NoiseMiner:
An Algorithm for Scalable Automatic Computational Noise and Software Interference Detection

Isaac Dooley, Chao Mei, Laxmikant Kale

6th Annual Workshop on
Charm++ and its Applications
May 1 2008

For details, see our paper in the HIPS workshop at IPDPS 2008

NoiseMiner: An Algorithm for Scalable Automatic Computational Noise and Software Interference Detection

NoiseMiner: An Algorithm for Scalable Automatic Computational Noise and Software Interference Detection

Isaac Dooley*, Chao Mei, Laxmikant Kale
University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, IL USA
{idooley2,chaomei2,kale}@uiuc.edu

Abstract

This paper describes a new scalable stream mining algorithm called NoiseMiner that analyzes parallel application traces to detect computational noise, operating system interference, software interference, or other irregularities in a parallel application's performance. The algorithm detects these occurrences of noise during real application runs, whereas standard techniques for detecting noise use carefully crafted test programs to detect the problems.

This paper concludes by showing the output of NoiseMiner for a real-world case in which 6 ms delays, caused by a bug in an MPI implementation, significantly limited the performance of a molecular dynamics code on a new supercomputer.

1. Introduction

Automated analysis techniques are critical when scaling applications to hundreds of thousands of processors. A single person cannot possibly monitor online data streams from an application, or even look through trace logs for all processors in a reasonable amount of time. Therefore performance analysis techniques are useful when they can scale to such volumes of data. One technique for scalably analyzing large volumes of data either online or offline is *stream mining*. Stream mining refers to any data mining algorithm that only makes a single linear pass through its input data, while maintaining a synopsis of the data. The synopsis must have bounded memory requirements, and at any point in the processing of the stream, it can be converted into a useful report. This paper proposes and provides results from a novel scalable stream mining algorithm that detects computational noise that interferes with the performance of parallel applications on large clusters.

2. Motivating Problem

In January 2007, developers discovered that the widely used molecular dynamics code NAMD[15, 16] performed poorly on a brand new large supercomputer cluster named Lonestar at the Texas Advanced Computing Center. After manual analysis of post-mortem performance trace visualizations, they discovered that some events on a specific processor would take 6 ms longer than expected. The 6 ms delays were not limited to a single portion of the computation, but rather occurred in various parts of the computation. Eventually these developers determined that the culprit was a bug in the MPI library whereby some MPI calls would take around 6 milliseconds. The problem was finally fixed by a software update to the MPI library.

The histogram in Figure 1 shows the durations of events from a NAMD run on the new Lonestar cluster, with hundreds of events each taking 6 ms longer than expected. The left portion of the histogram represents events with expected durations less than 1 ms. In the middle of the same histogram are a a group of events with exceptionally long durations. This paper describes an automated scalable algorithm that detects such exceptionally long events. Automatically detecting such exceptional events can increase the productivity of application developers while porting applications to new machines. The method proposed in this paper can detect multiple types of noise even though this paper just shows its use in the presence of a single predominant source as seen in Figure 1.

3. The Problem of Interference

Historically, systems have been plagued with operating system (OS) daemons or other processes that interrupt par-

*This work was funded in part by the DOE High Performance Computer Science Fellowship Program.

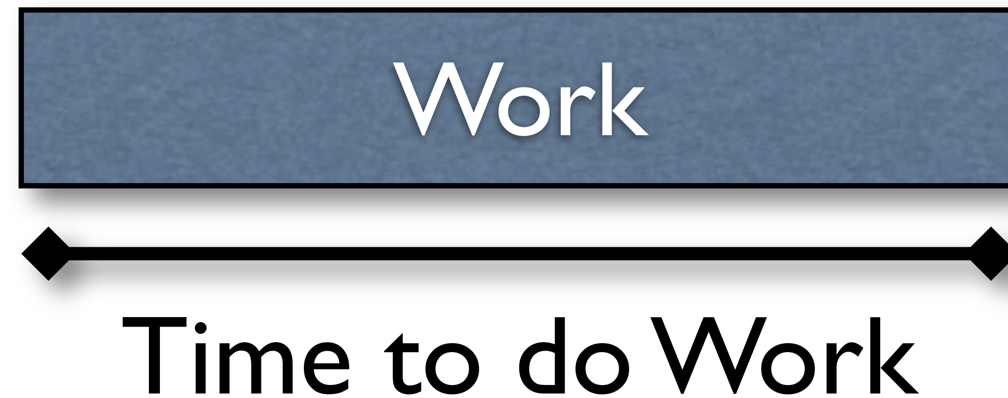
Motivation

A run of NAMD on a new cluster at a supercomputing center was performing poorly.

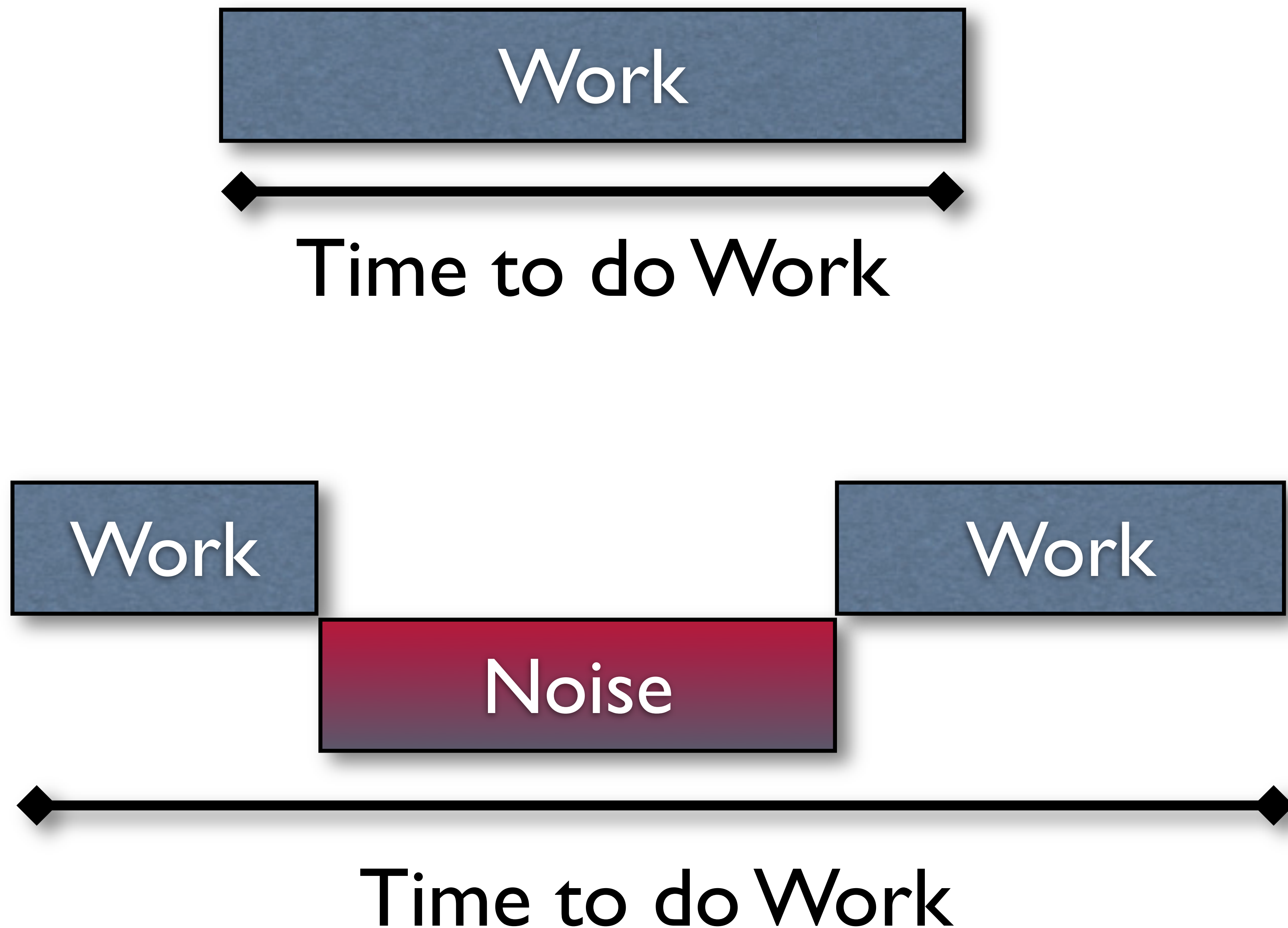
Why?

Timely manual analysis of trace logs for simplified runs showed that some portions of the computation were taking 6ms longer than expected. (Due to a bug in the MPI library)

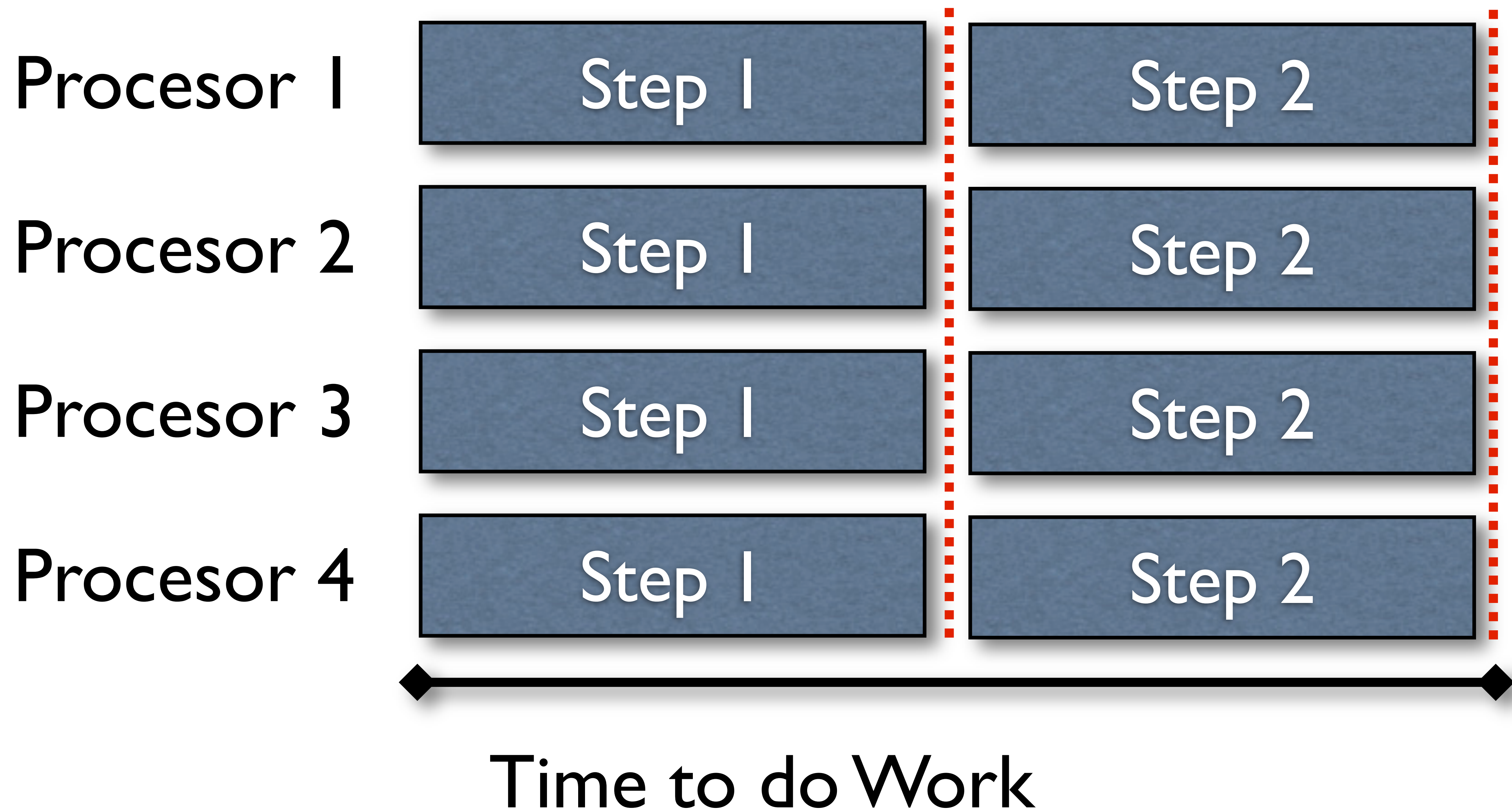
Computational Noise



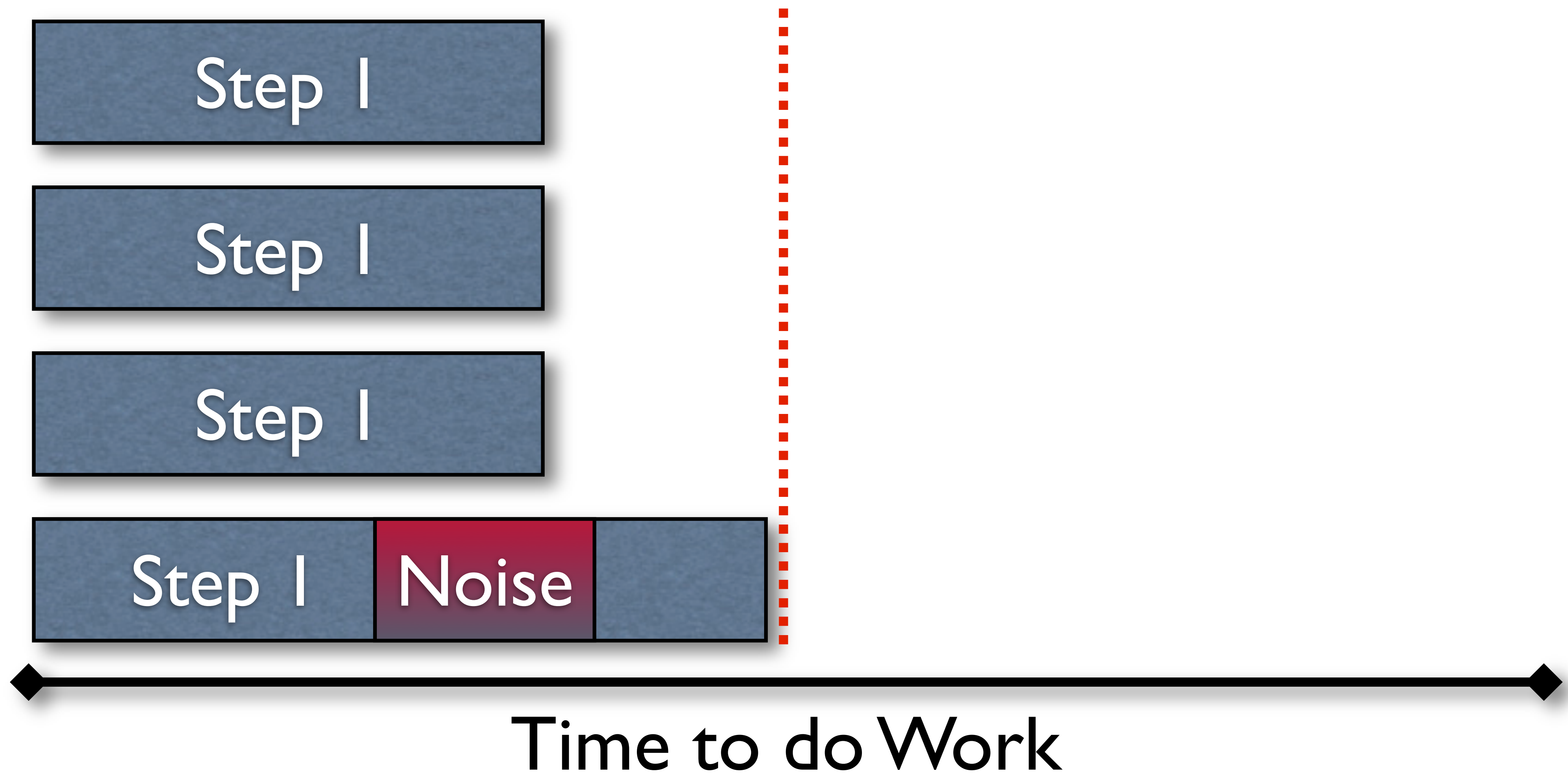
Computational Noise



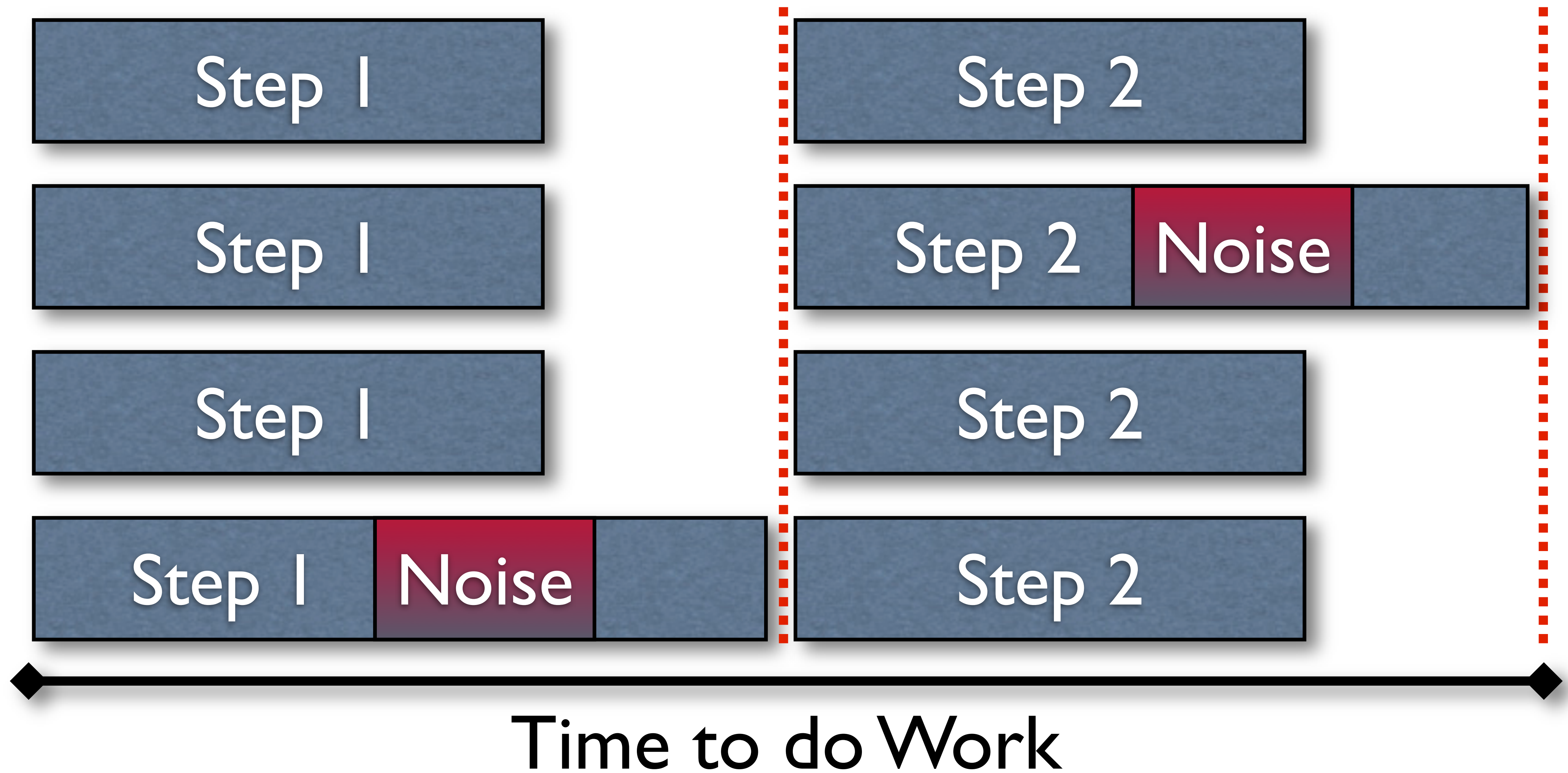
Noise In Parallel



Noise In Parallel



Noise In Parallel



How To Detect Noise?

Manually analyze logs

Use a micro-benchmark

<u>Automated analysis tools</u>

Our Contributions

NOISEMINER is the first automated tool* for detecting computational noise in complex real-world parallel applications.

NOISEMINER reduces the time to identify potential symptoms of computational noise.

* A sequential pattern mining algorithm was proposed by Vahid Tabatabaee, Jeffrey K. Hollingsworth in their SC07 paper Automatic Software Interference Detection in Parallel Applications. To the best of my knowledge the tools have not been publicly released.

Assumptions

A parallel program's execution can be decomposed into a set of events of various types.

The events have durations.

Events of the same type should have roughly similar durations.

Events with abnormally long durations are potential symptoms of noise.

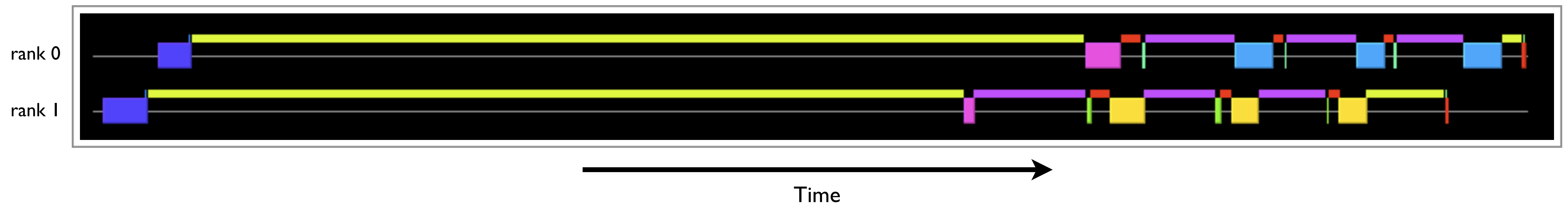
Events in Charm++

Two types of events:

- Entry Method Invocations
- When neither idle nor executing an entry method.

Events in MPI

```
int main(int argc, char ** argv){
        ...
        MPI_Init( &argc, &argv );
        ...
        MPI_Barrier(MPI_COMM_WORLD);
        for(int i=0;i<NITER;i++){
          if(rank % 2 == 0) {
            MPI_Send(..., rank+1, ...);
            MPI_Recv(...,rank+1, ...);
          } else {
            MPI_Recv(..., rank-1, ...);
            MPI_Send(..., rank-1, ...);
          }
          do_work();
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
      return 0;
}
```



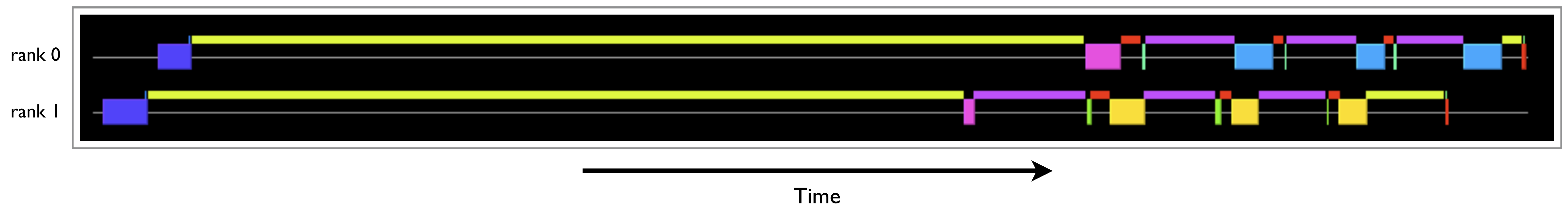
Time

Events in MPI

```
int main(int argc, char ** argv){
        ...
        MPI_Init( &argc, &argv );
        ...
        MPI_Barrier(MPI_COMM_WORLD);
        for(int i=0;i<NITER;i++){
          if(rank % 2 == 0) {
            MPI_Send(..., rank+1, ...);
            MPI_Recv(...,rank+1, ...);
          } else {
            MPI_Recv(..., rank-1, ...);
            MPI_Send(..., rank-1, ...);
          }
          do_work();
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
      return 0;
}
```

Events are regions between consecutive MPI_* calls

Events are distinguished into types based on the preceding MPI Call(source location)



Time

Events in MPI

rank 0

MPI_Recv(...,rank+1, ...);

Preceding MPI Call

do_work();

rank 1

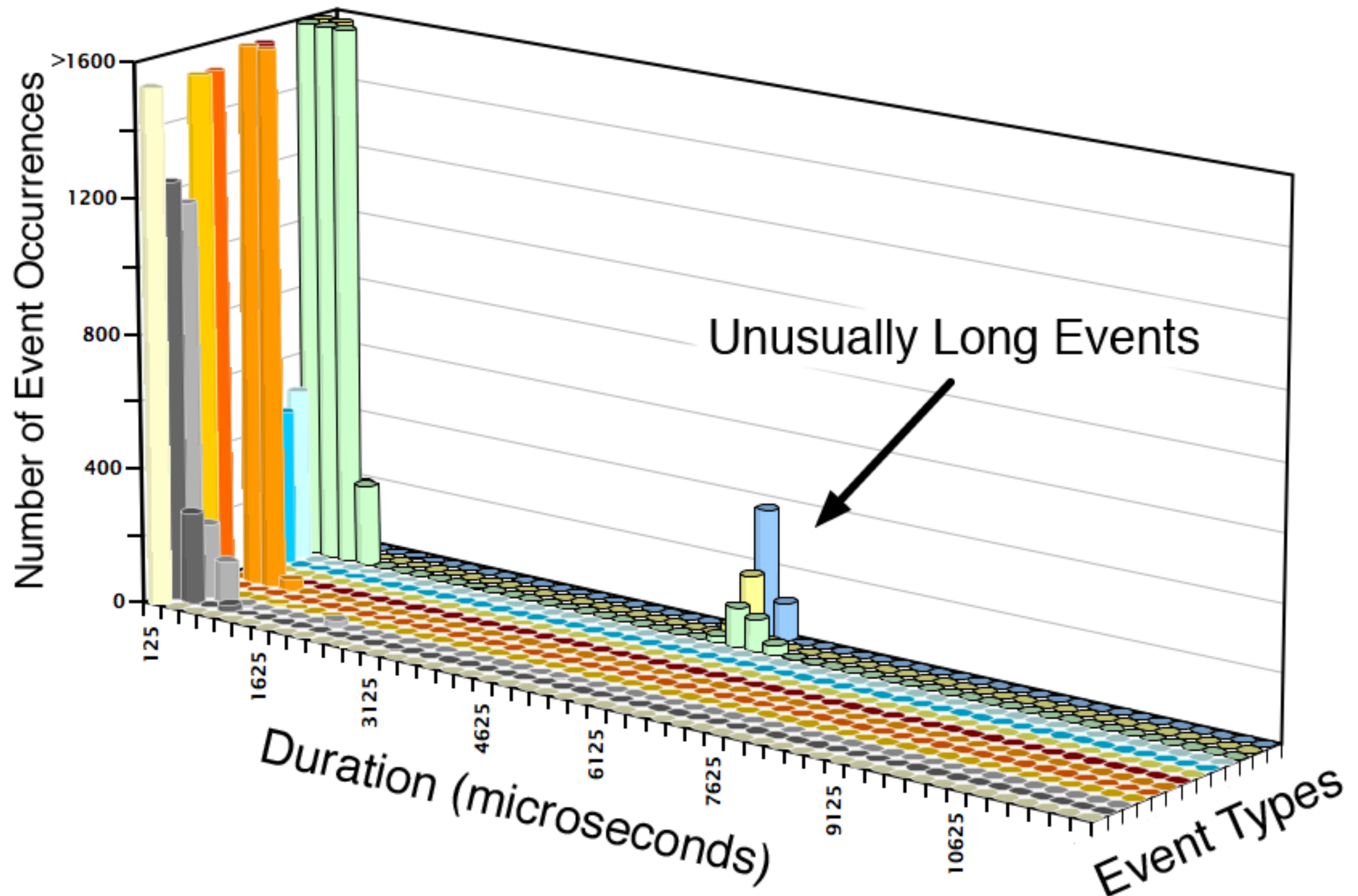
MPI_Send(..., rank-1, ...);

Preceding MPI Call

do_work();

Time

Noise Occurrences



NAMD with MPI bug

Problem

Quickly and scalably detect the abnormally long events in a parallel program's execution.

Then provide a useful analysis of the important events.

Solution: NoiseMiner

A Stream Mining Algorithm:

- Single pass through application traces
- Bounded memory usage
- Can be used online or offline(with traces)
- Fast

Implementation

Noise Miner is included in Projections

Projections currently supports logs from:

Charm++, AMPI -tracemode projections

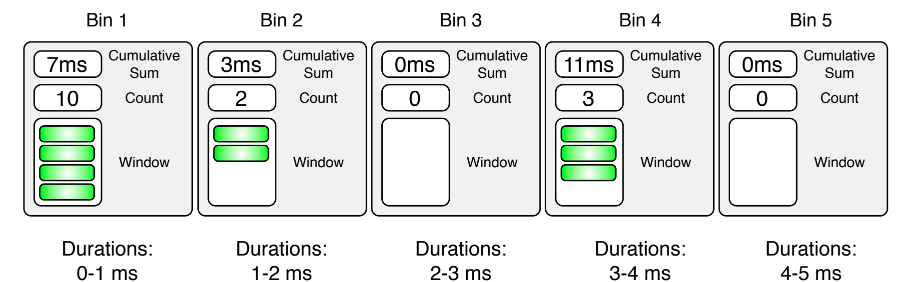
MPI linked with PMPI_Projections*

* PMPI_Projections module in Charm CVS

NoiseMiner Overview

1. Maintain a synopsis of events seen so far (inserting events into windowed histograms)
2. Generate a report:
   1. Cluster the histogram bins into groups
   2. Determine a noise duration for each group
   3. Filter out less-important groups
   4. Merge groups across processors into Noise Components
   5. Display the most important Noise Components

NOISEMINER Synopsis



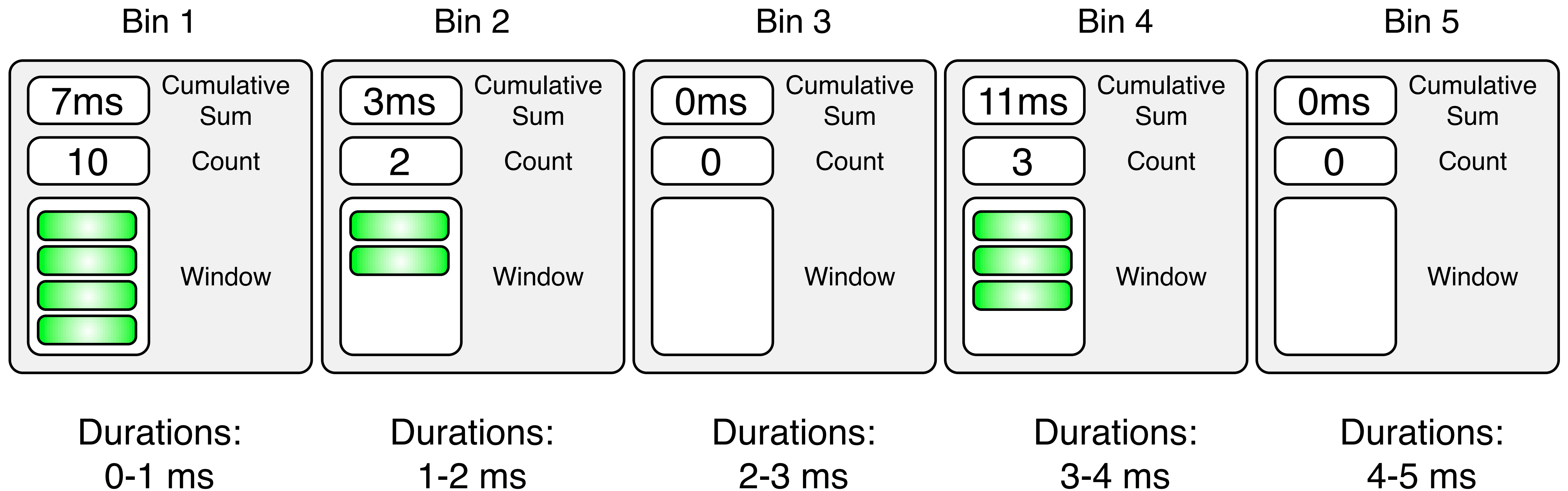
For each processor:

- A histogram for each event type.
- Each bin represents a range of event durations.

Each histogram bin contains:

- A window of a fixed number of recent events
- Count of events
- Average duration of events

Example Histogram

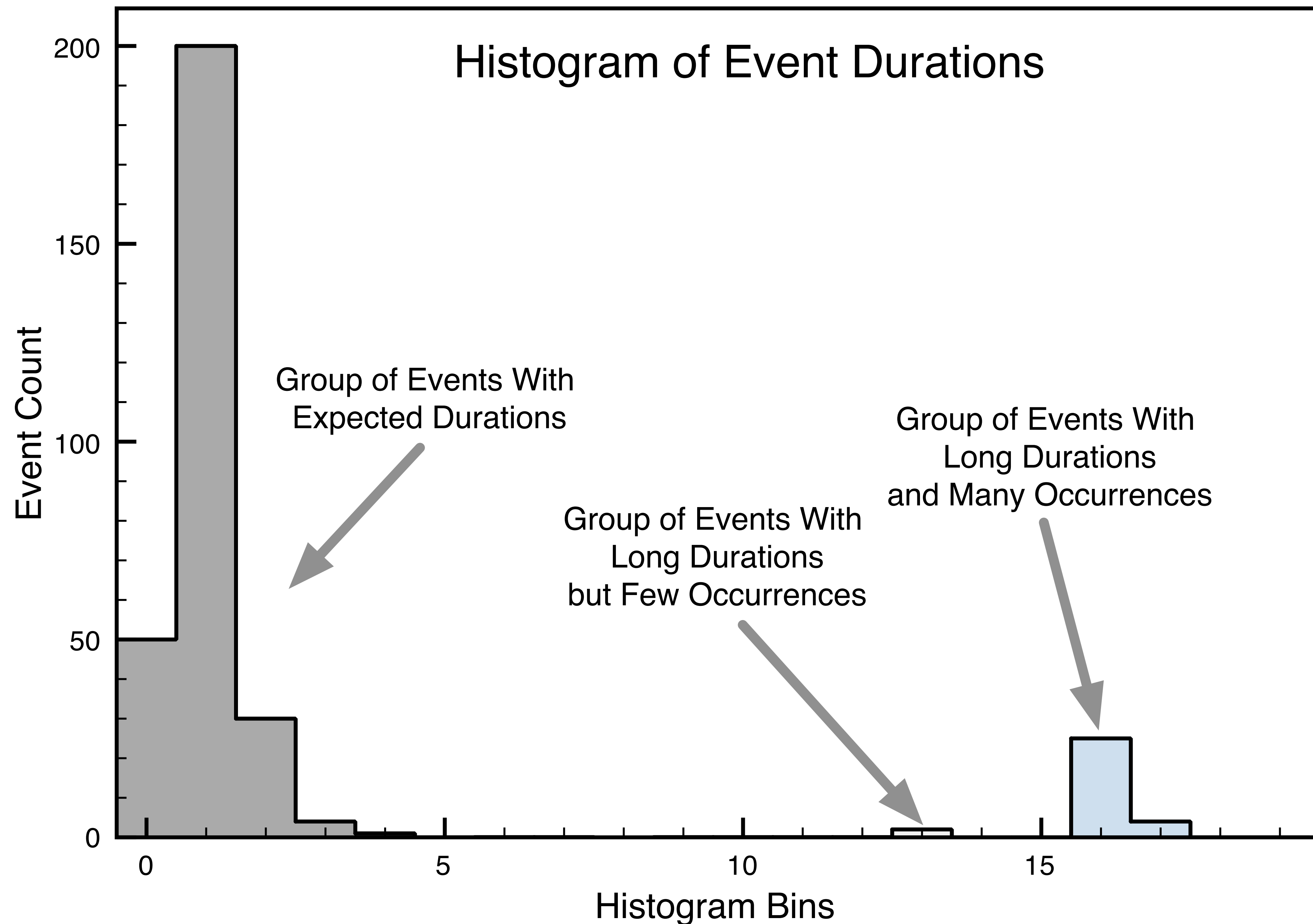


NoiseMiner Report

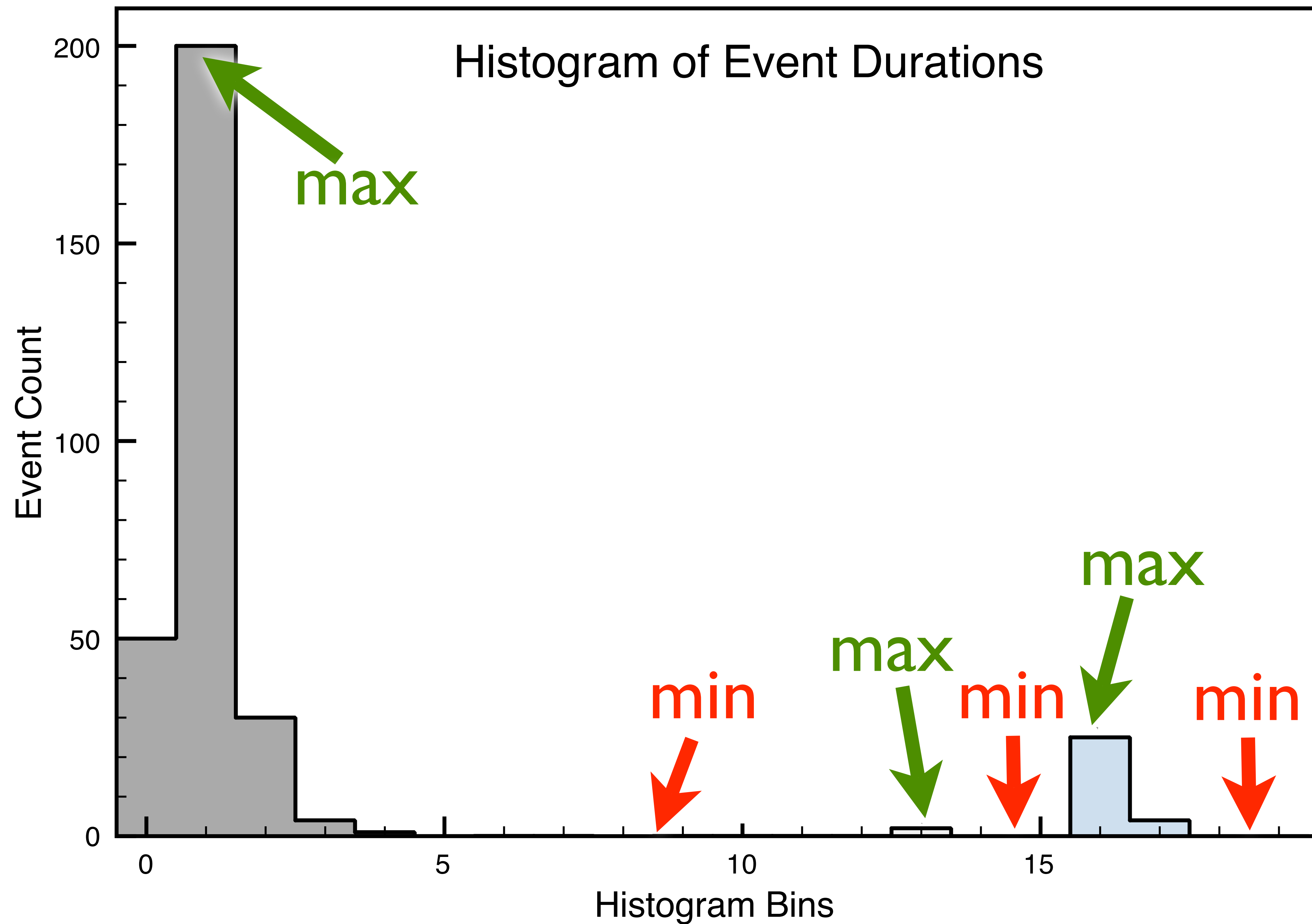
Now we have a Synopsis:
Histogram for each event type on each processor

Lets generate a Report!

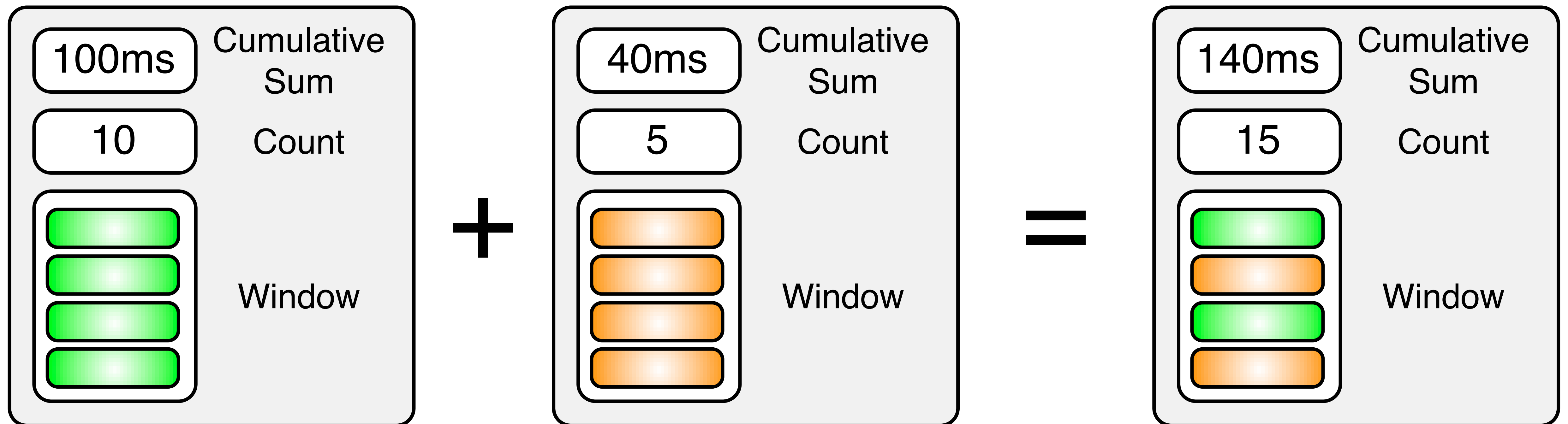
Step 1: Group the Bins



Step 1: Group the Bins



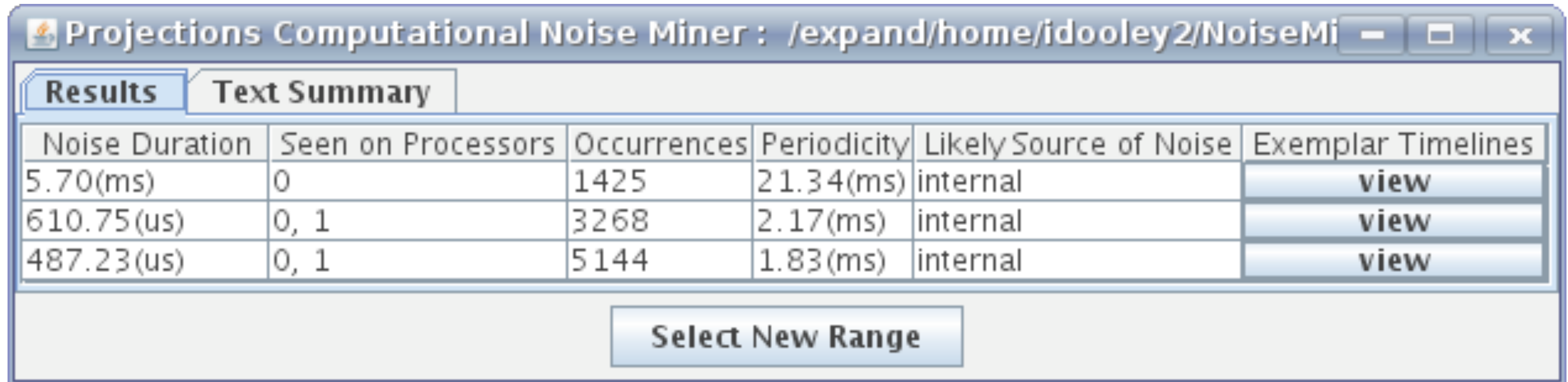
Merging Bins to Form Groups

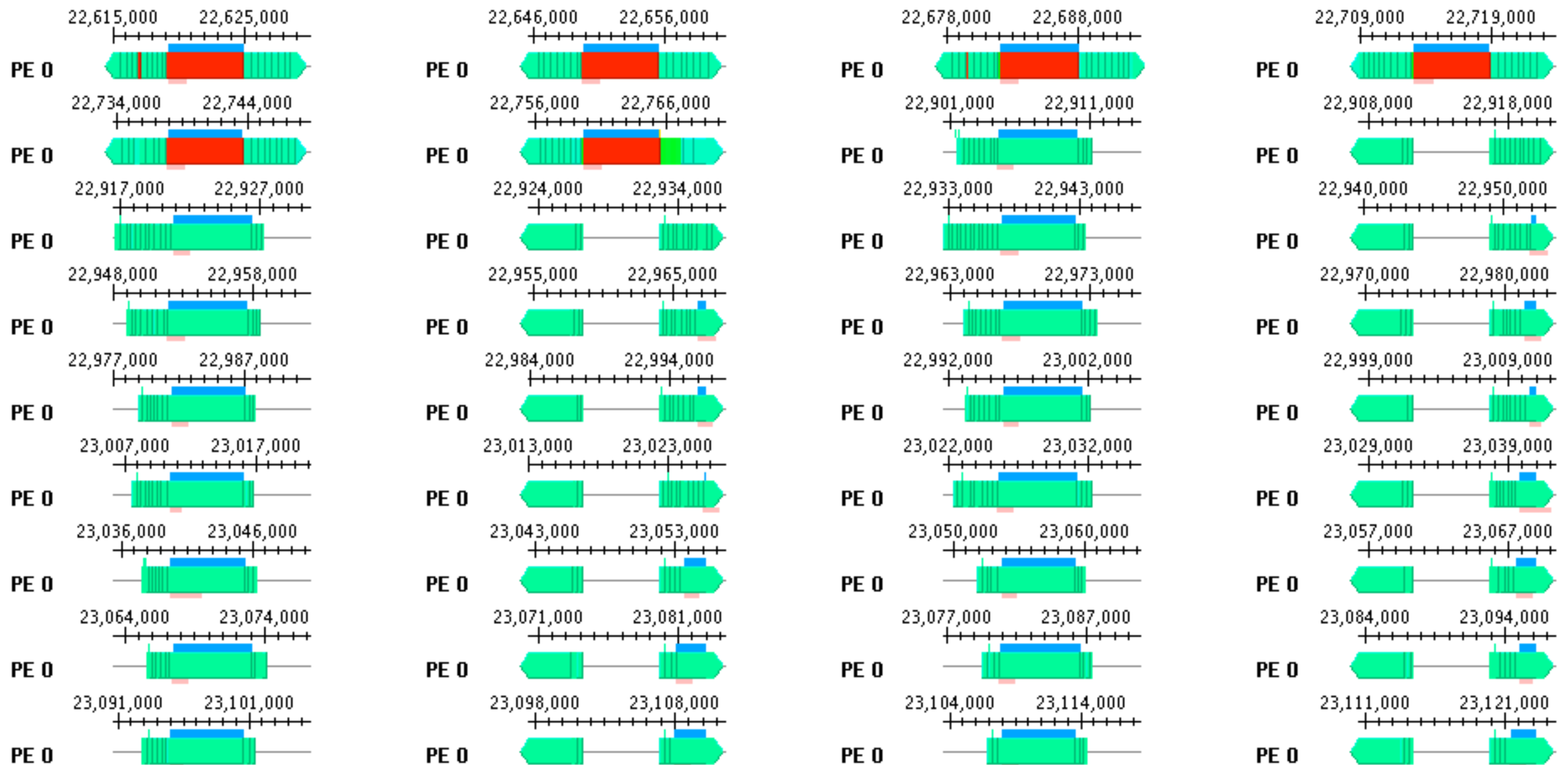


Step 2:
Noise Components

1. Take each group and normalize its duration by the expected duration.
2. Cluster all groups across processors
   Iteratively merge similar groups based on a similarity function
3. The resulting clustered groups are called Noise Components
4. Filter and Display important Noise Components

NoiseMiner Views:





NoiseMiner Exemplars
Below are 36(out of 1425) mini-timelines that show a selected set of exemplar regions where extraordinarily long events occurred. In the center of each is the stretched event. Each stretched event was approximately 5.70(ms) longer than other entry methods of the same type(but not necessarily on the same object). Each timeline does not have the same scale, so direct comparisons are meaningless.
22,615,000 22,625,000
22,646,000 22,656,000
22,678,000 22,688,000
22,709,000 22,719,000
PE 0
22,734,000 22,744,000
22,756,000 22,766,000
22,901,000 22,911,000
22,908,000 22,918,000
22,917,000 22,927,000
22,924,000 22,934,000
22,933,000 22,943,000
22,940,000 22,950,000
22,948,000 22,958,000
22,955,000 22,965,000
22,963,000 22,973,000
22,970,000 22,980,000
22,977,000 22,987,000
22,984,000 22,994,000
22,992,000 23,002,000
22,999,000 23,009,000
23,007,000 23,017,000
23,013,000 23,023,000
23,022,000 23,032,000
23,029,000 23,039,000
23,036,000 23,046,000
23,043,000 23,053,000
23,050,000 23,060,000
23,057,000 23,067,000
23,064,000 23,074,000
23,071,000 23,081,000
23,077,000 23,087,000
23,084,000 23,094,000
23,091,000 23,101,000
23,098,000 23,108,000
23,104,000 23,114,000
23,111,000 23,121,000

Scalability

1. Inserting an event into synopsis is $O(1)$
2. Memory usage is $O(1)$ on each processor
3. Combining groups across processors is scalable, and can be performed in parallel as a user defined reduction in a tree in $O(\log(p))$ time.

Conclusions

NOISEMINER has helped detect computational noise in real-world complicated application runs.

NOISEMINER is scalable, fast, and can analyze arbitrarily large application runs.