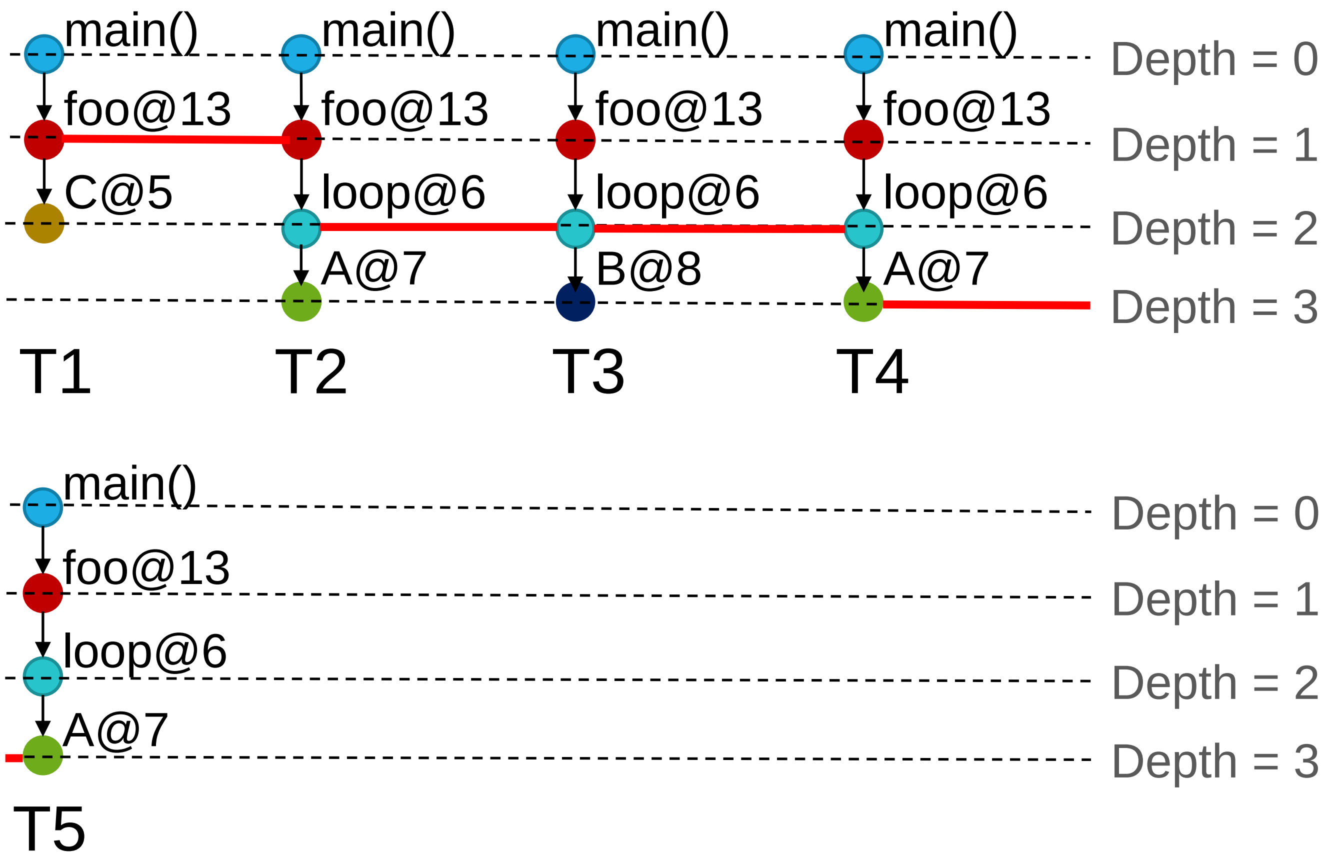
Collect call path samples over time

```
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;
```



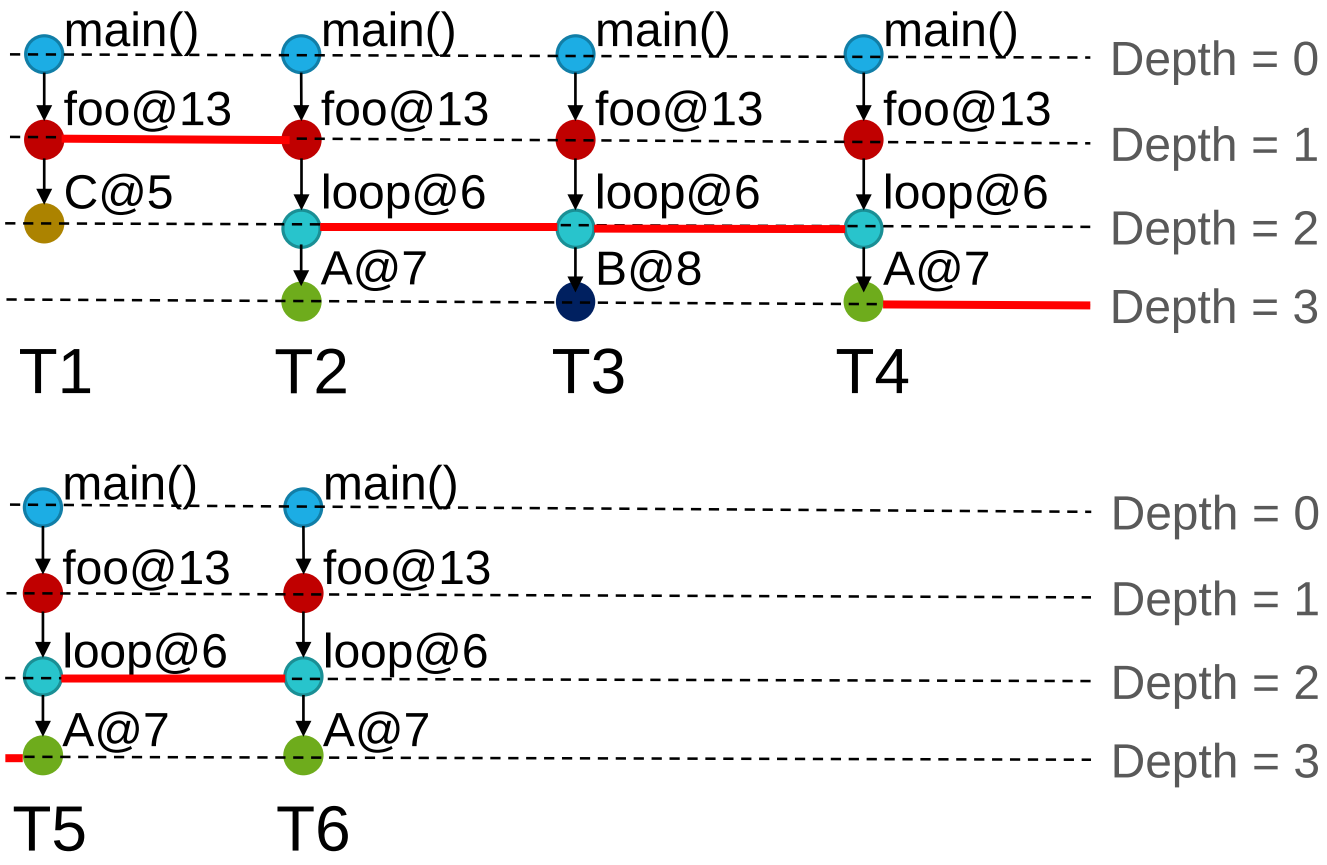
Collect call path samples over time

```
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;
```



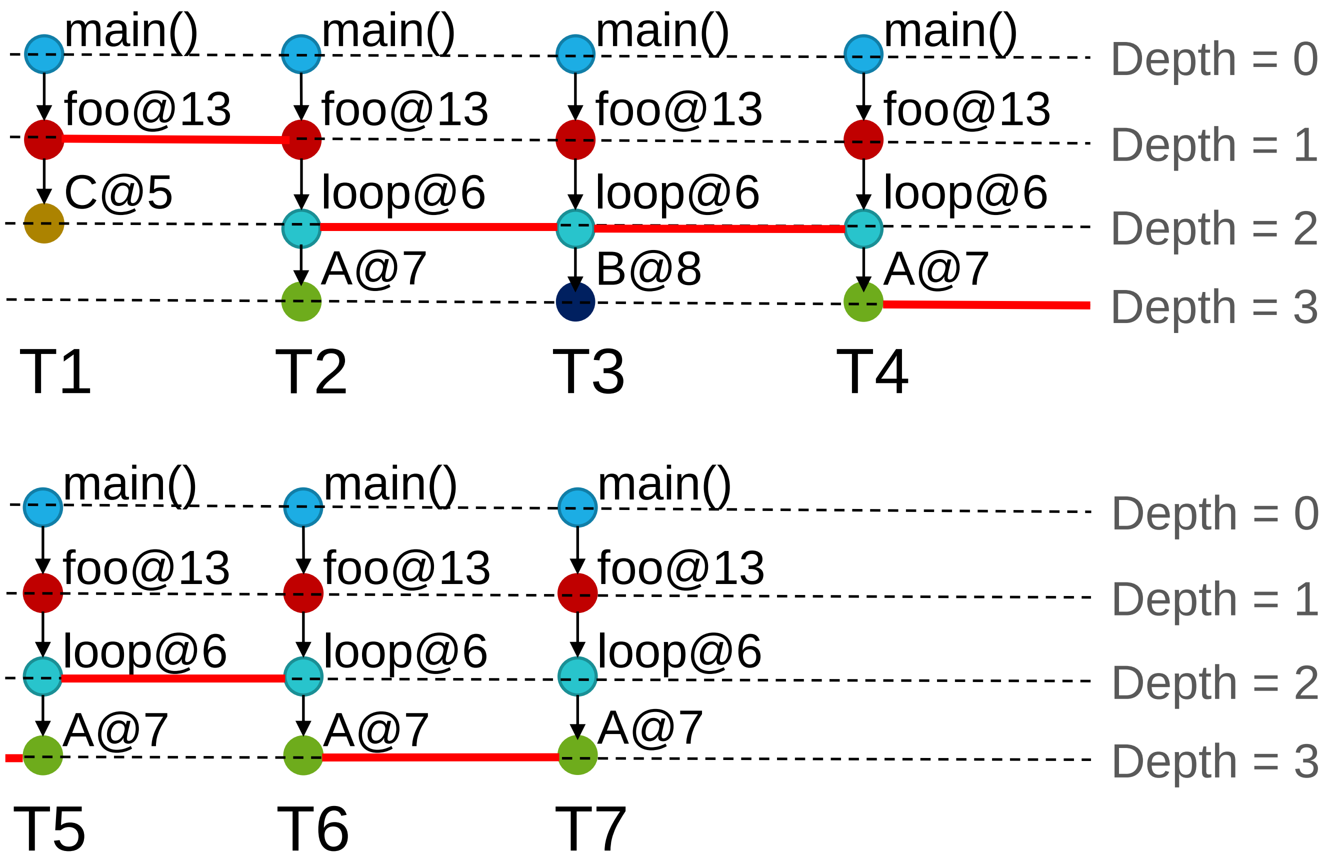
Collect call path samples over time

```
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;
```



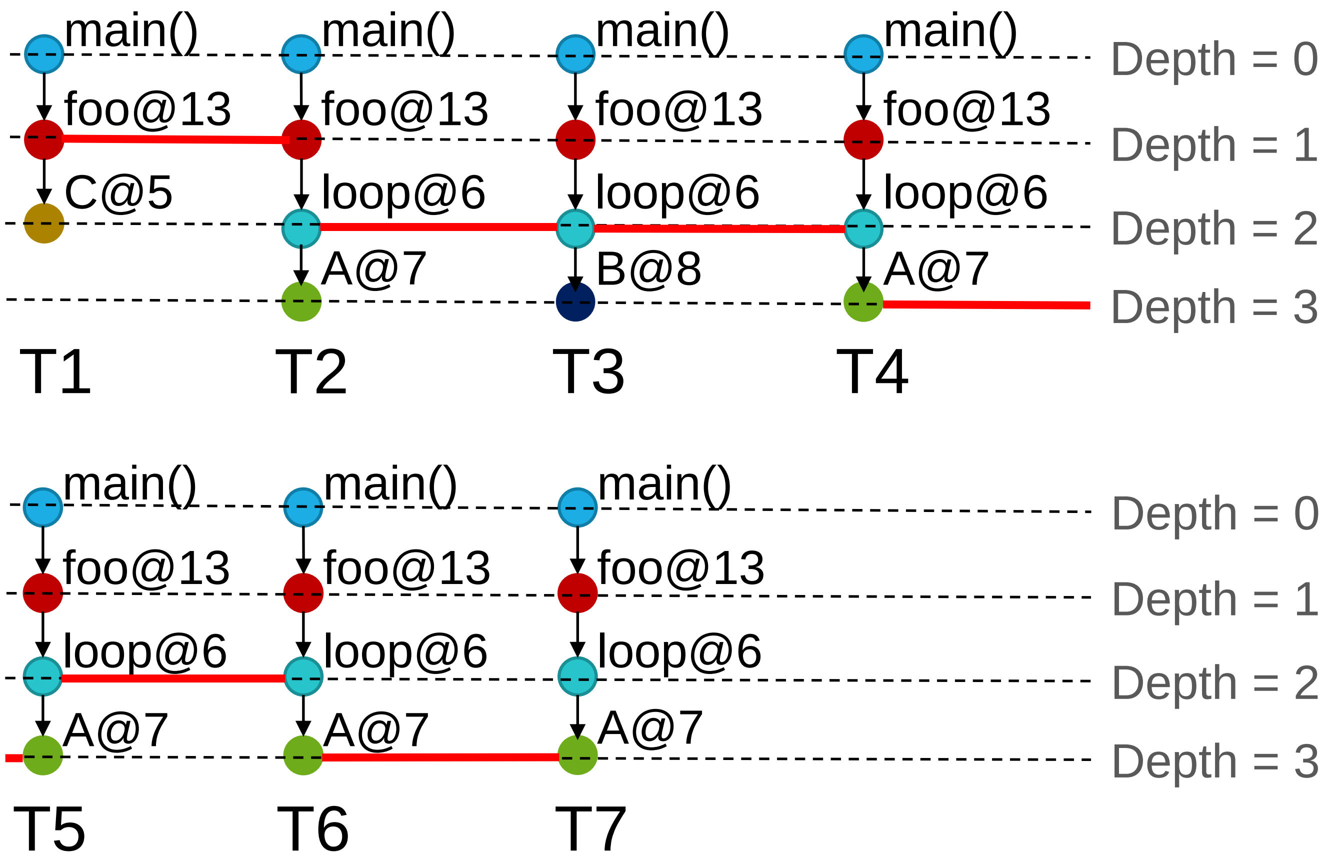
Collect call path samples over time

```
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;
```



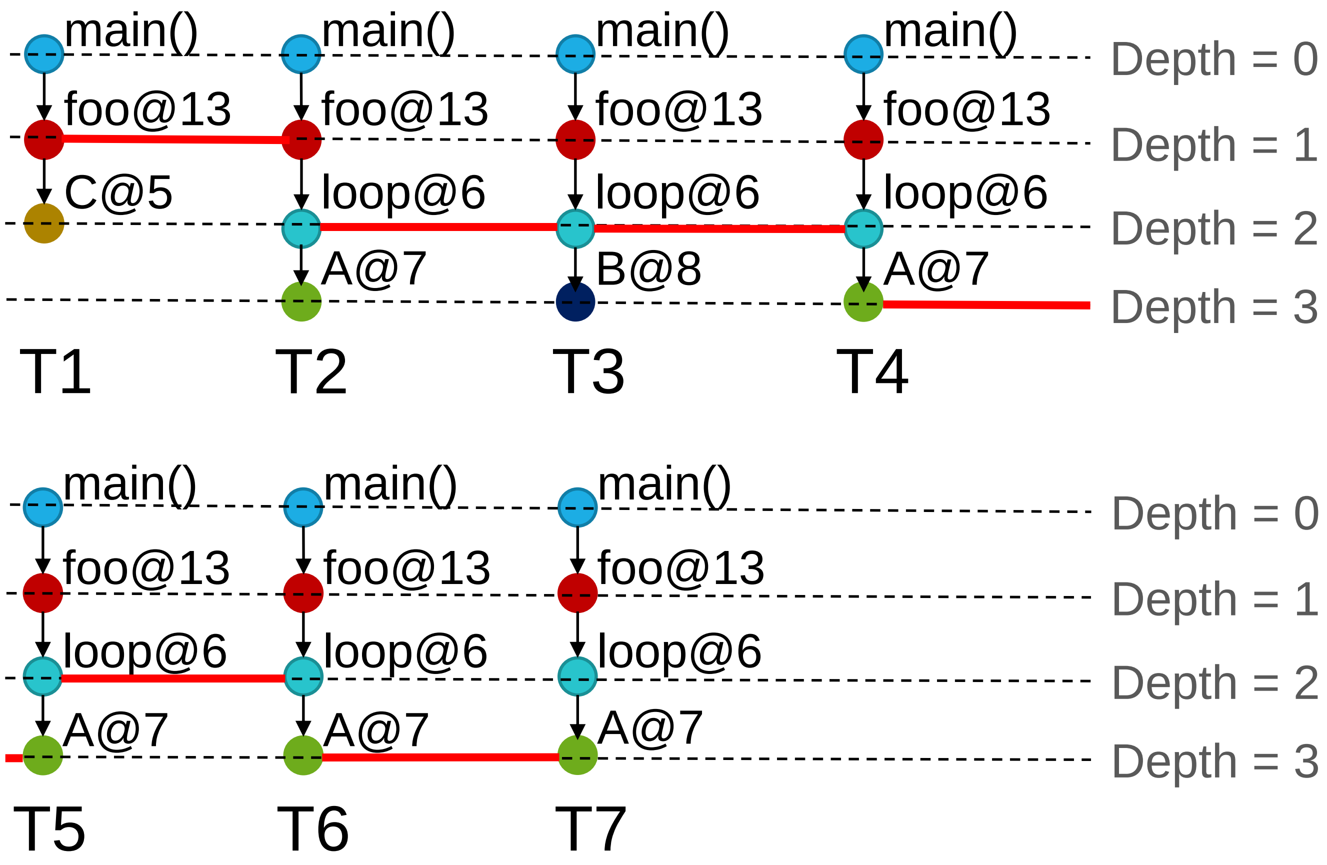
Collect call path samples over time

```
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;
```



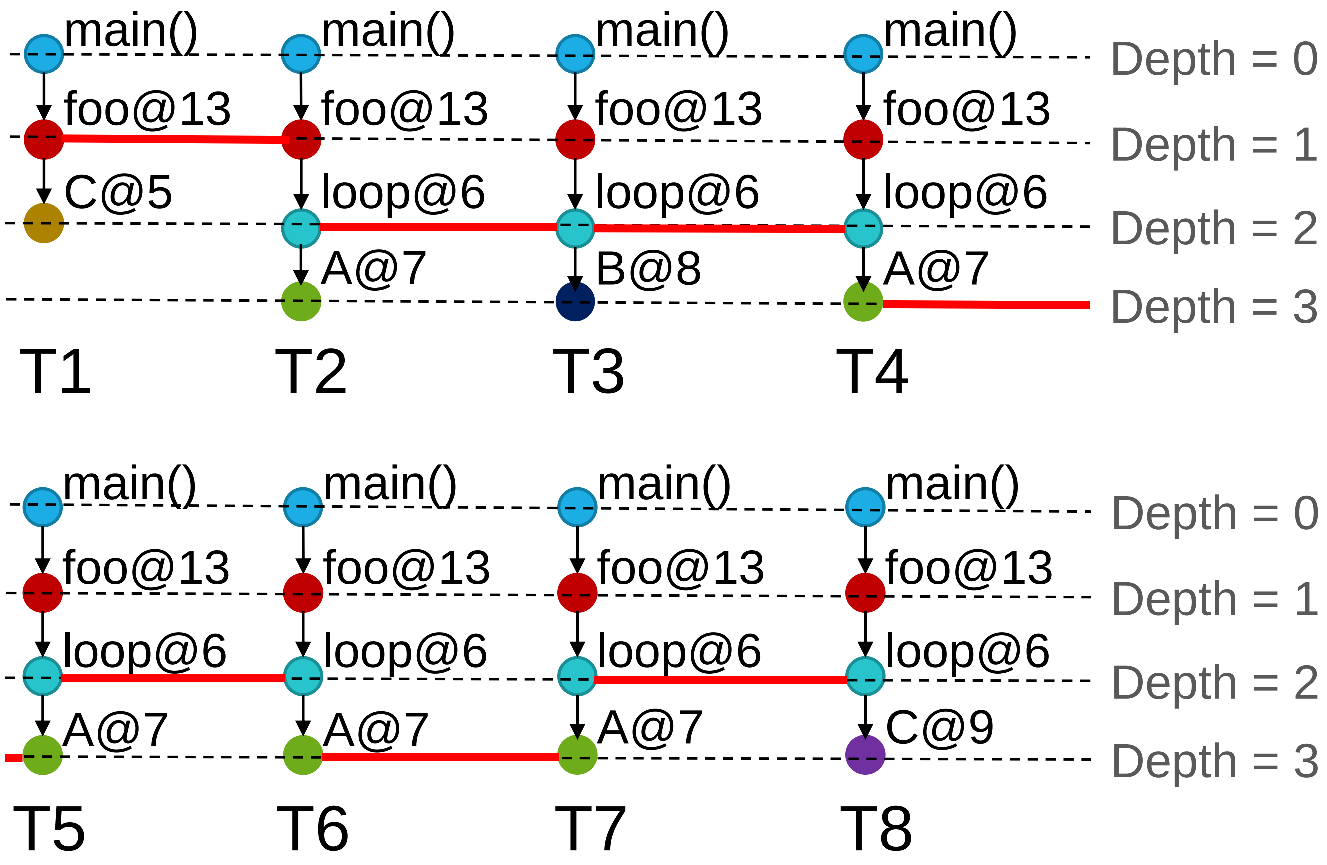
Collect call path samples over time

```
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;
```

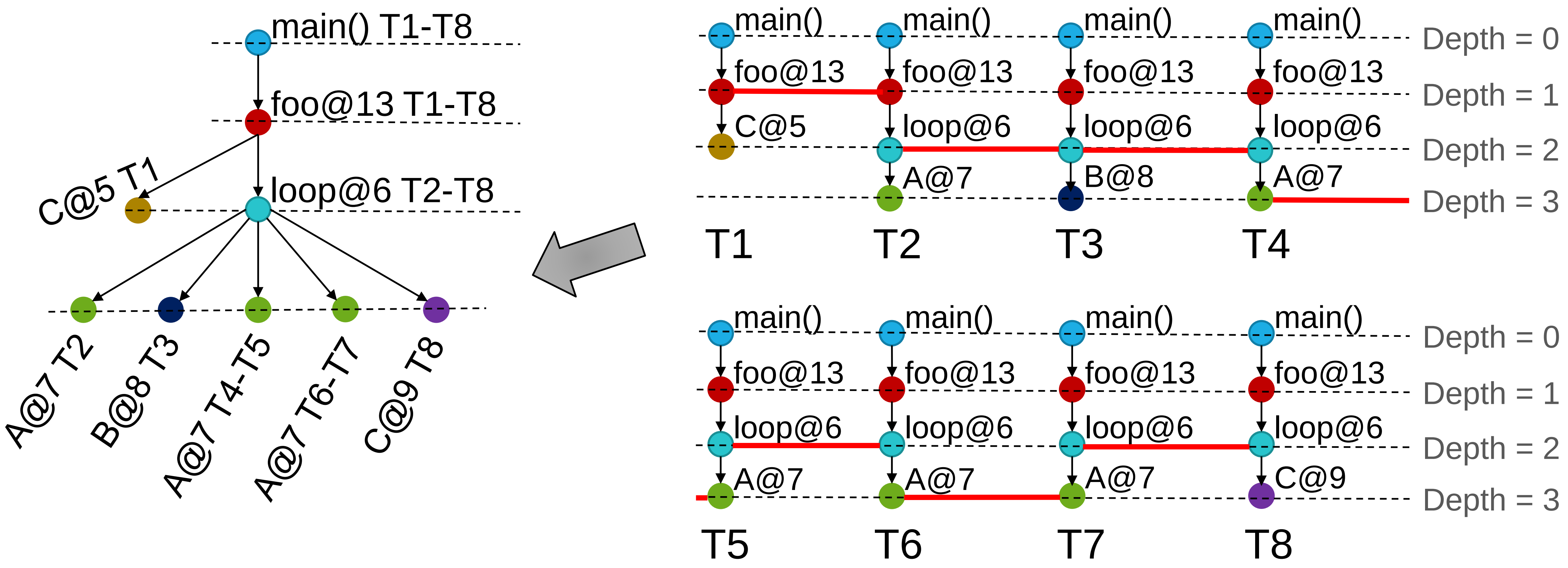


Collect call path samples over time

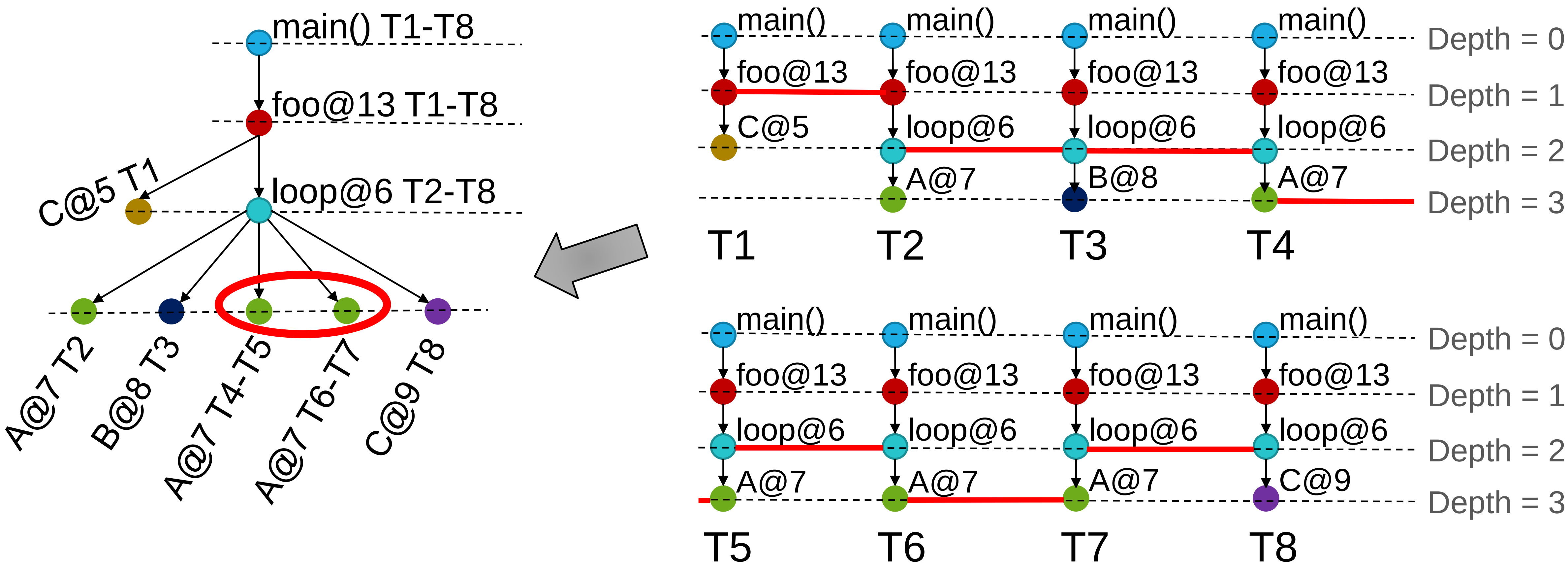
```
1  void A() { ... }
2  void B() { ... }
3  void C() { ... }
4  void foo() {
5      C();
6      for (int k = 0; k < 3; k++) {
7          A();
8          if (k==0) B();
9          if (k==2) C();
10     }
11 }
12 int main() {
13     foo();
14     return 0;
```



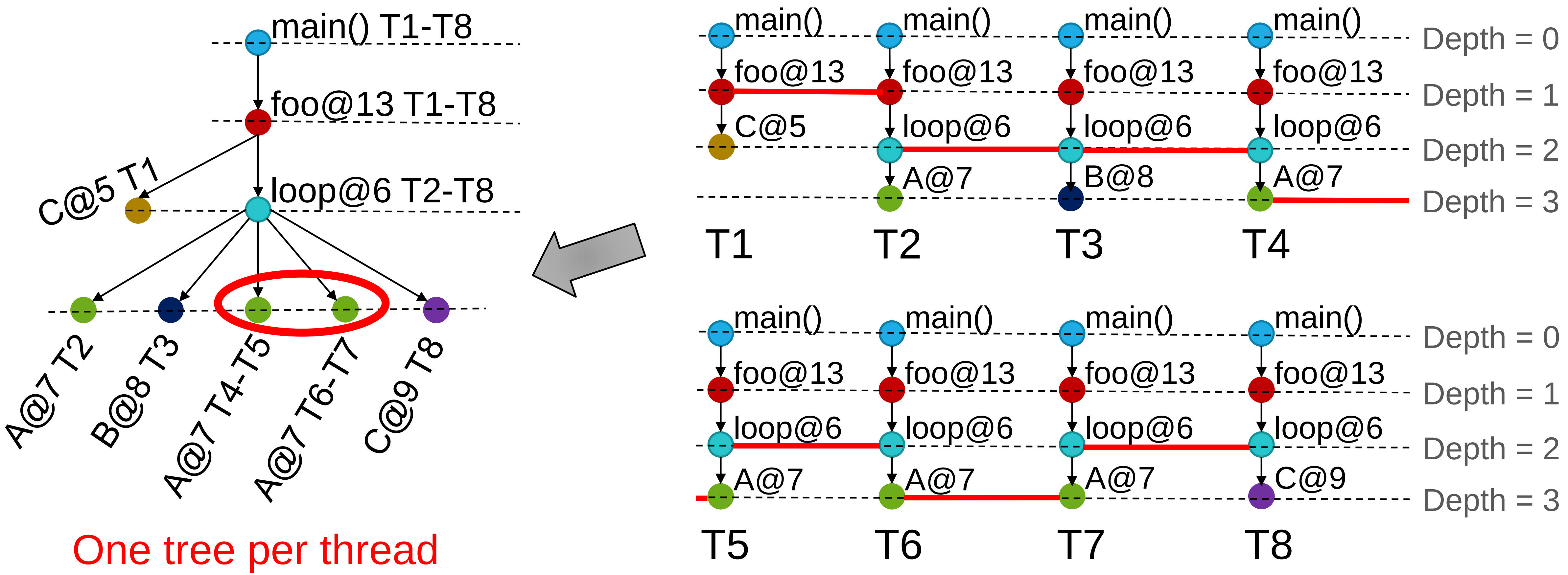
Construct a *Temporal Context Tree*



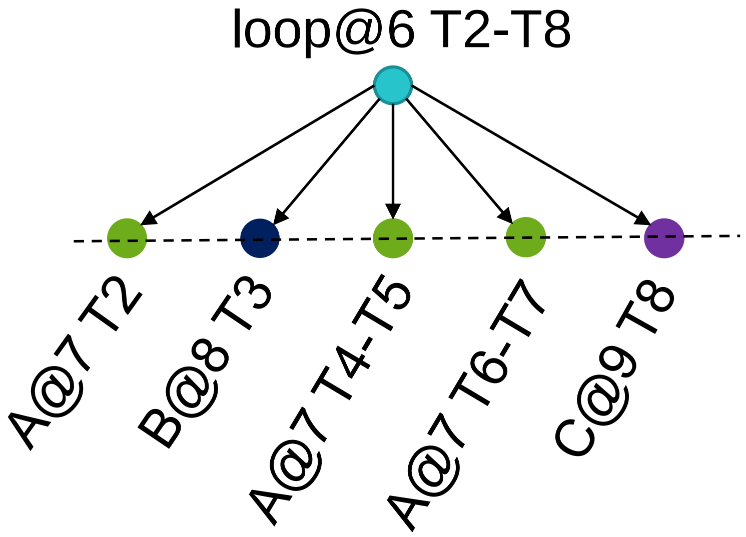
Construct a *Temporal Context Tree*



Construct a *Temporal Context Tree*

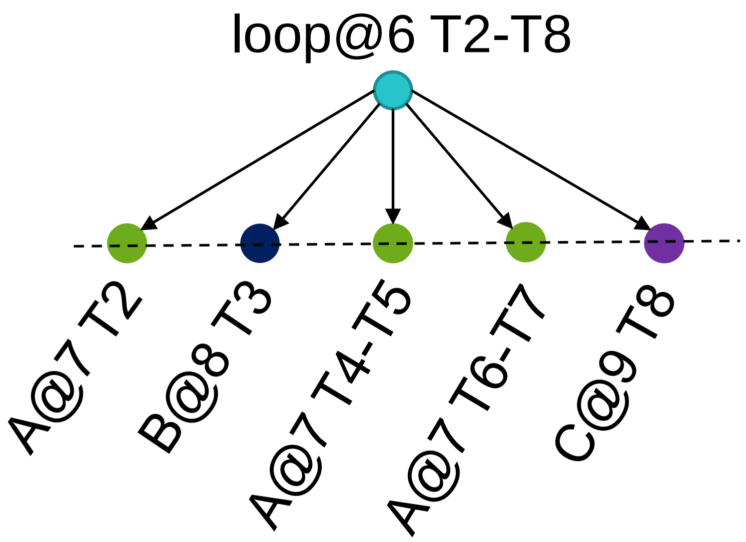


Identify iterations



Identify iterations

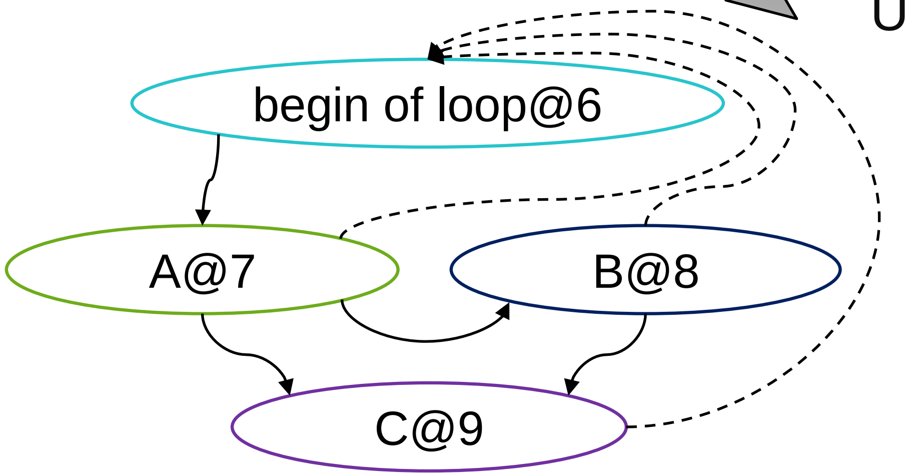
```
for (int k = 0; k < 3; k++) {
        A();
        if (k==0) B();
        if (k==2) C();
}
```

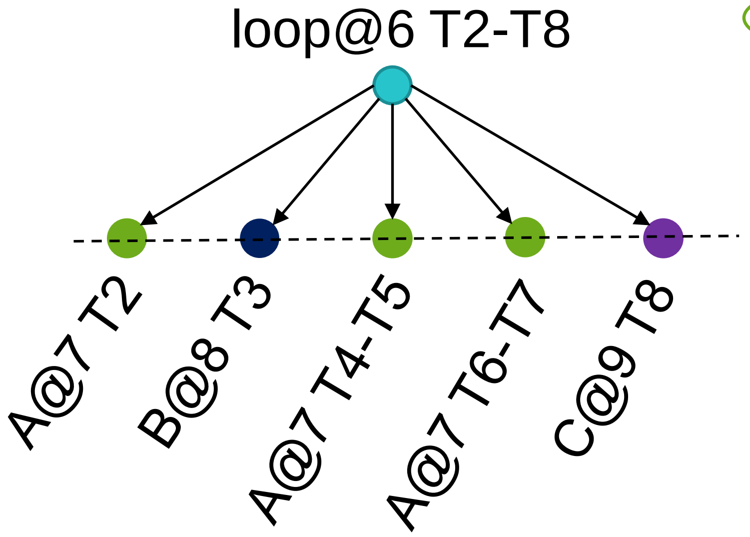


Identify iterations

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

Use ParseAPI to analyze binaries

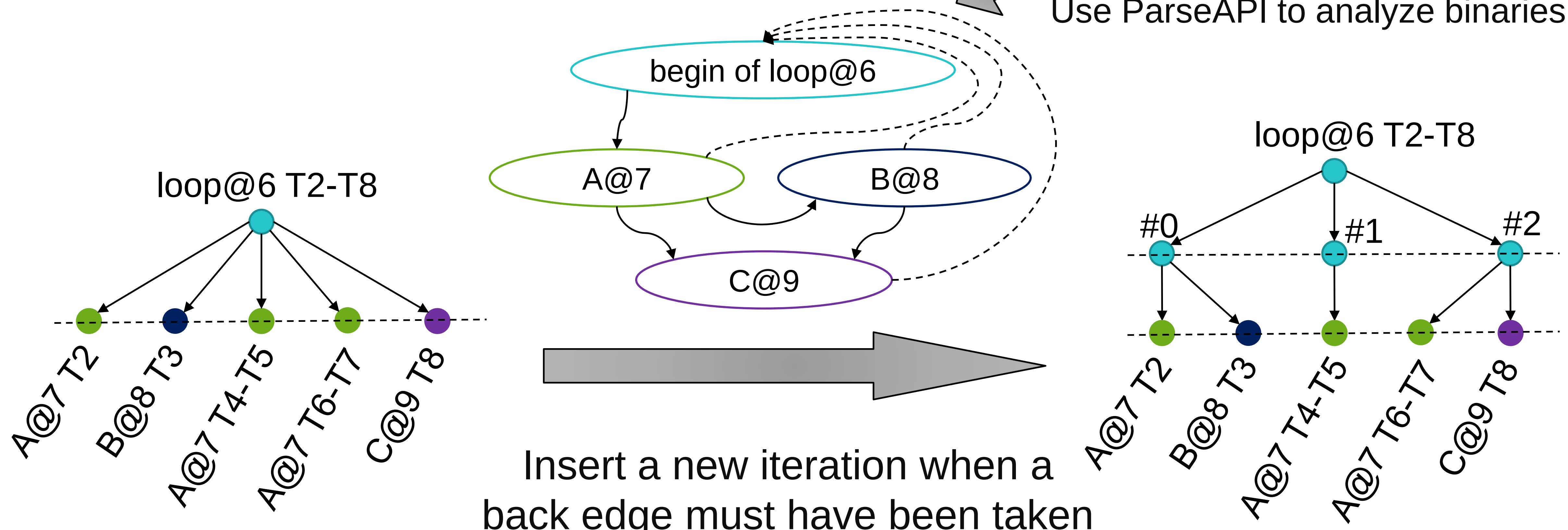




Identify iterations

```
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(N*K*G).   N = number of instances; G = size of *TCTs*
  - Multi-level clustering ✓   Parallelization ✓

Approach

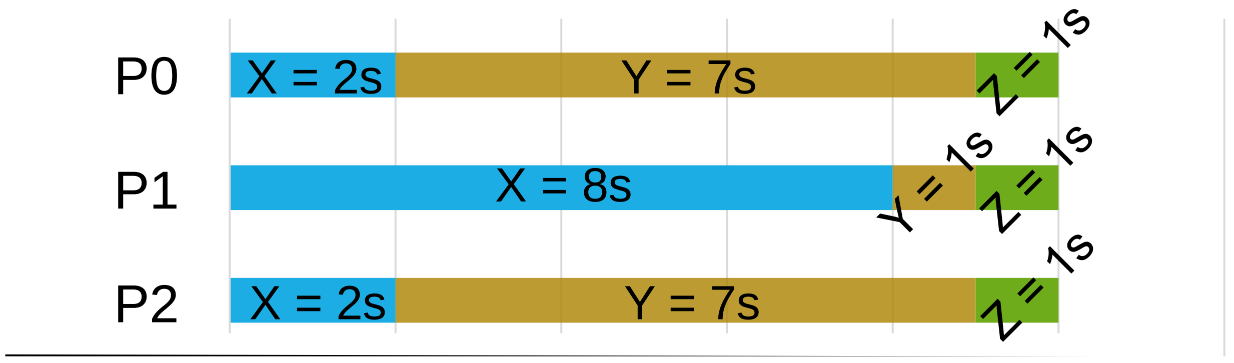
1. Collect and prepare sample-based time-series for further analysis
   - Collect a time series of call paths with HPCToolkit
   - Organize each time series as a tree of program calling contexts
   - Identify iterative behaviors in the time series

2. Build clusters across threads and loop iterations

3. Quantify performance losses and attribute them to call paths

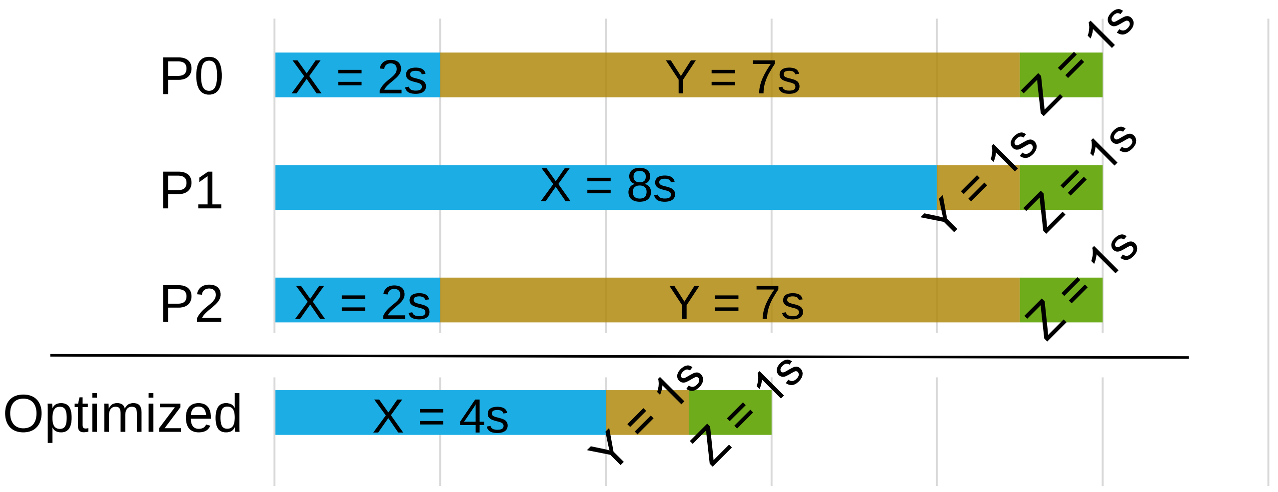
Quantify performance losses

Variation across threads provides clues to performance losses



Quantify imbalance

Assume X as computation, Y & Z as synchronization

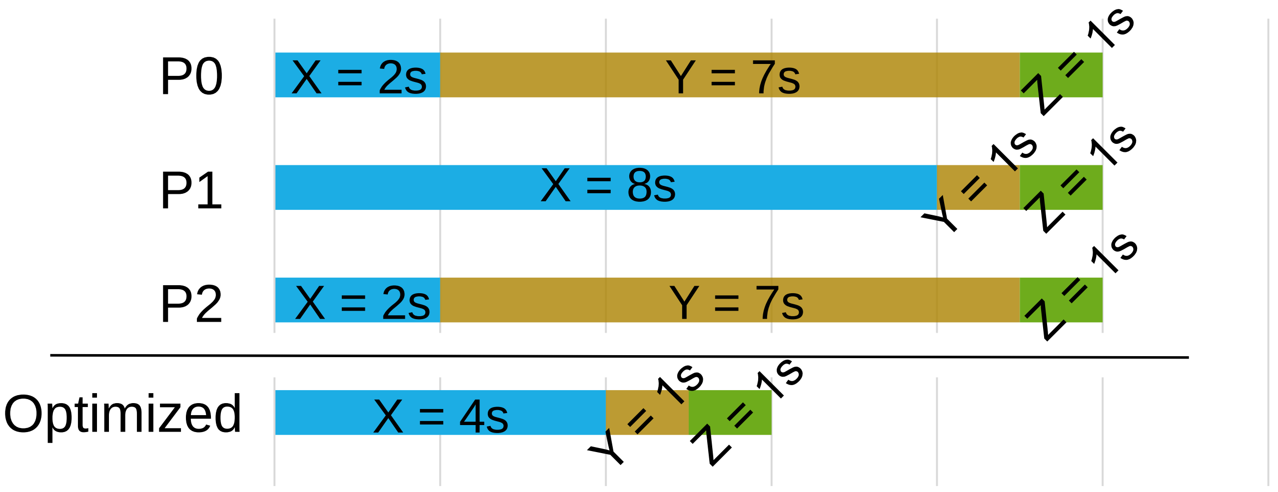


For a computation node C

- *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

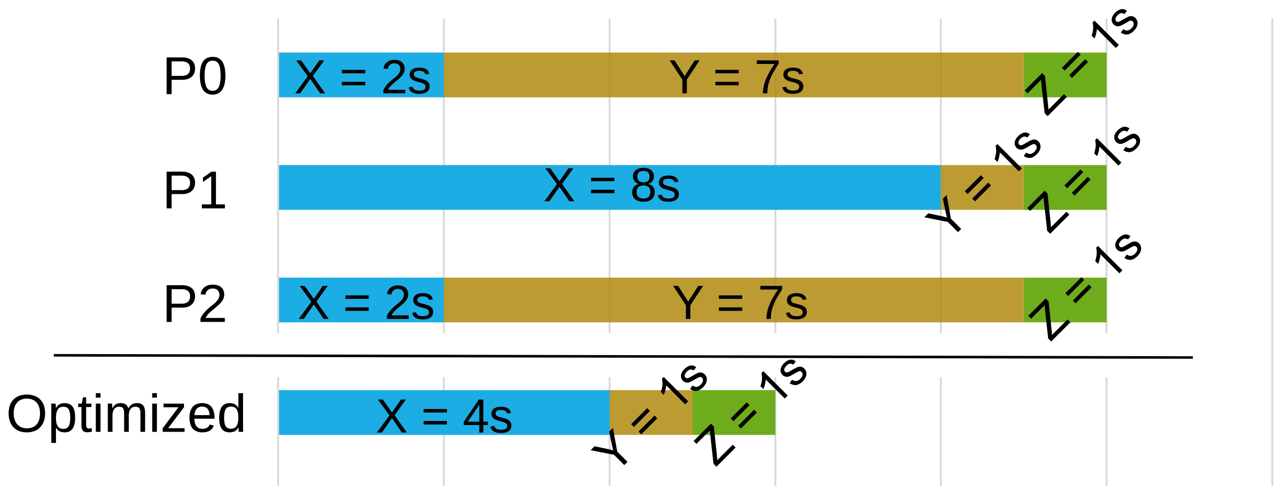


For a computation node C

- *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



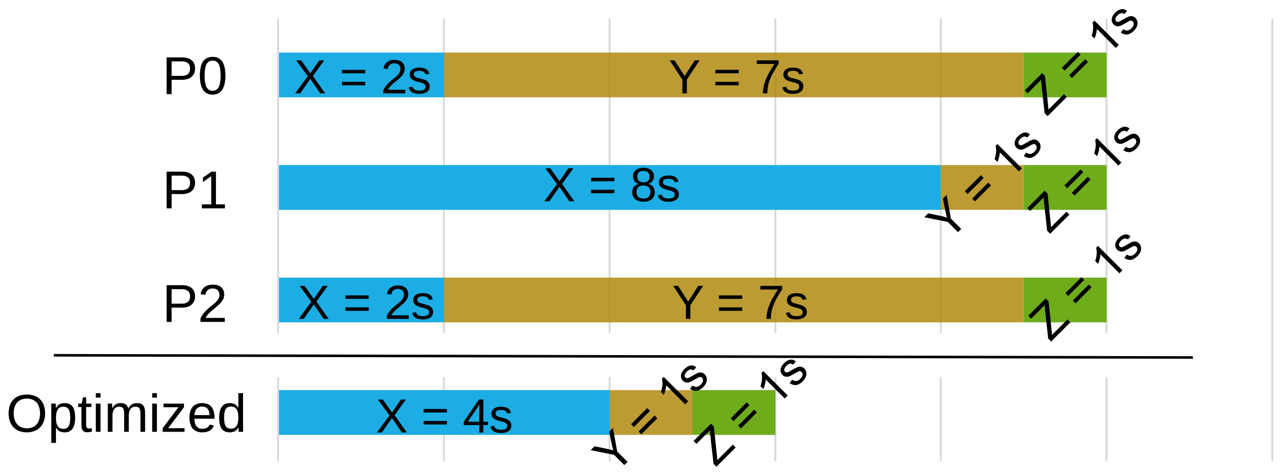
imb(X) = max(X) - avg(X)
= 8s - 4s
= 4s

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

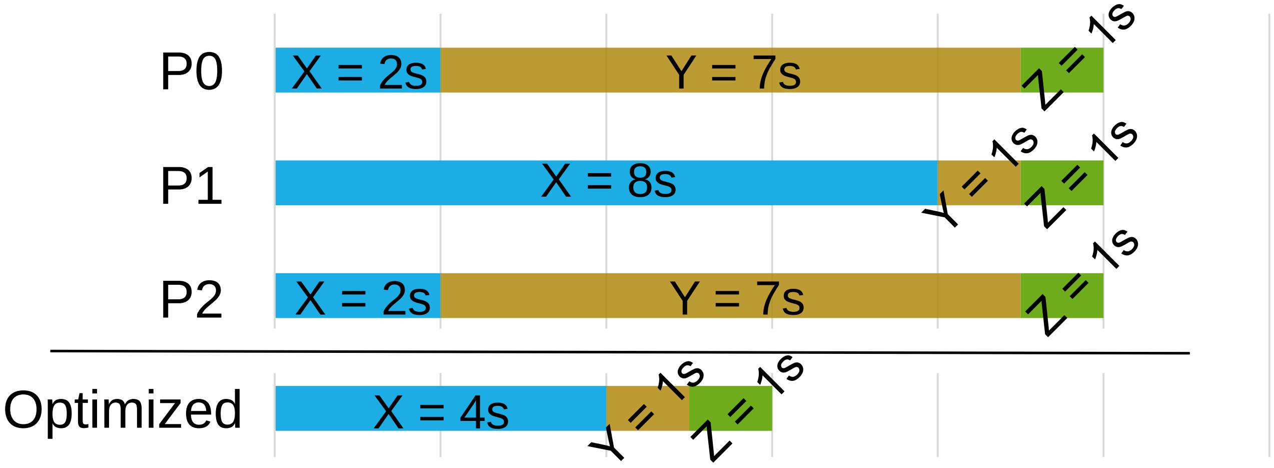


For a synchronization node S

- *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



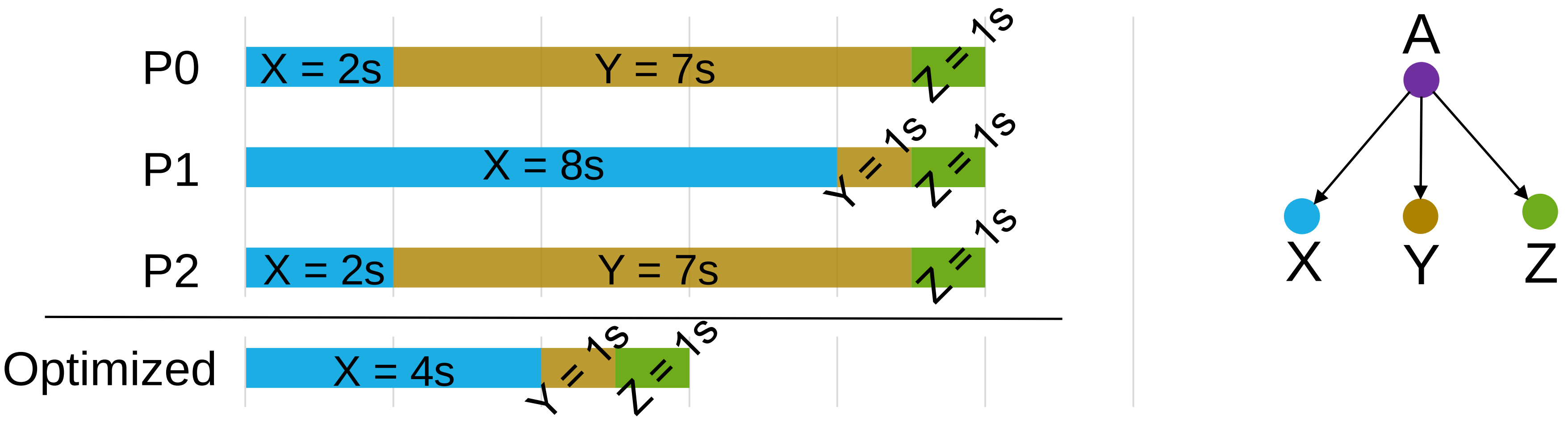
imb(Y) = imb(X)
= 4s

For a synchronization node S

- *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



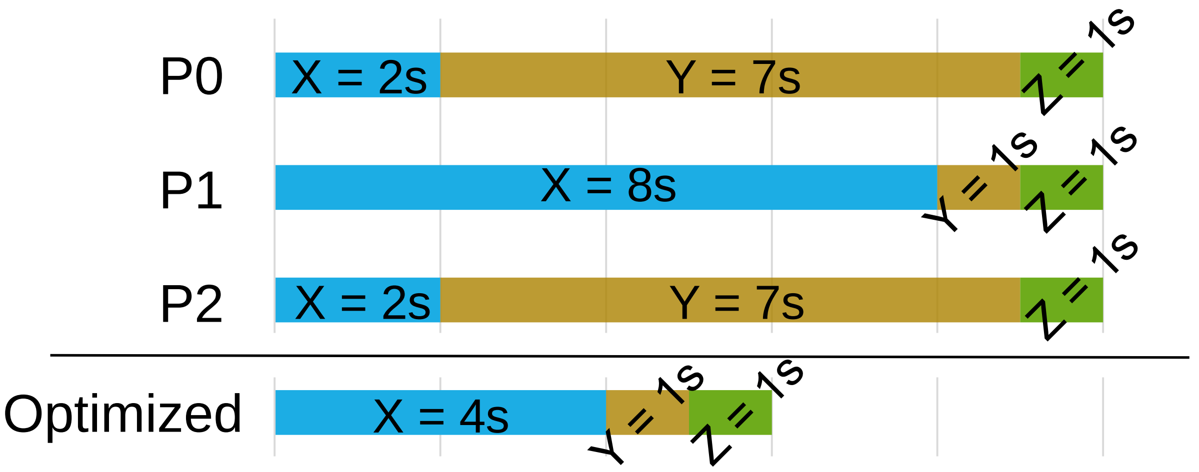
sumImb(N) for each node N in *TCT*

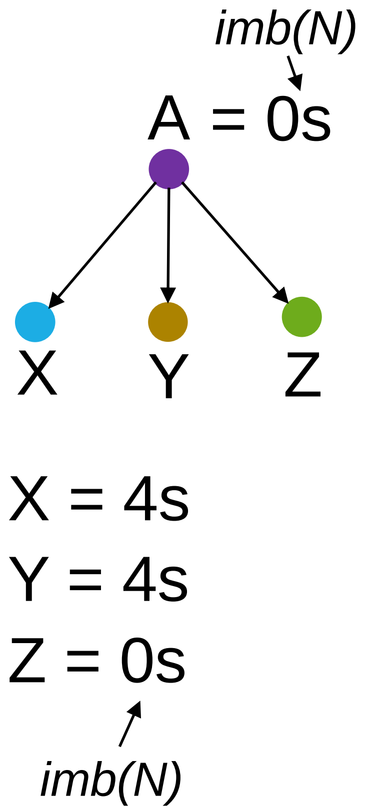
= *imb(N)* if N is a leaf

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

Attribute imbalance

Assume X as computation, Y & Z as synchronization





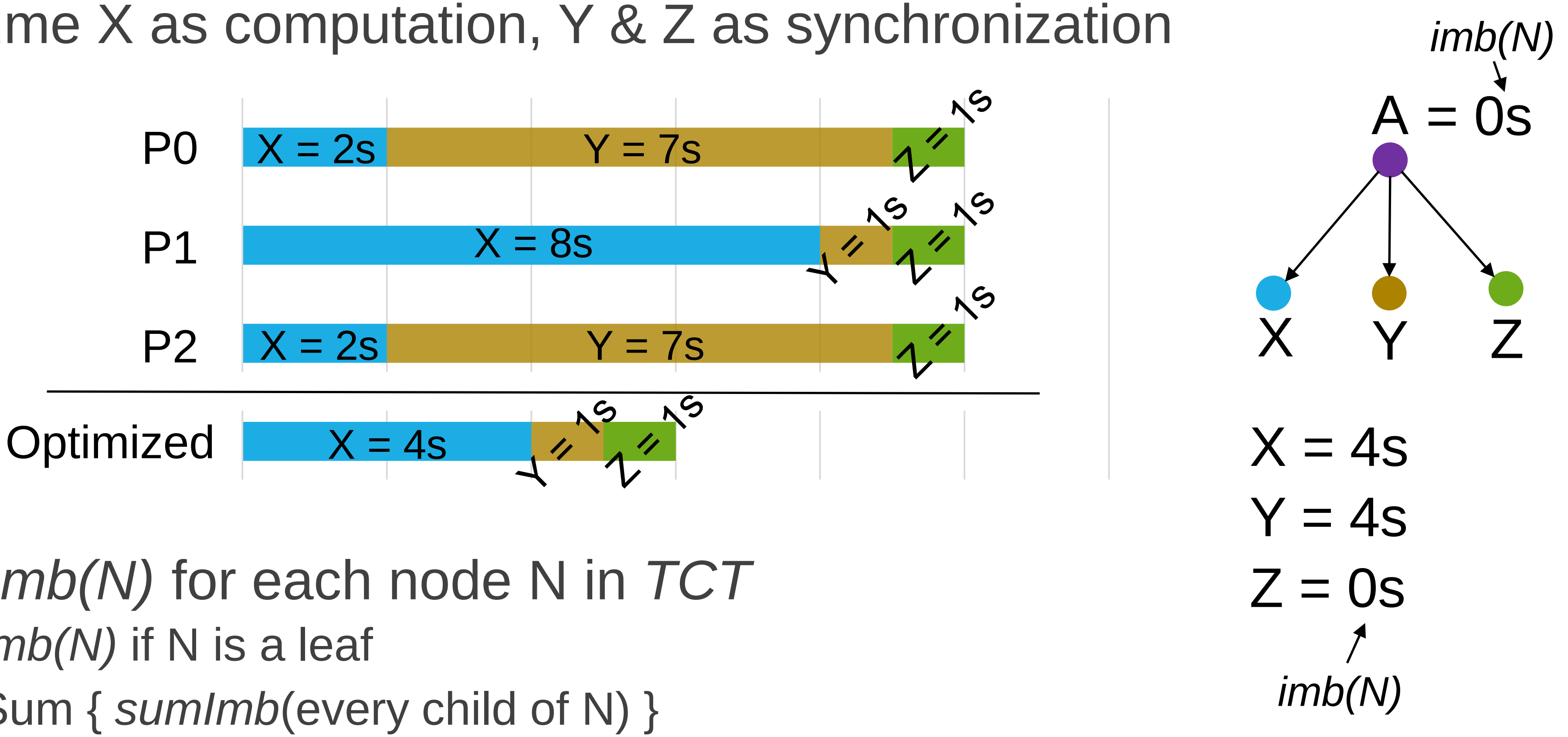
sumImb(N) for each node N in *TCT*

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

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

Attribute imbalance

Assume X as computation, Y & Z as synchronization



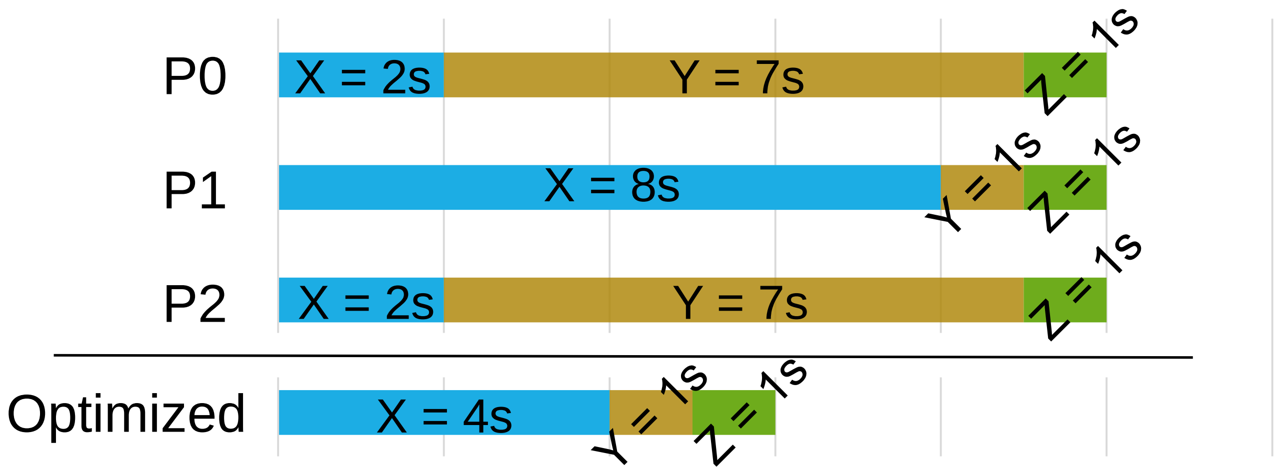
sumImb(N) for each node N in *TCT*

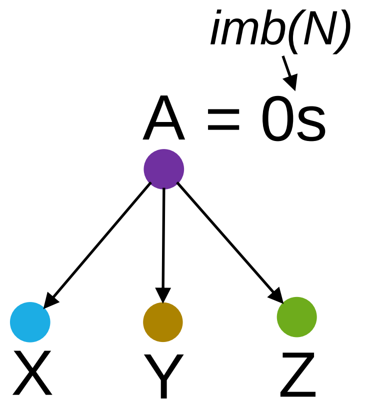
= *imb(N)* if N is a leaf

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

Attribute imbalance

Assume X as computation, Y & Z as synchronization





X = 4s / 4s

Y = 4s / 4s

Z = 0s / 0s

imb(N) *sumImb(N)*

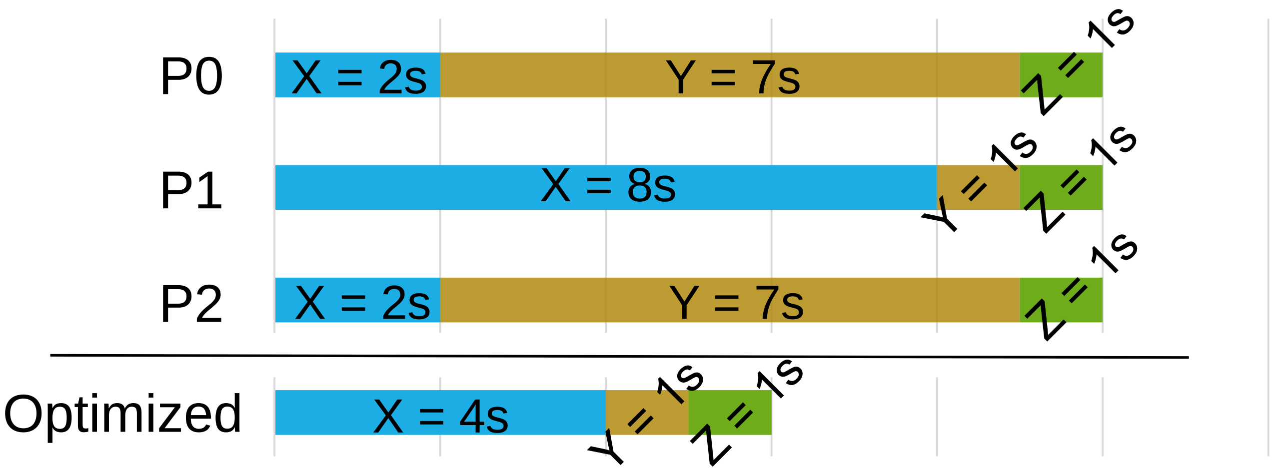
sumImb(N) for each node N in *TCT*

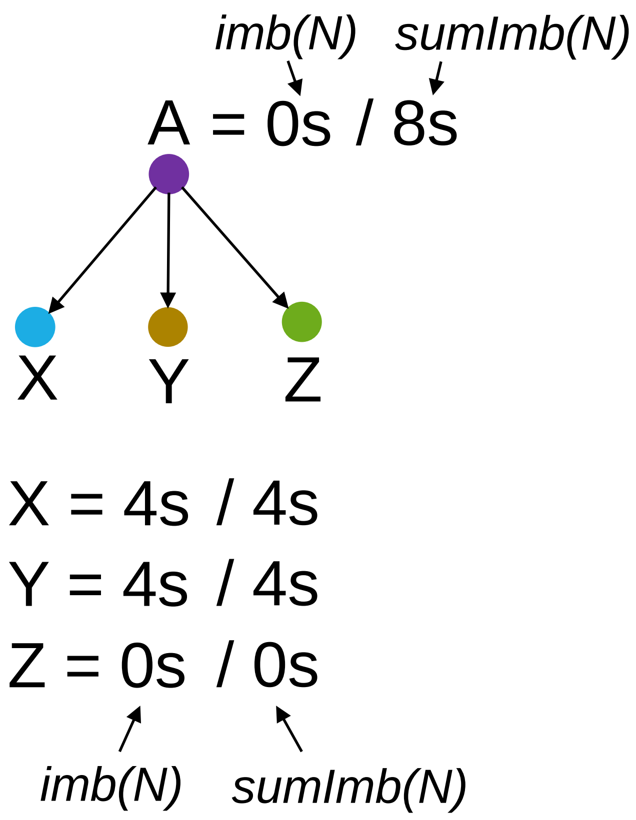
= *imb(N)* if N is a leaf

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

Attribute imbalance

Assume X as computation, Y & Z as synchronization





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

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

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

Highlight significant call paths

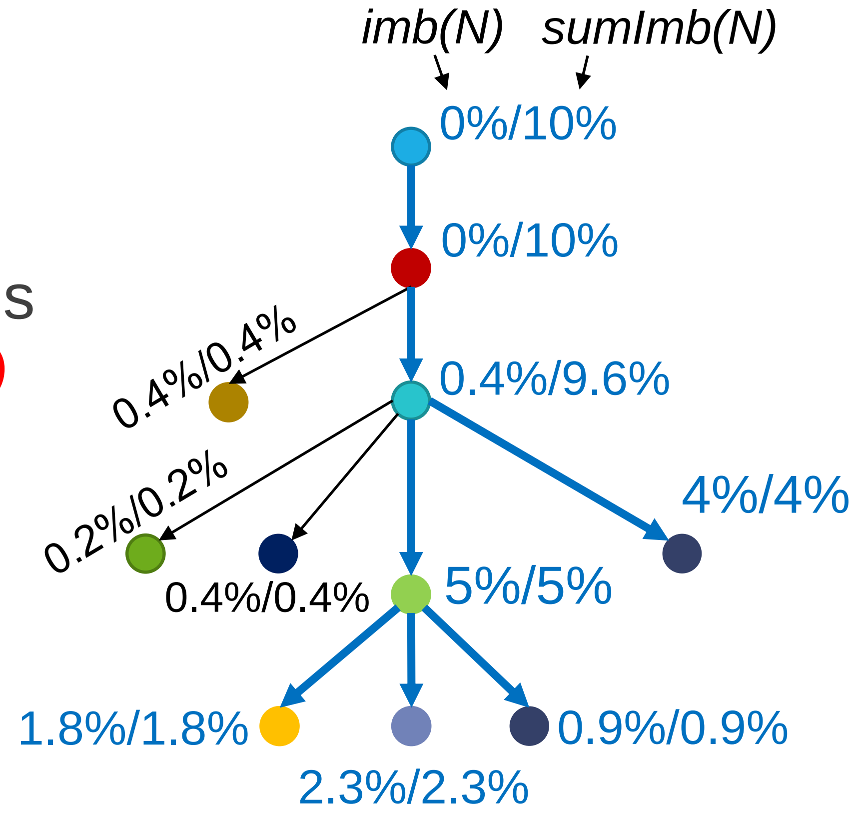
Pick call paths that contribute significantly to imbalance

- *sumImb(N)* / RunTime > *significanceRatio (= 1%)*
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths

- *imb(N) / sumImb(N) > appropriateDepthRatio (= 70%)*
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way



Highlight significant call paths

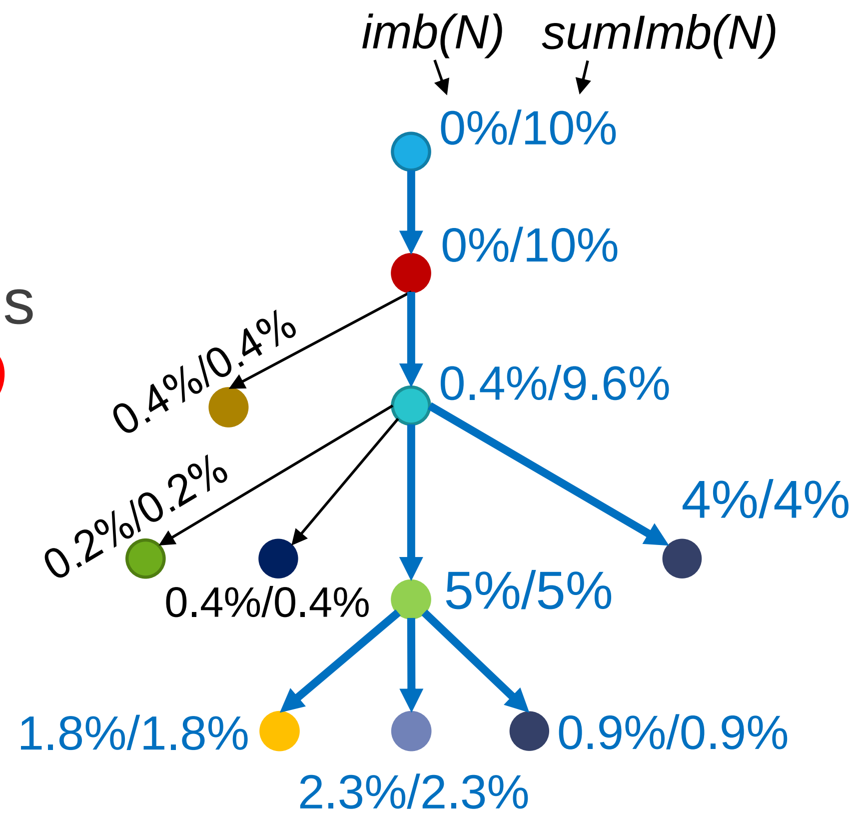
Pick call paths that contribute significantly to imbalance

- *sumImb(N)* / RunTime > *significanceRatio (= 1%)*
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths

- *imb(N) / sumImb(N) > appropriateDepthRatio (= 70%)*
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way



Highlight significant call paths

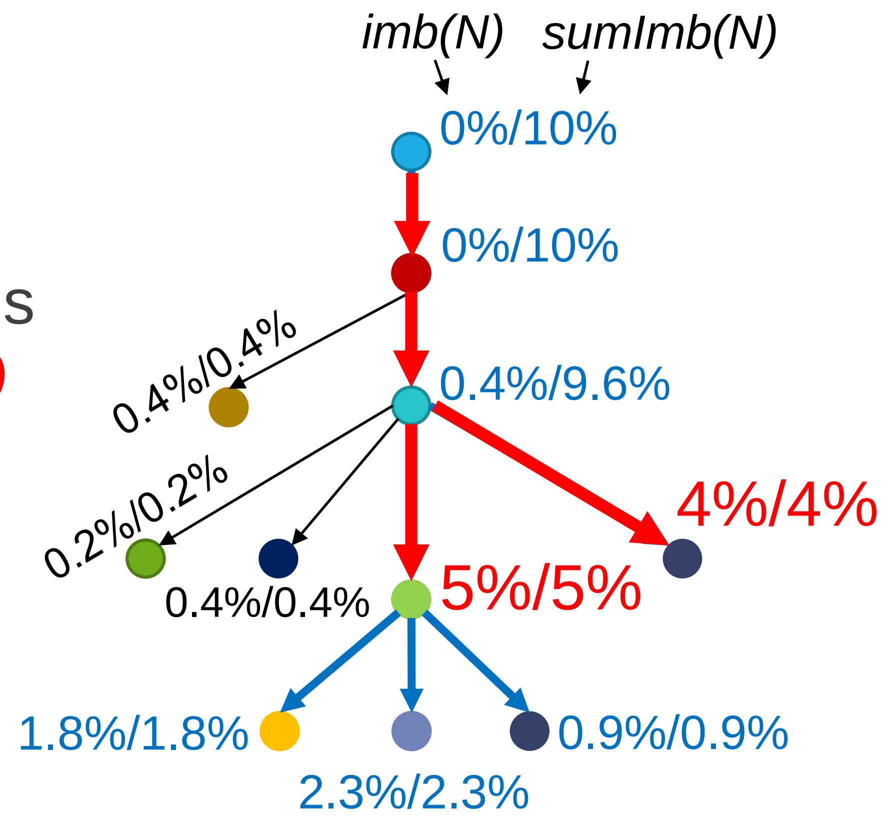
Pick call paths that contribute significantly to imbalance

- *sumImb(N)* / RunTime > *significanceRatio (= 1%)*
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths

- *imb(N) / sumImb(N) > appropriateDepthRatio (= 70%)*
- Avoid reporting too many children with small losses

Quantify and attribute waiting in a similar way



Highlight significant call paths

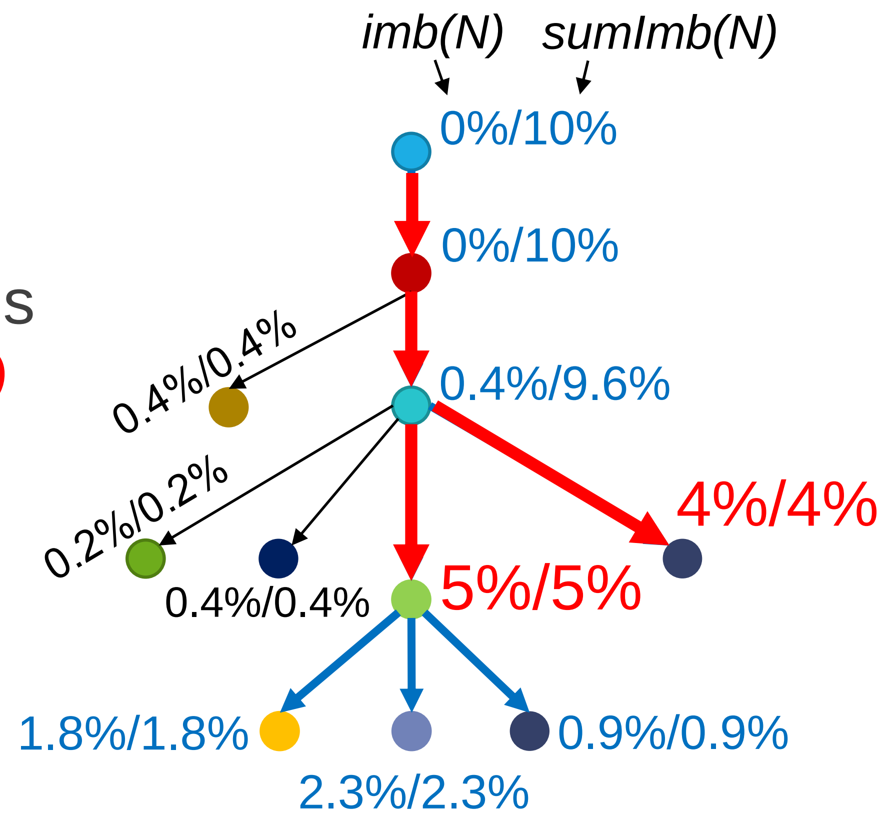
Pick call paths that contribute significantly to imbalance

- *sumImb(N)* / RunTime > *significanceRatio (= 1%)*
- Avoid reporting call paths with tiny losses

Pick appropriate depths for significant call paths

- *imb(N) / sumImb(N) > 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

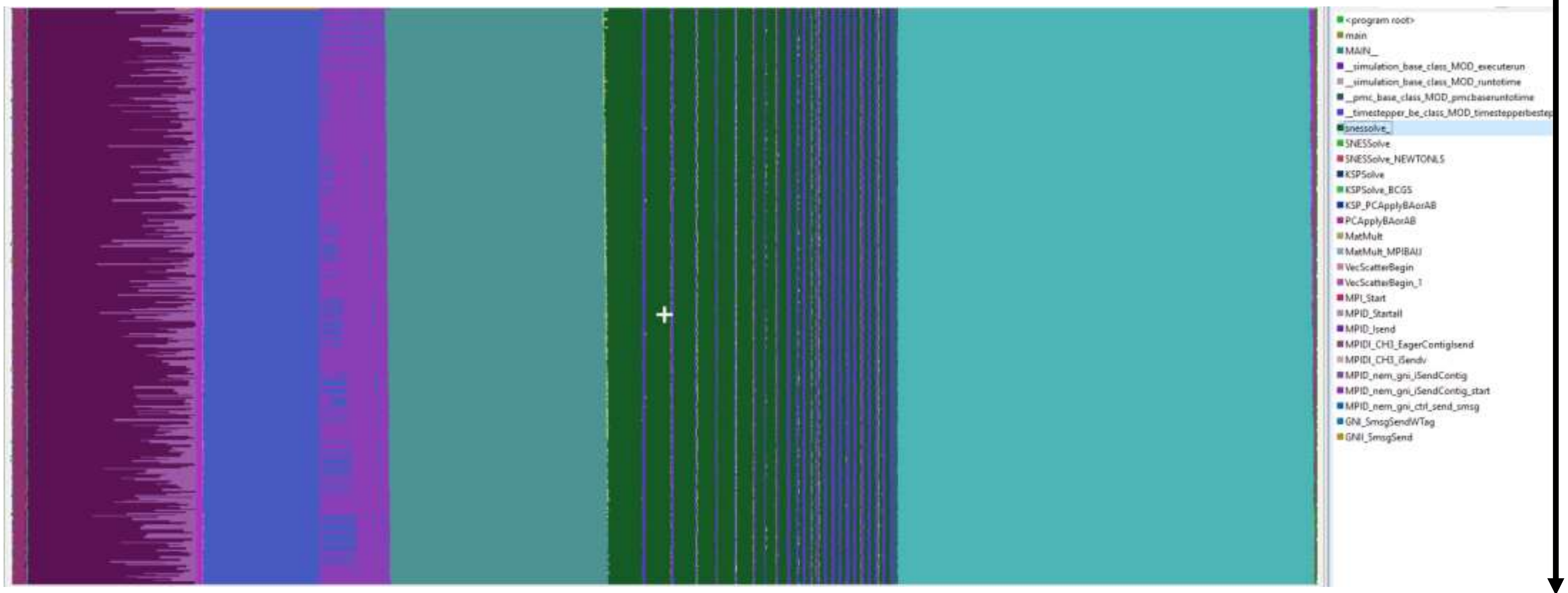
- AMG2013 @ 512 MPI ranks, ~23 seconds
  - Parallel solver of structured/unstructured linear systems

Manual analysis of PFLOTRAN?

Run time = 178s

P0

P511



Depth of callpath

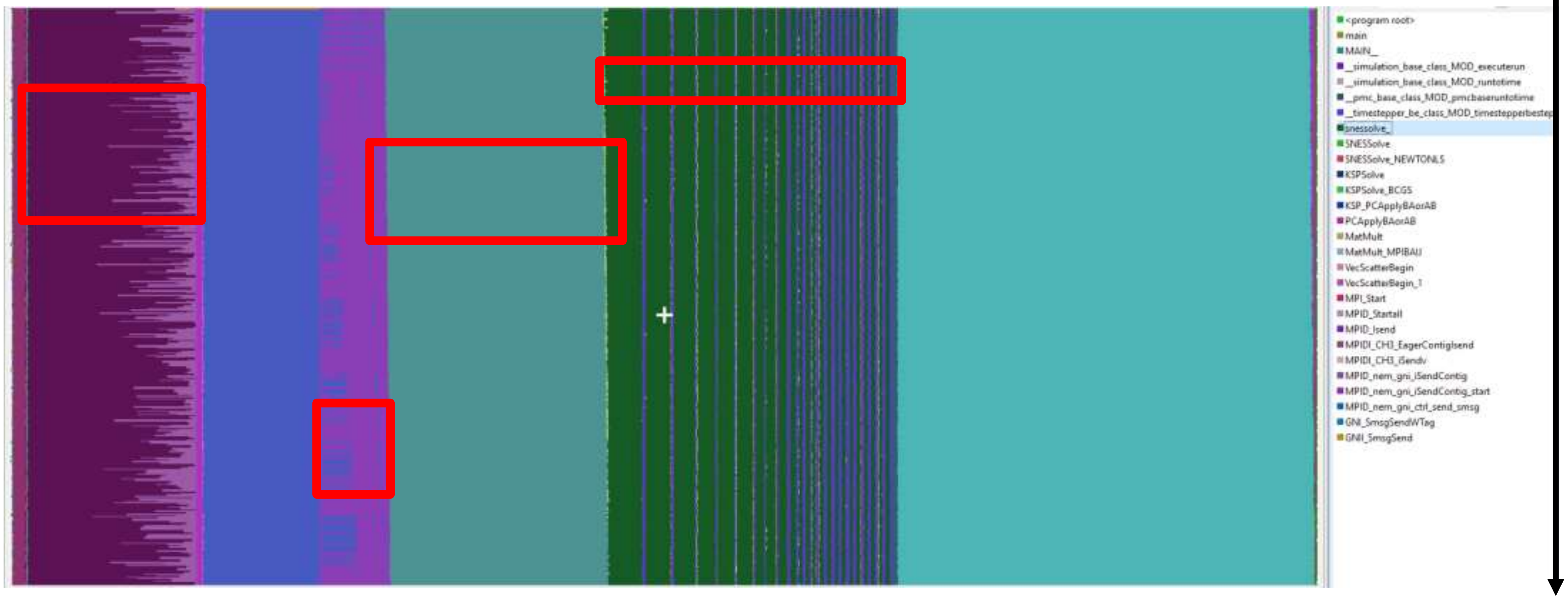
Manual analysis of PFLOTRAN?

Run time = 178s

P0

P511

Depth of callpath



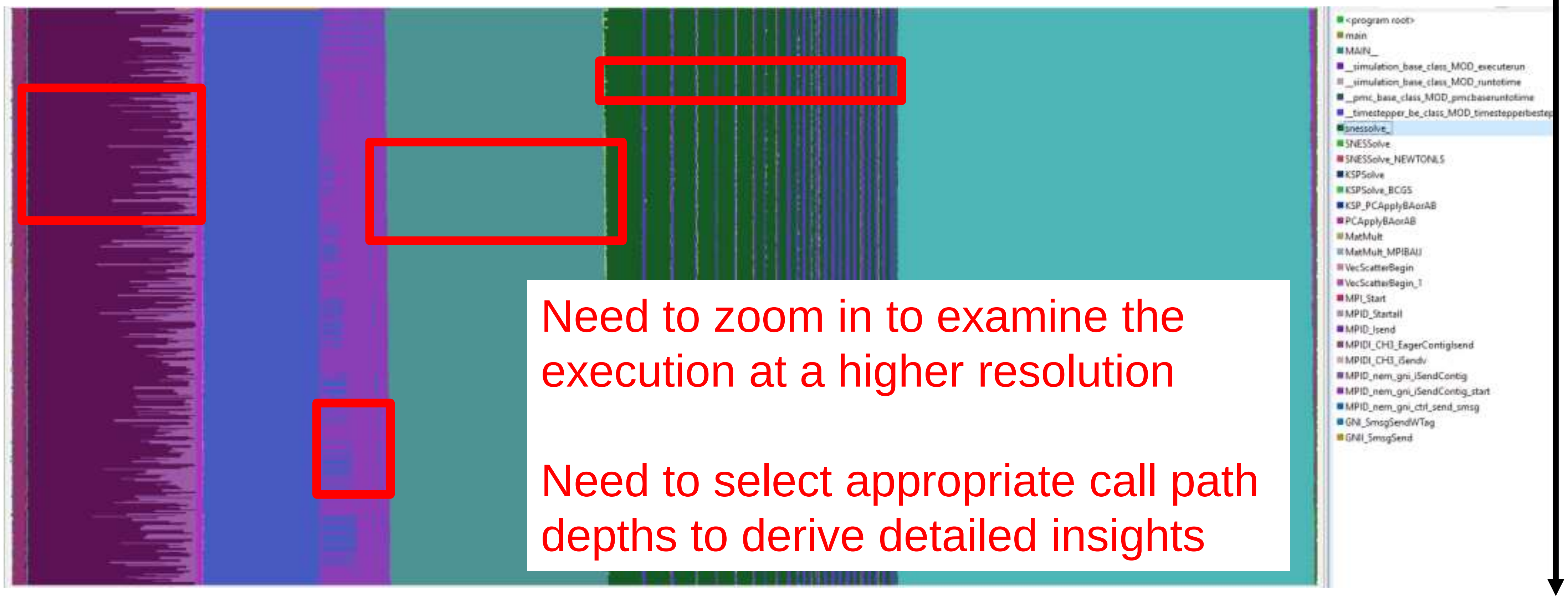
Manual analysis of PFLOTRAN?

Run time = 178s

P0

P511

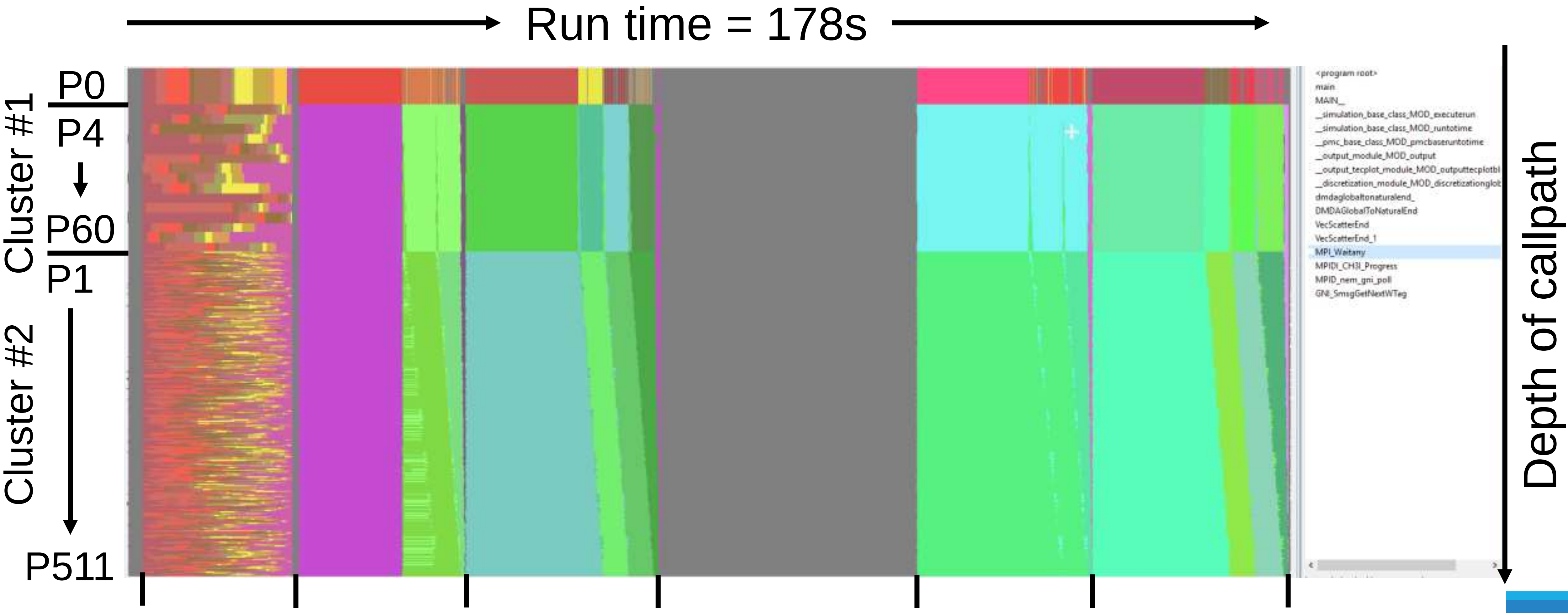
Depth of callpath



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

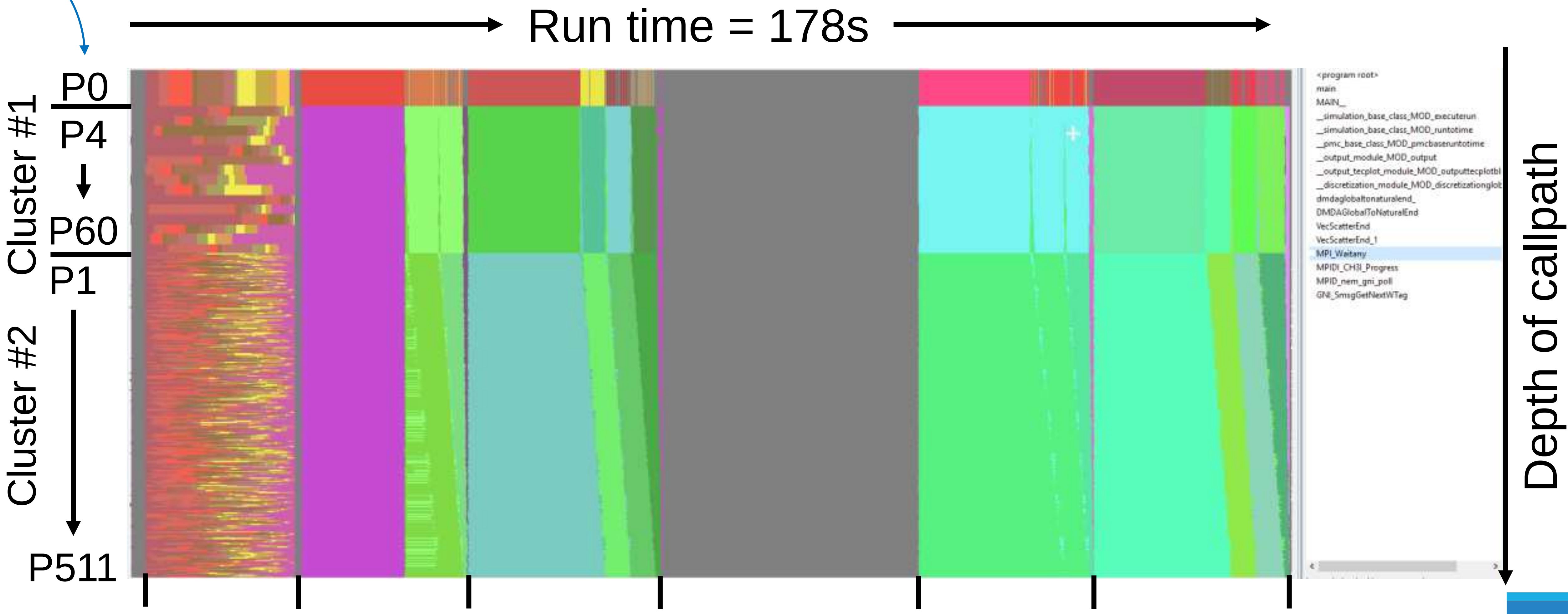
Need to select appropriate call path depths to derive detailed insights

Visualization with automated insights



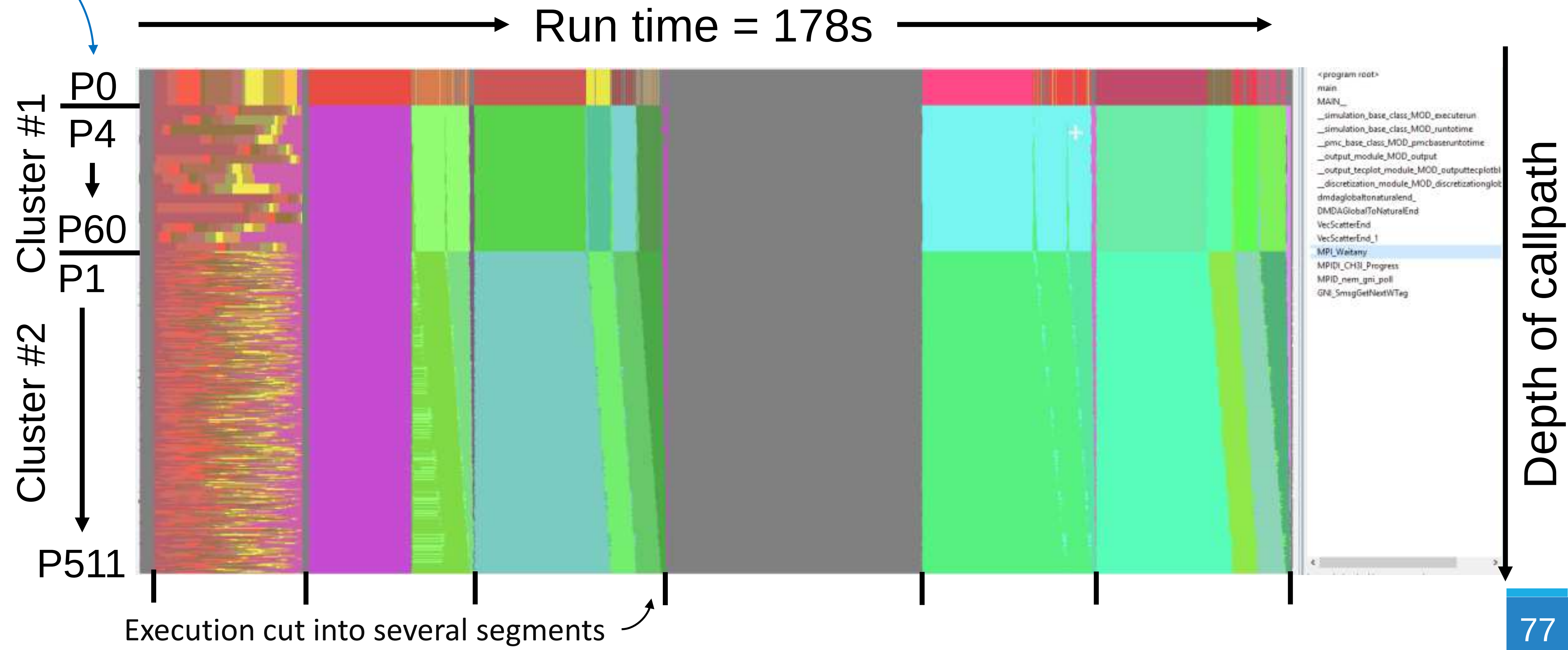
Processes are ordered first by clusters and then by MPI rank
Height of each cluster is proportional to $\log_2(\text{size}+1)$

Visualization with automated insights



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

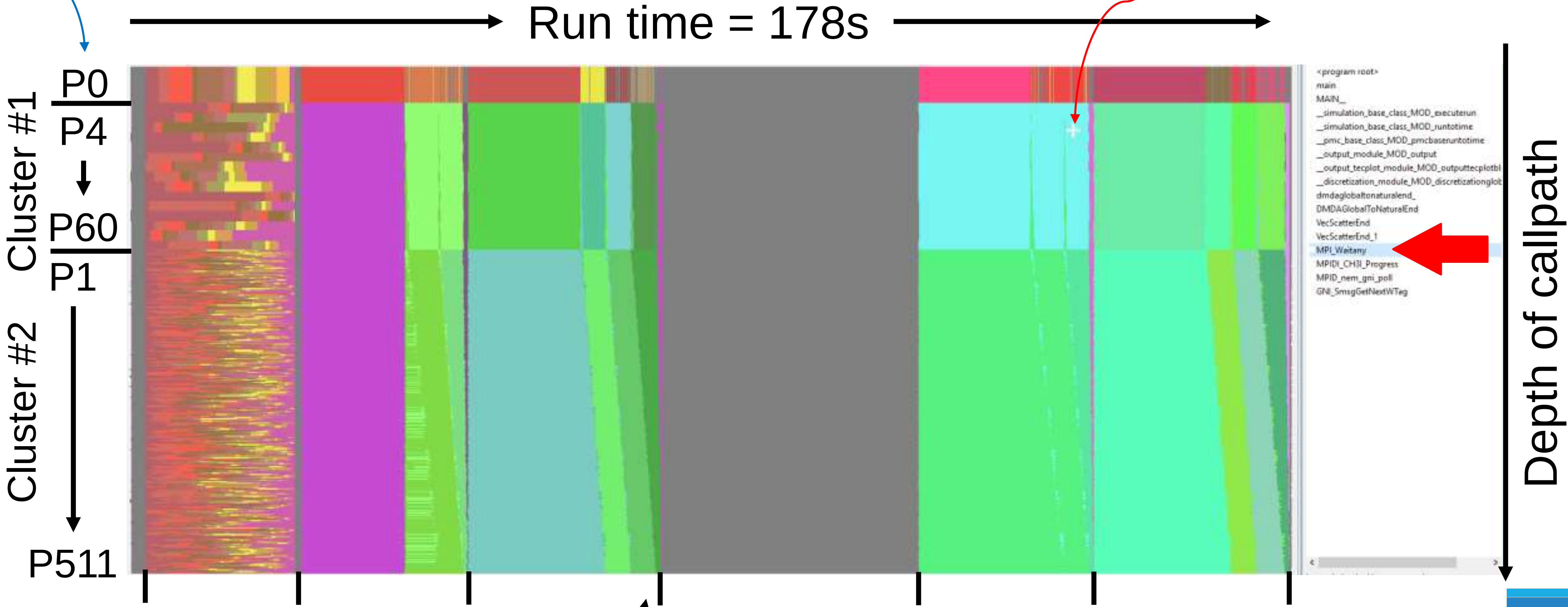
Visualization with automated insights



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

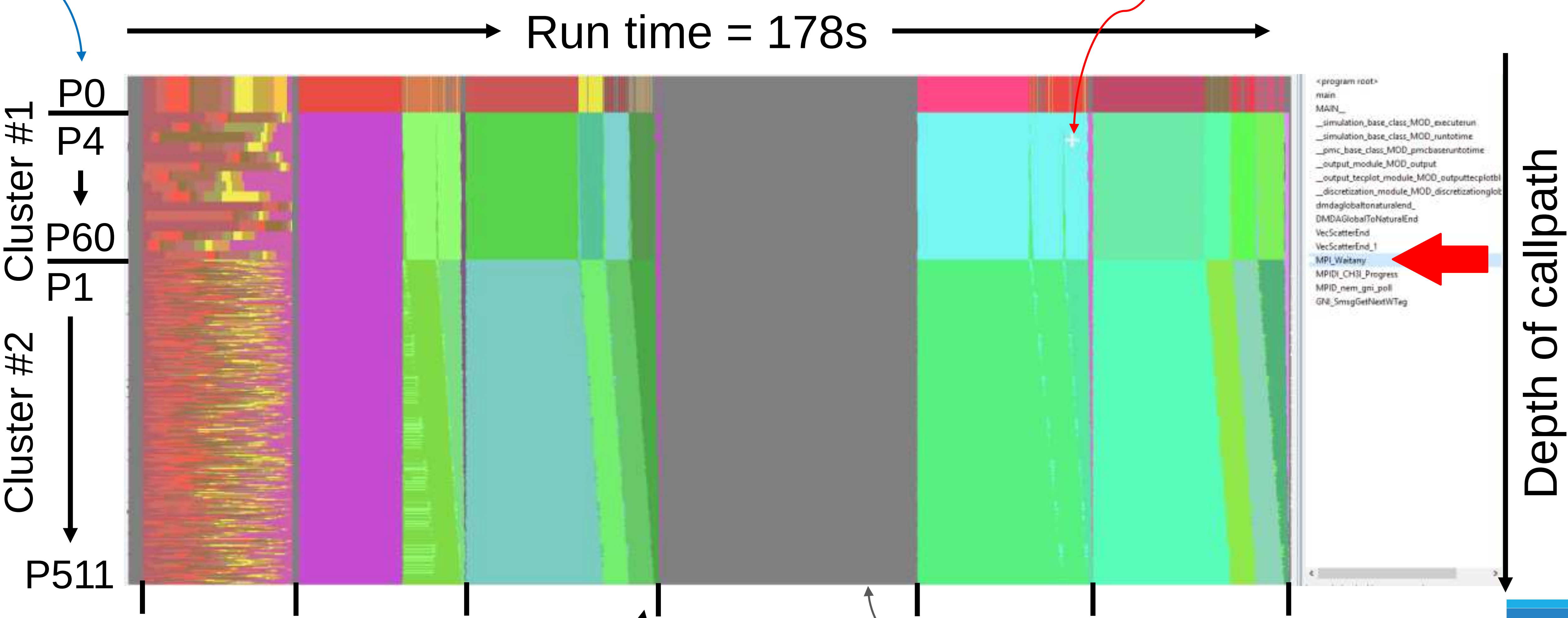


Execution cut into several segments

Visualization with automated insights

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



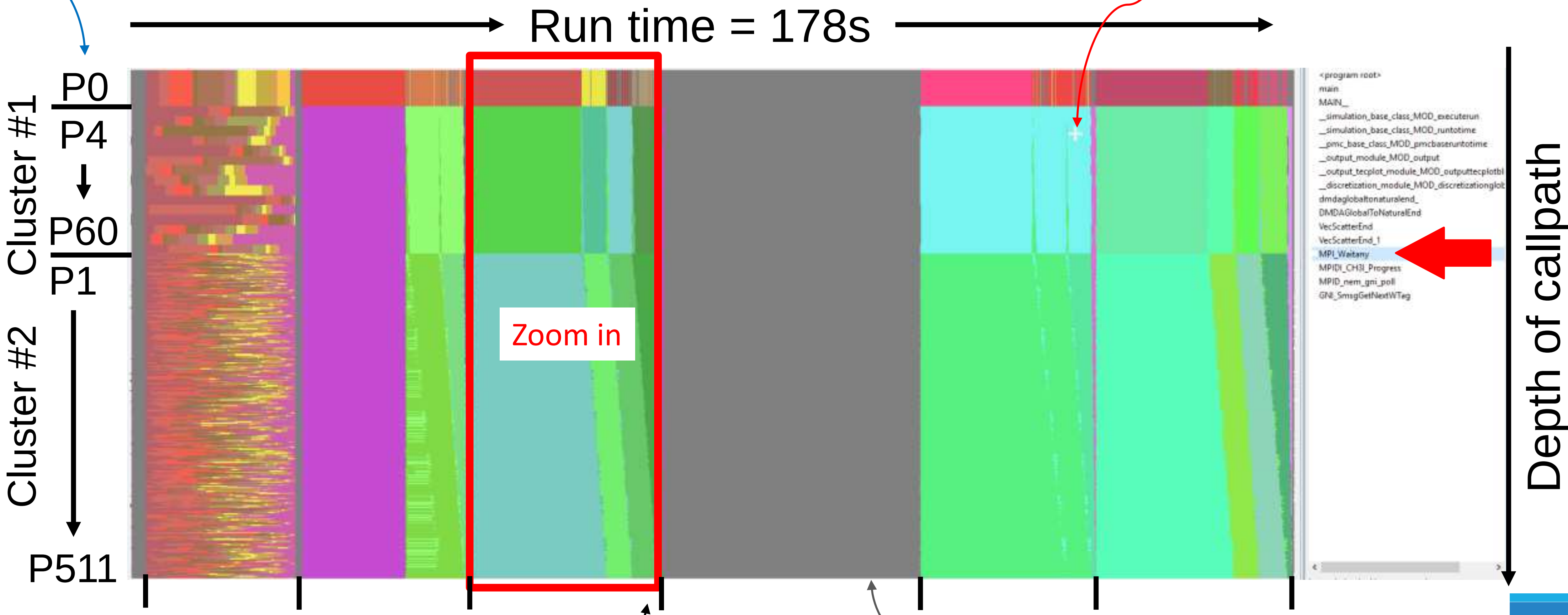
Execution cut into several segments

Insignificant call paths are colored grey

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

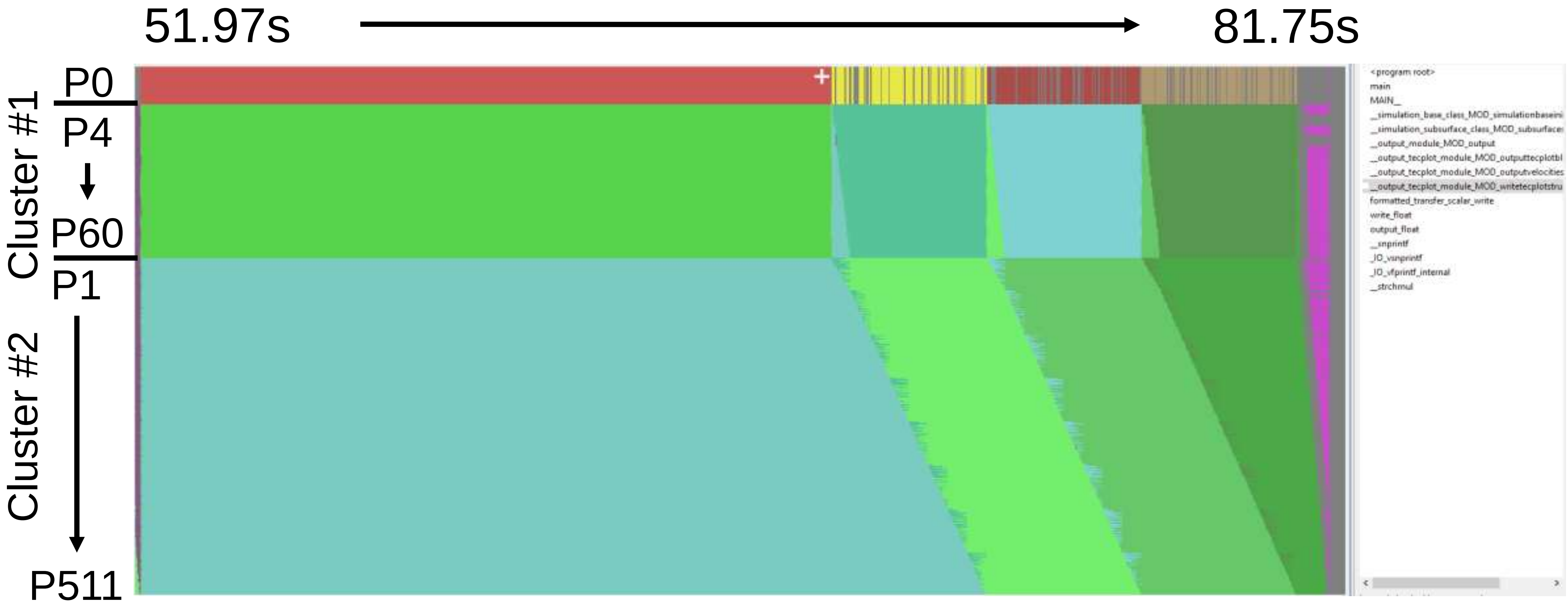
Visualization with automated insights



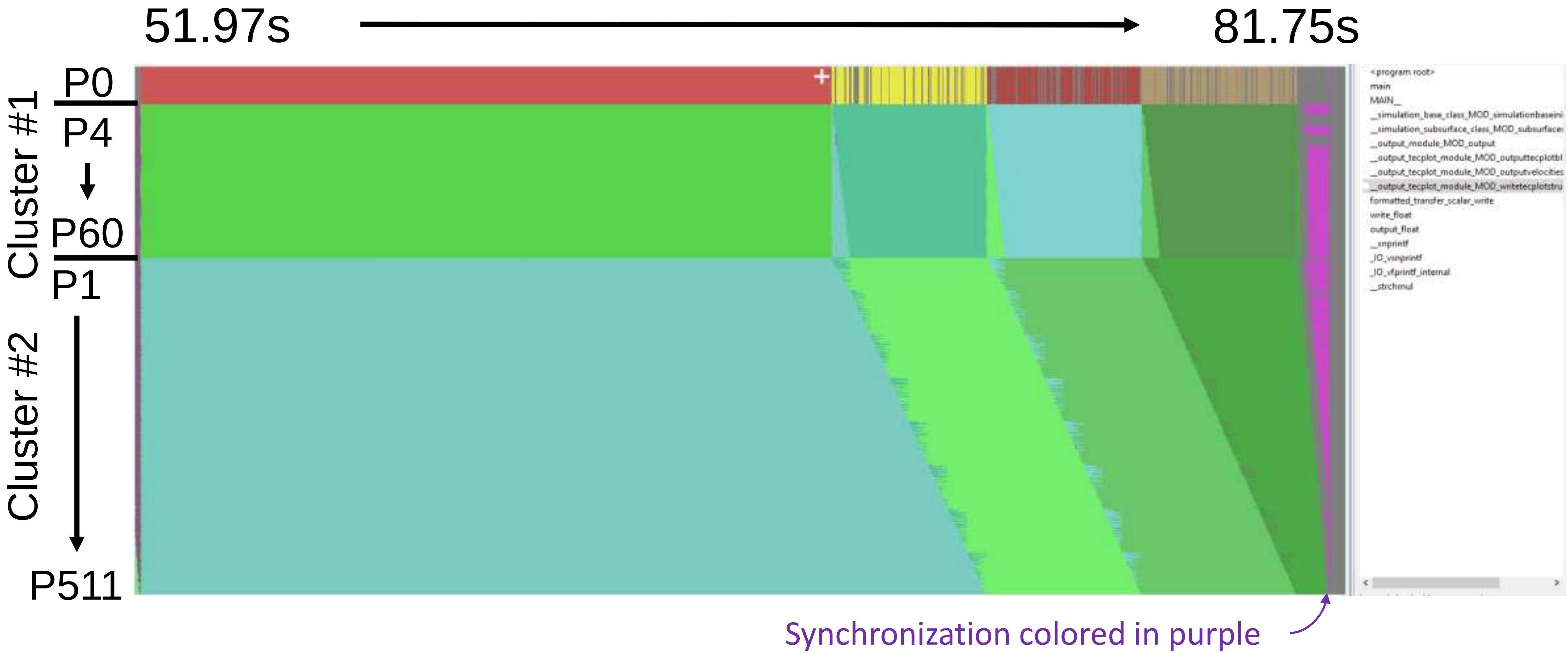
Execution cut into several segments

Insignificant call paths are colored grey

Understand behavior of PFLOTRAN



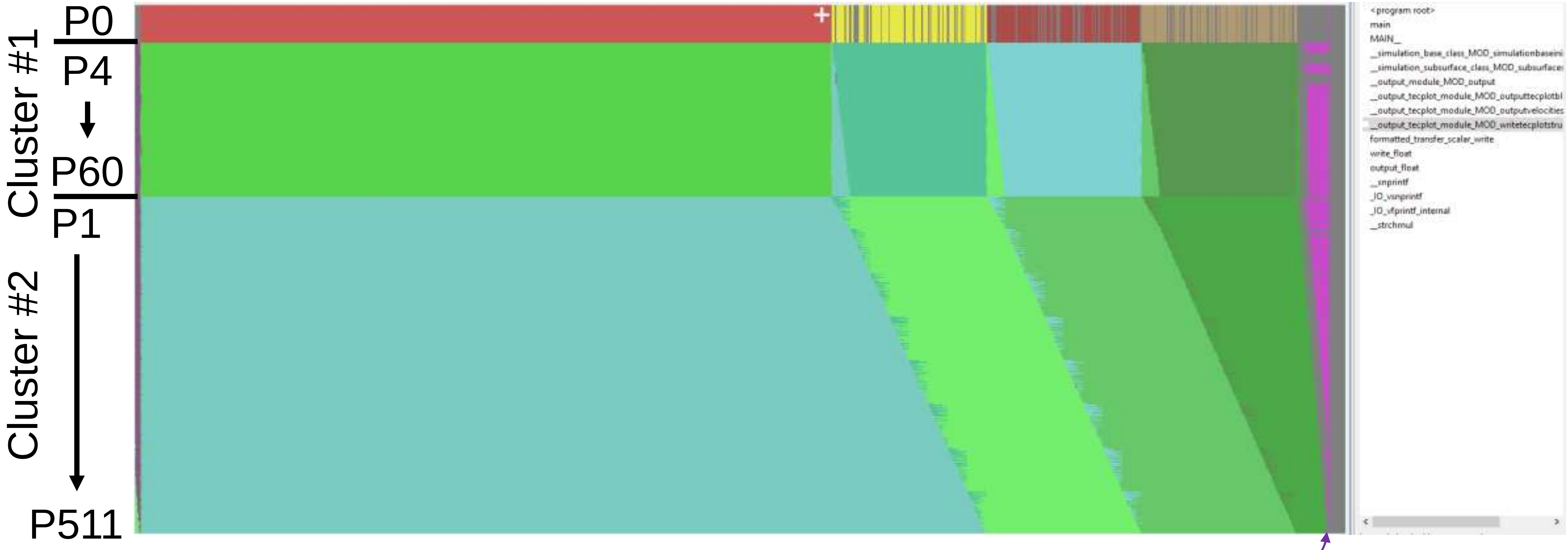
Understand behavior of PFLOTRAN



Work colored in red, yellow, brown

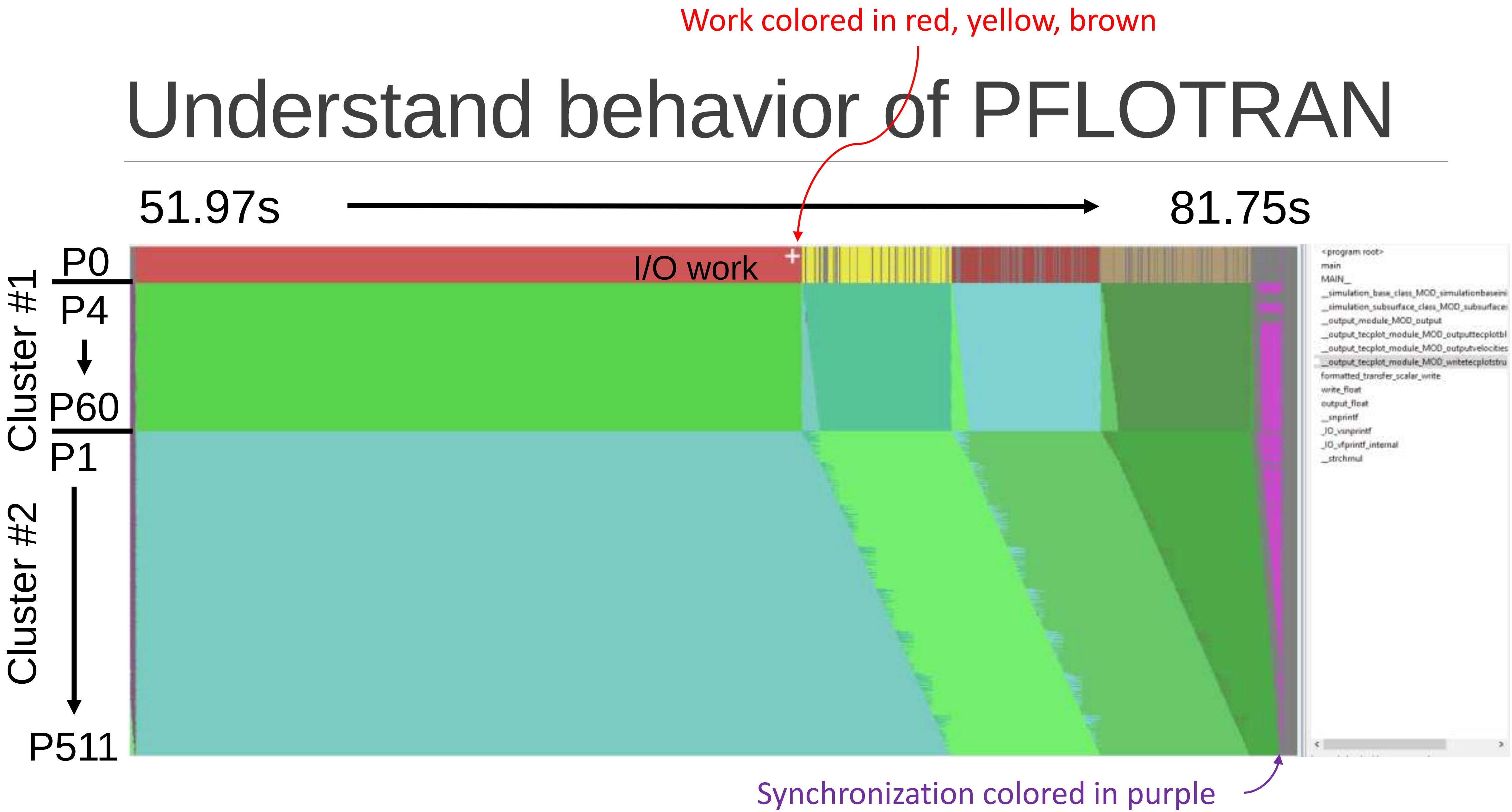
Understand behavior of PFLOTRAN

51.97s → 81.75s

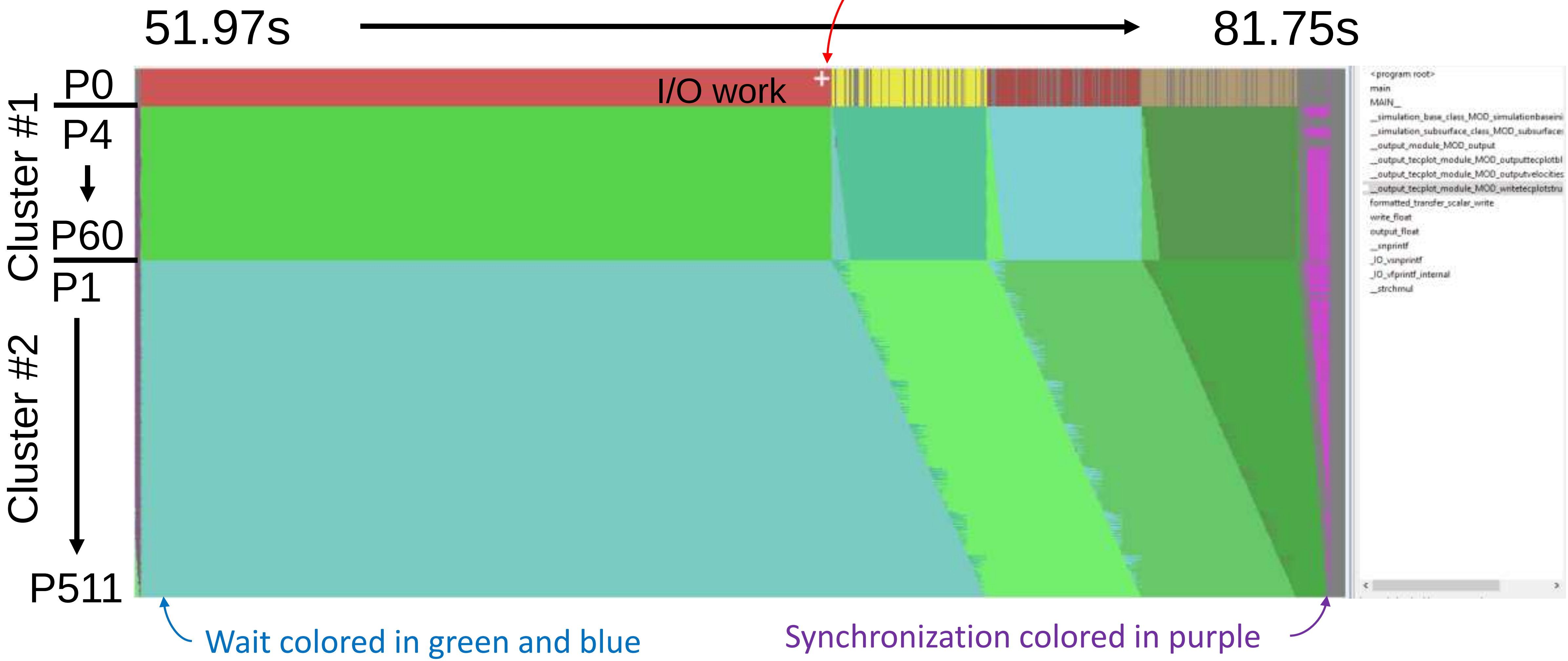


Synchronization colored in purple

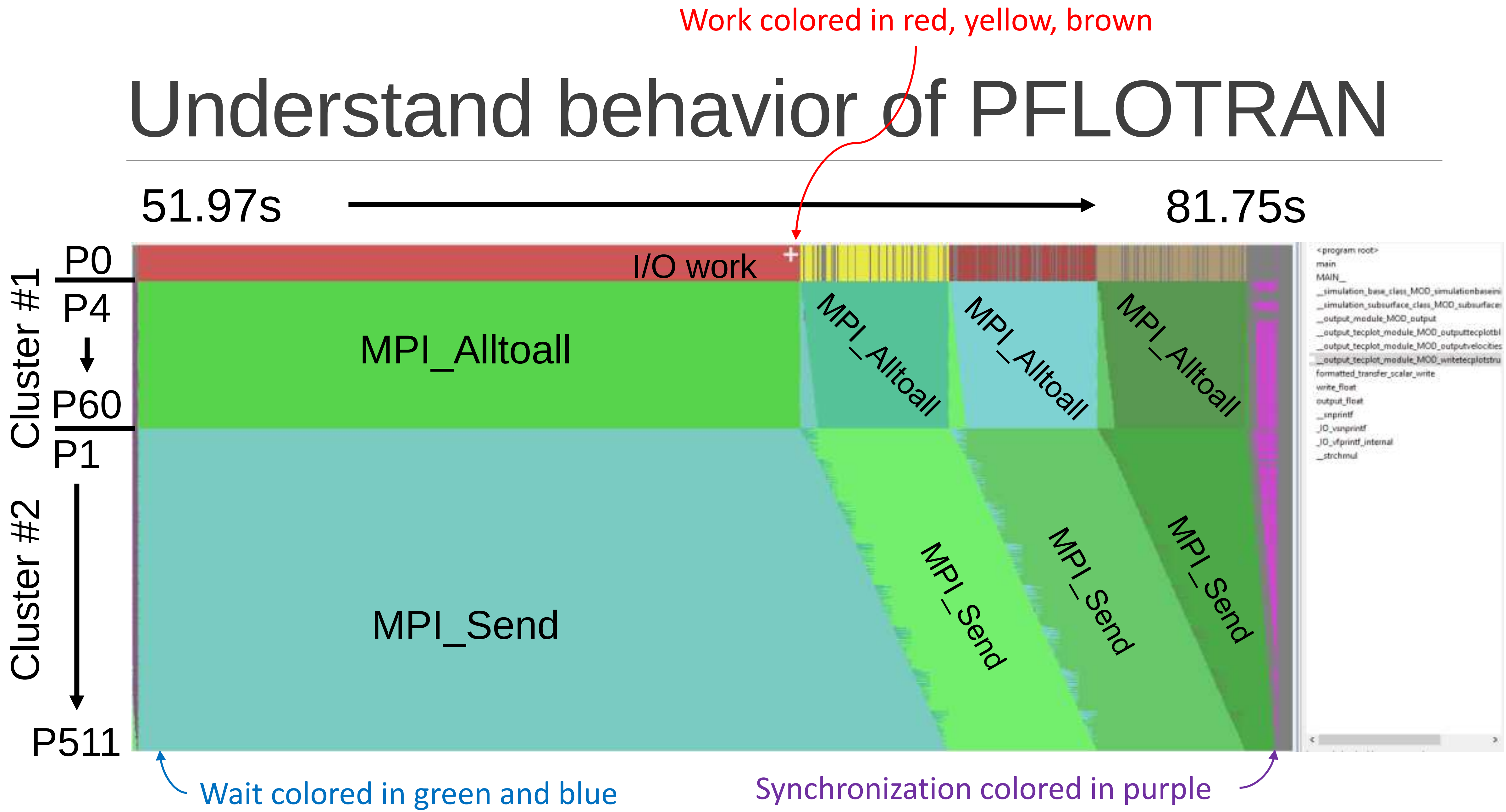
Understand behavior of PFLOTRAN



Understand behavior of PFLOTRAN

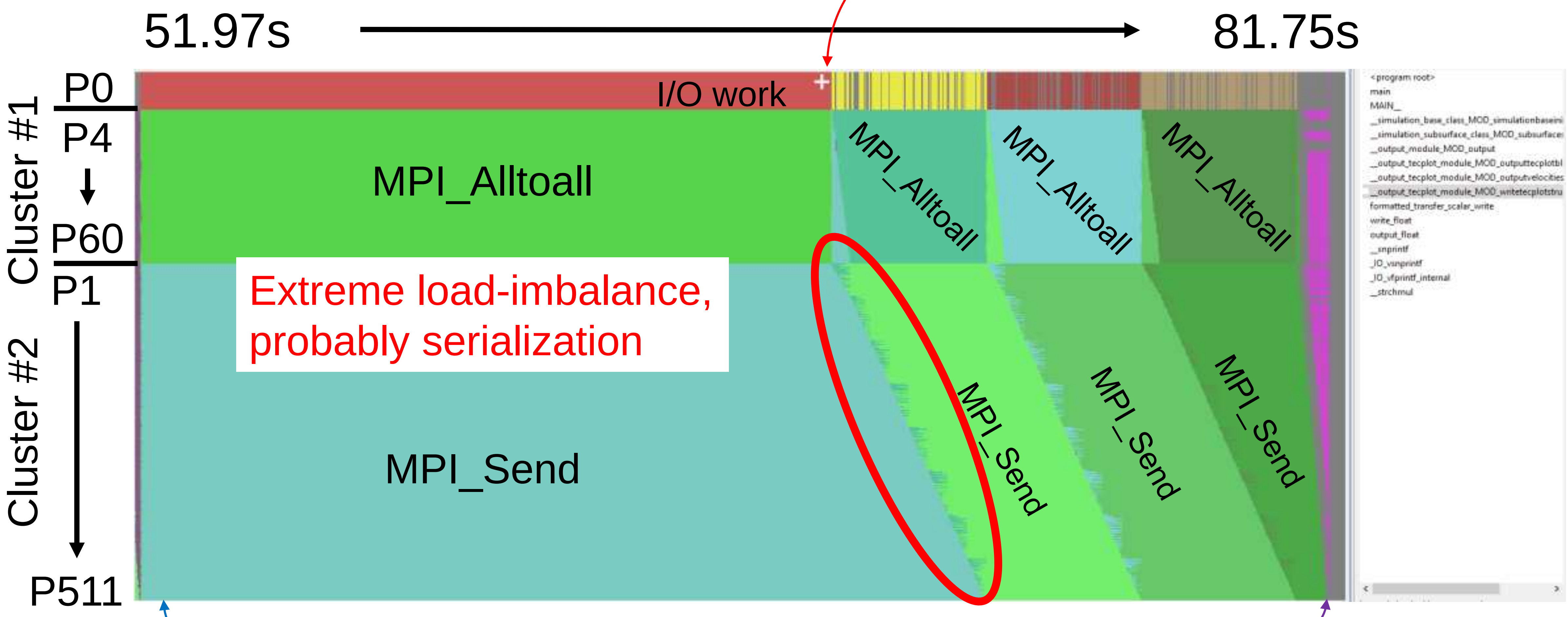


Understand behavior of PFLOTRAN



Understand behavior of PFLOTRAN

Work colored in red, yellow, brown



Wait colored in green and blue

Synchronization colored in purple

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 ← Symptom of serialization
- 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

Cause of serialization

Symptom 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