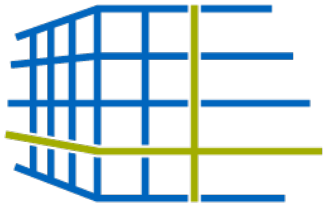
VI-HPS

Performance Analysis Basics

Michael Gerndt (gerndt@in.tum.de)
Felix Wolf (f.wolf@grs-sim.de)

October 31 2010

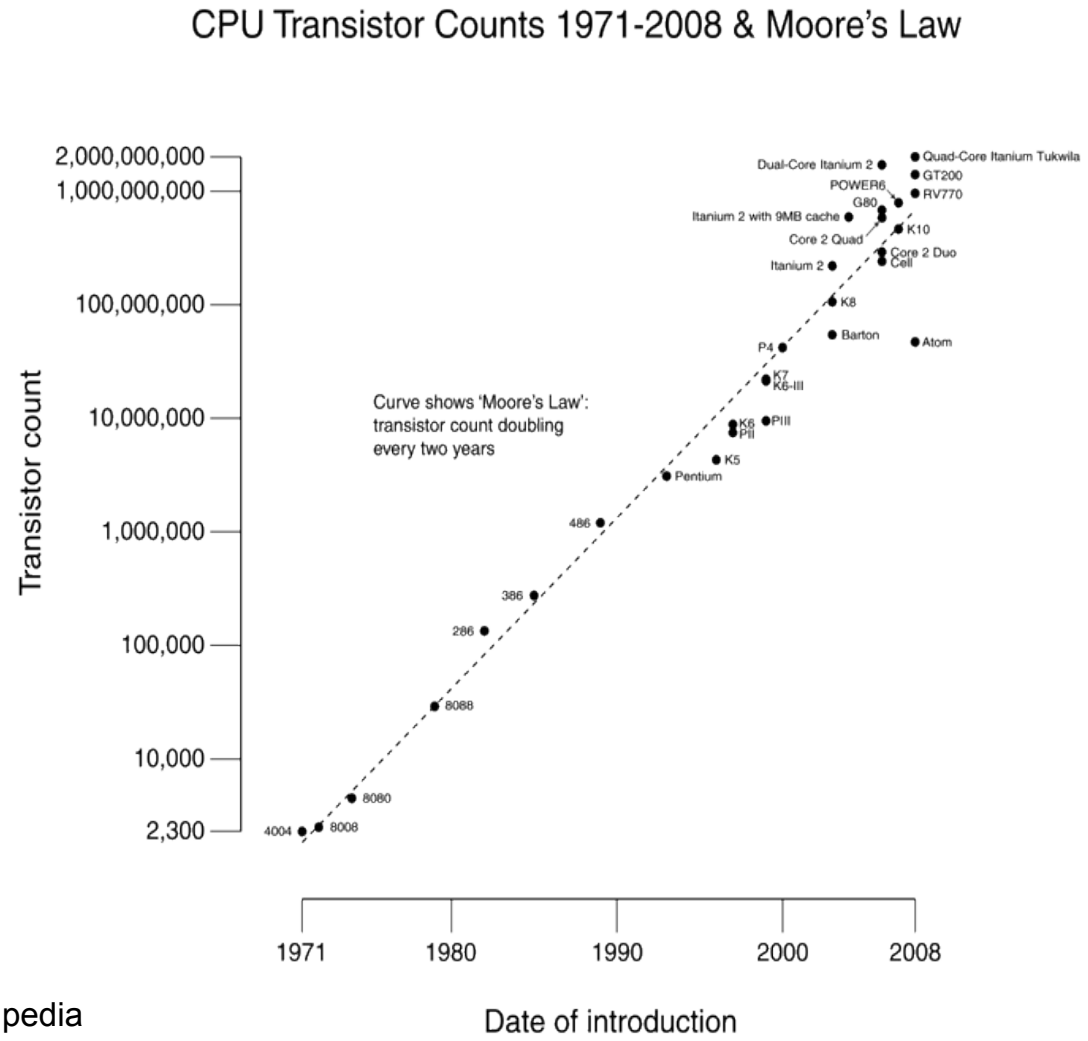


Munich Centre for
Advanced Computing

Overview

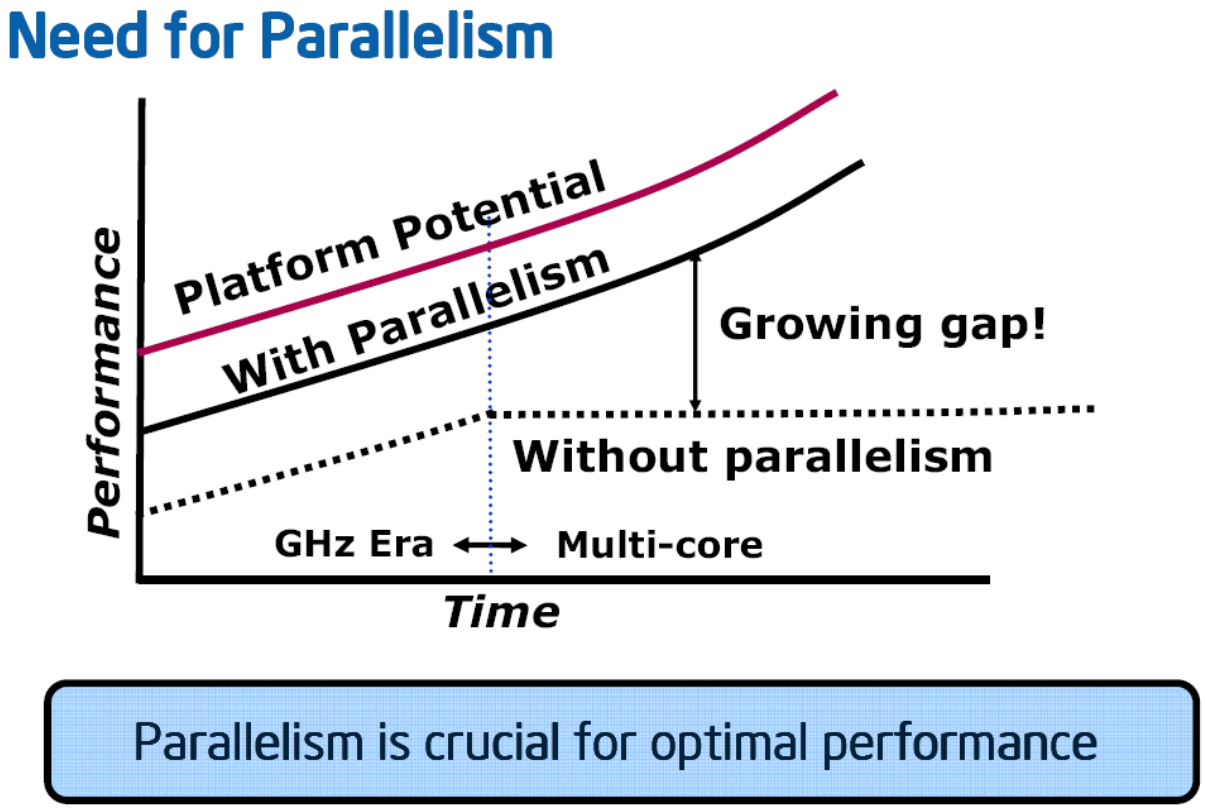
1. Motivation
2. Performance analysis basics
3. Measurement techniques
4. Analysis techniques
5. The tools

- Why performance analysis at all? Moore's Law still in charge



Source: Wikipedia

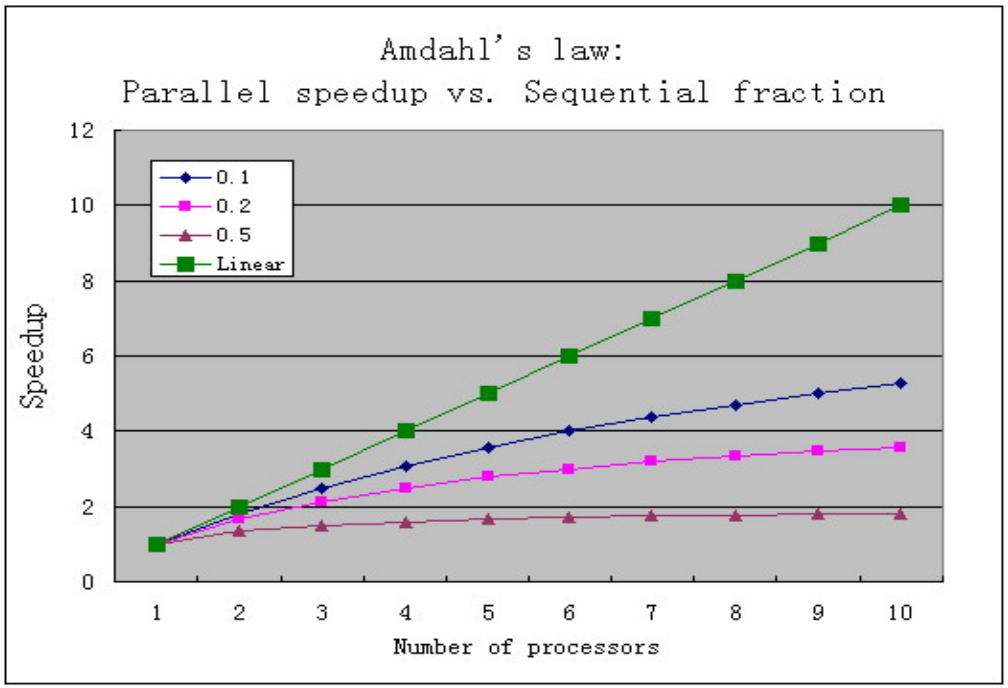
- But *Free Lunch* is over



Motivation

- Two parallel architectures
  - Shared memory
    - OpenMP
    - Pthreads
    - Threading Building Blocks
    - …
  - Distributed memory
    - MPI
    - Partitioned Global Address Space (e.g., UPC)
    - Sockets
- Today also heterogeneous systems
  - CUDA, OpenCL
  - Not covered here

- Amdahl's Law:
  - The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.



Source: Wikipedia

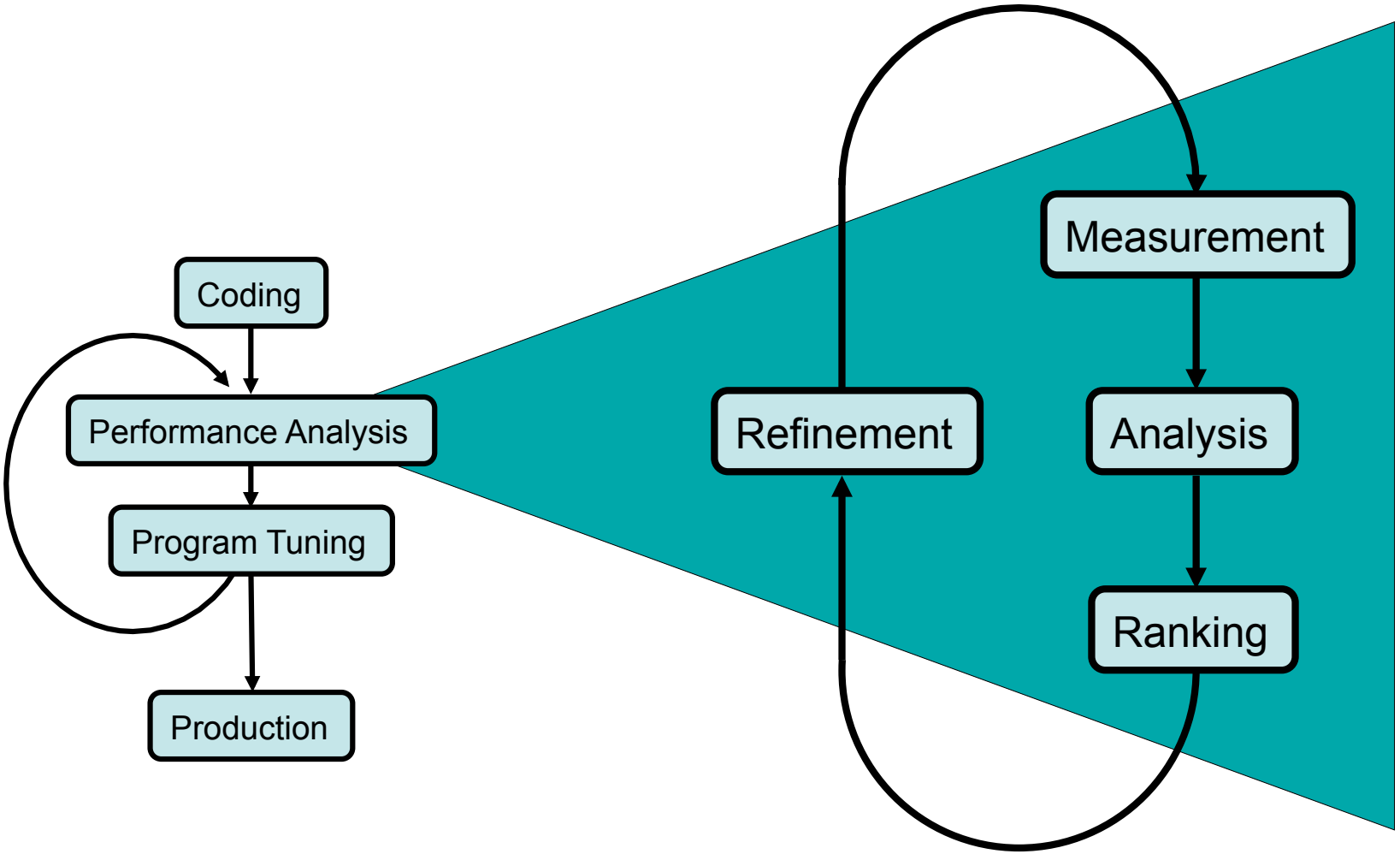
Motivation

- High complexity in parallel and distributed systems
  - Application
    - Algorithm, data structures
  - Parallel programming interface
    - Compiler, parallel libraries, communication, synchronization
  - Operating system
    - Process and memory management, IO
  - Hardware
    - CPU, memory, network

Motivation

- Factors which influence performance of parallel programs
  - "Sequential" factors
    - Computation
    - Cache and memory
    - Input / output
  - "Parallel" factors
    - Communication (message passing)
    - Threading
    - Synchronization
    - Parallel I/O

Performance Analysis Process



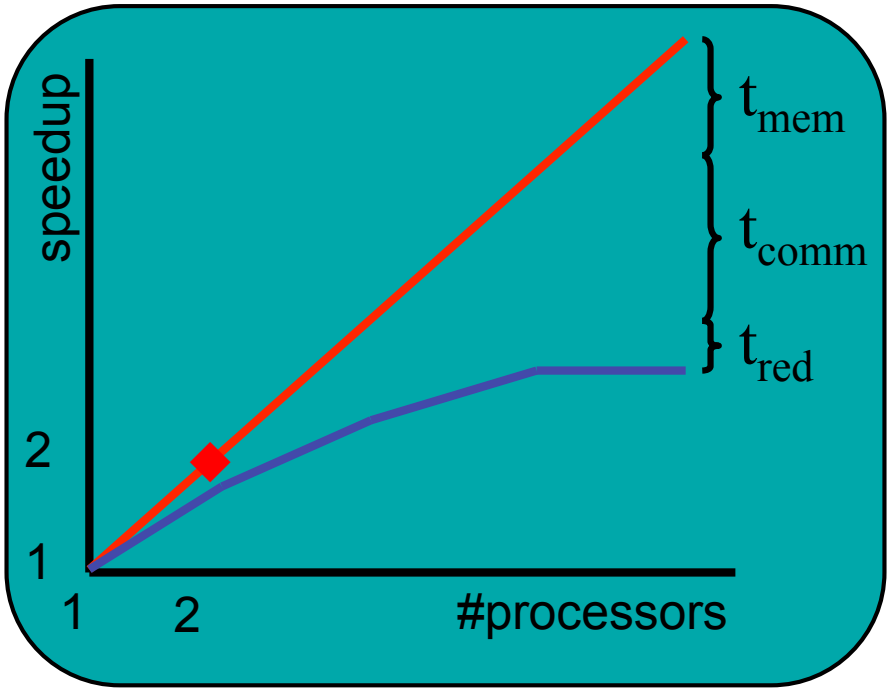
- **Performance analysis** determines the performance on a given machine.
- **Performance prediction** allows to evaluate programs for a hypthetical machine. It is based on:
  - runtime data of an actual execution
  - machine model of the target machine
  - analytical techniques
  - simulation techniques
- **Benchmarking** determines the performance of a computer system on the basis of a set of typical applications.

1. Motivation
2. Performance analysis basics
3. Measurement techniques
4. Analysis techniques
5. The tools

- How to decide whether a code performs well:
  - Comparison of measured FLOPS with peak performance
  - Comparison with a sequential version

$$speedup(p) = \frac{t_s}{t_p}$$

  - Estimate distance to ideal time via overhead classes
    - t_{mem}
    - t_{comm}
    - t_{sync}
    - t_{red}
    - ...



- Scaled speedup
  - Problem size grows with machine size

$$\text{scaled\_speedup}(p) = \frac{t_s(n_p)}{t_p(n_p)}$$

  - Speedup can not be defined as Time(1) / Time(p) for scaled up problem since time(1) is hard to measure and inappropriate
  - Insert performance=work/time in speedup formula gives

$$Speedup_{MC}(p) = \frac{Work(p)}{Time(p)} / \frac{Work(1)}{Time(1)}$$

$$= \frac{Work(p)}{Work(1)} / \frac{Time(p)}{Time(1)}$$

$$= \frac{Increase\ in\ Work}{Increase\ in\ Time}$$

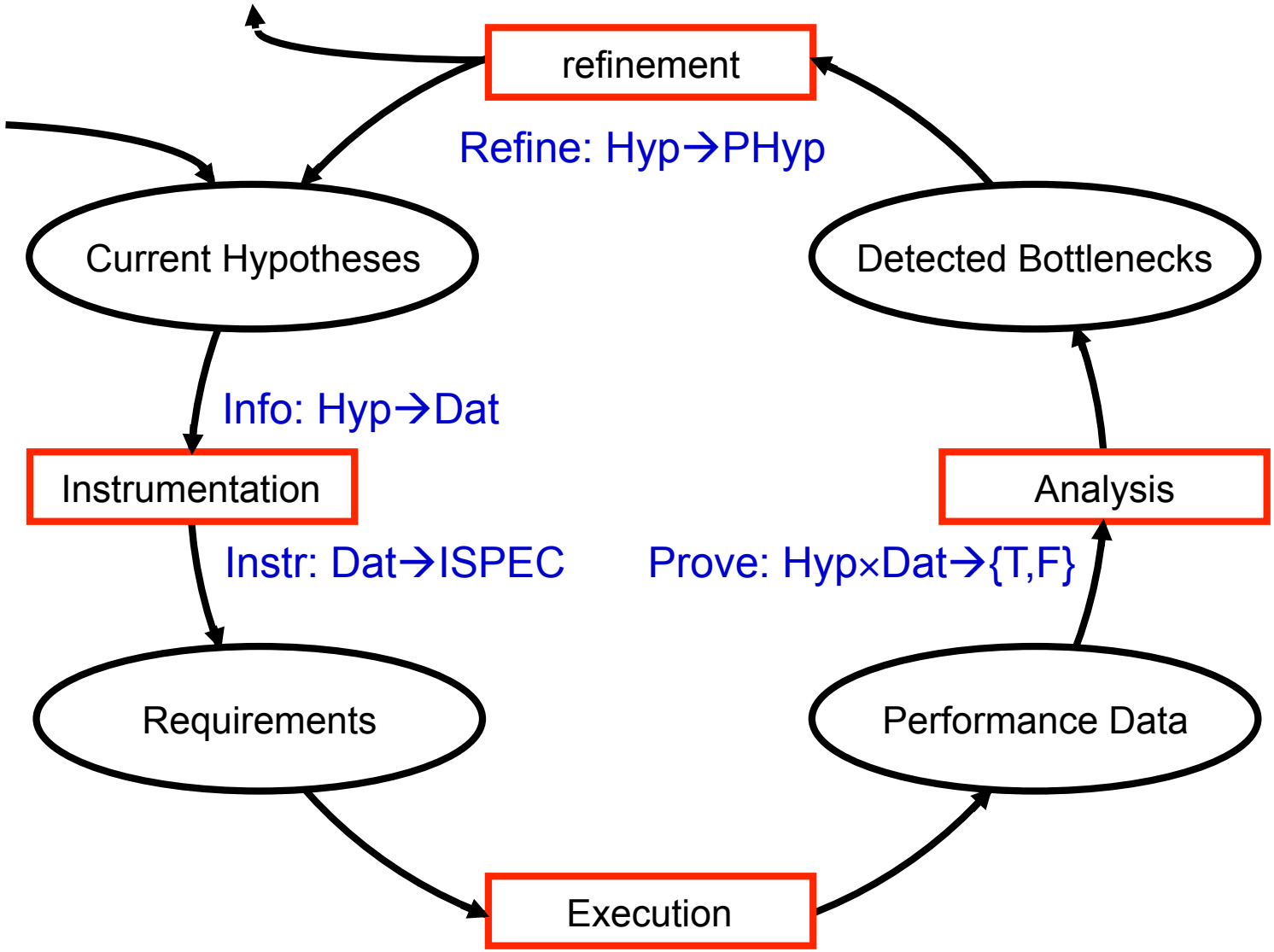
- Parallel efficiency: Percentage of ideal speedup

$$\text{efficiency}(p) = \text{speedup}(p) / p = \frac{t_s}{t_p * p}$$

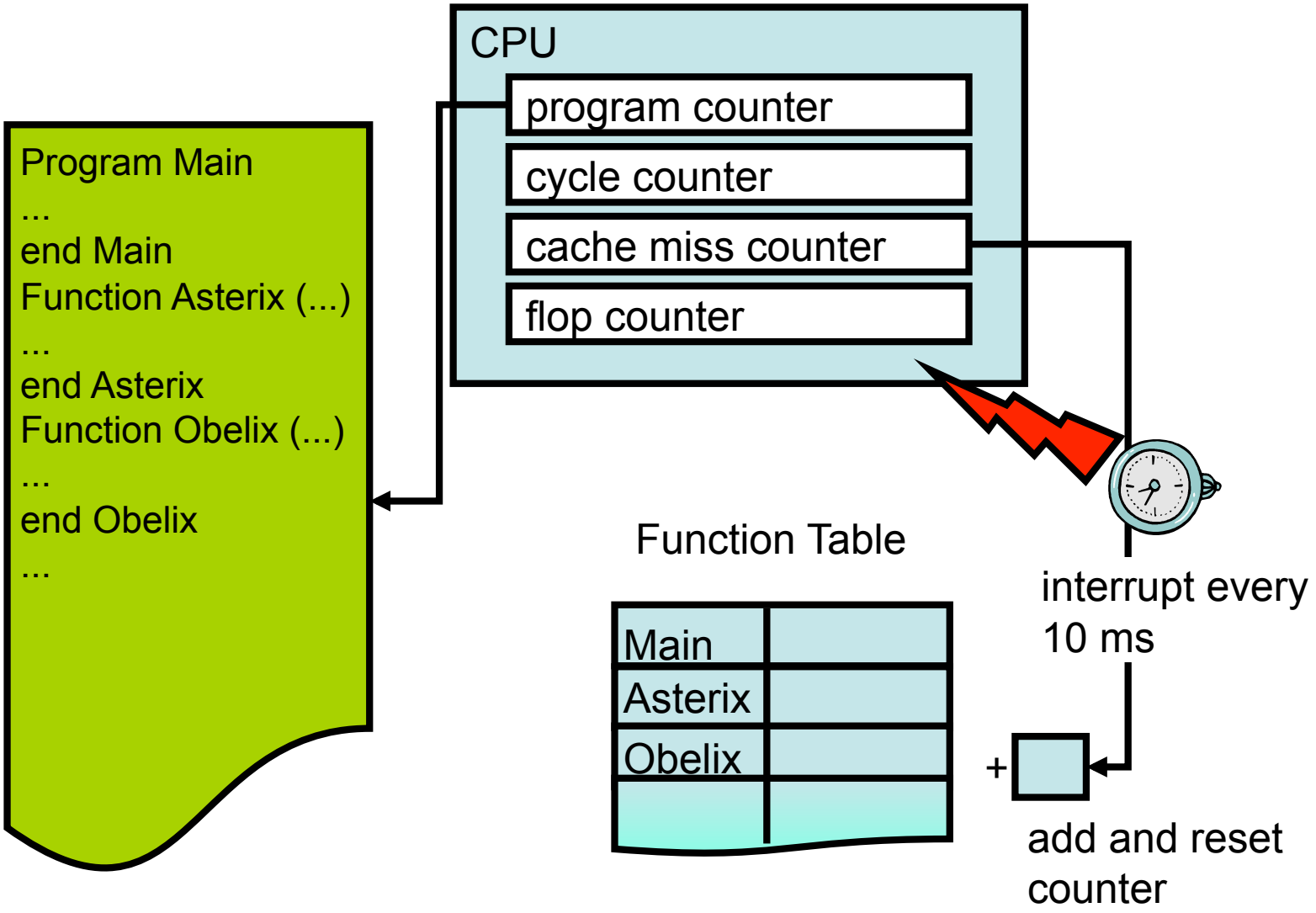
- Successful tuning is a combination of
  - right algorithms and libraries
  - compiler flags and directives
  - thinking!
- Measurement is better than guessing:
  - to determine performance problems
  - to validate tuning decisions and optimization
- Measurement should be repeated after each significant code modification and optimizations

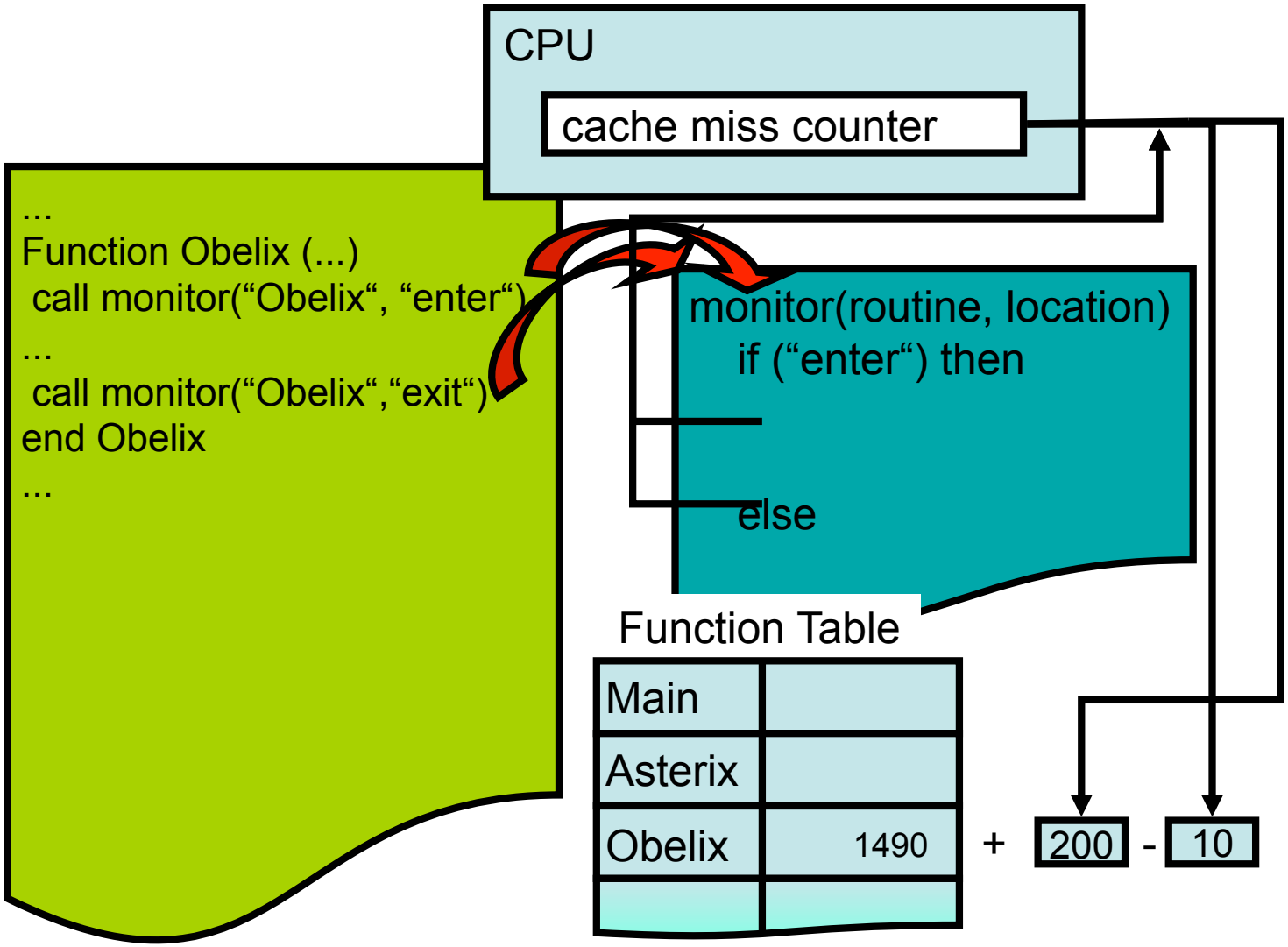
The Basics

- Do I have a performance problem at all?
  - Compare MFlops/MOps to typical rate
  - Speedup measurements
- What are the hot code regions?
  - Flat profiling
- Is there a bottleneck in those regions?
  - Single node: Hardware counter profiling
  - Parallel: Synchronization and communication analysis profiling
- Does the bottleneck vary over time or processor space?
  - Profiling individual processes and/or threads
  - Tracing
- Does the code behave similar for different configurations?
  - Analyze runs with different processor counts
  - Analyze different input configurations


refinement
Refine: Hyp→PHyp
Current Hypotheses
Detected Bottlenecks
Info: Hyp→Dat
Instrumentation
Analysis
Instr: Dat→ISPEC
Prove: Hyp×Dat→{T,F}
Requirements
Performance Data
Execution

- Event model of the execution
  - Events occur at a processor at a specific point in time
  - Events belong to event types
    - clock cycles
    - cache misses
    - remote references
    - start of a send operation
    - ...
- Profiling: Recording accumulated performance data for events
  - Sampling: Statistical approach
  - Instrumentation: Direct measurement
- Tracing: Recording performance data of individual events


CPU
program counter
cycle counter
cache miss counter
flop counter
Program Main
...
end Main
Function Asterix (...)
...
end Asterix
Function Obelix (...)
...
end Obelix
...
Function Table
Main
Asterix
Obelix
interrupt every
10 ms
+
add and reset
counter


CPU
cache miss counter
...
Function Obelix (...)
call monitor("Obelix", "enter")
...
call monitor("Obelix","exit")
end Obelix
...
monitor(routine, location)
if ("enter") then
else
Function Table
Main
Asterix
Obelix
1490
+
200
-
10

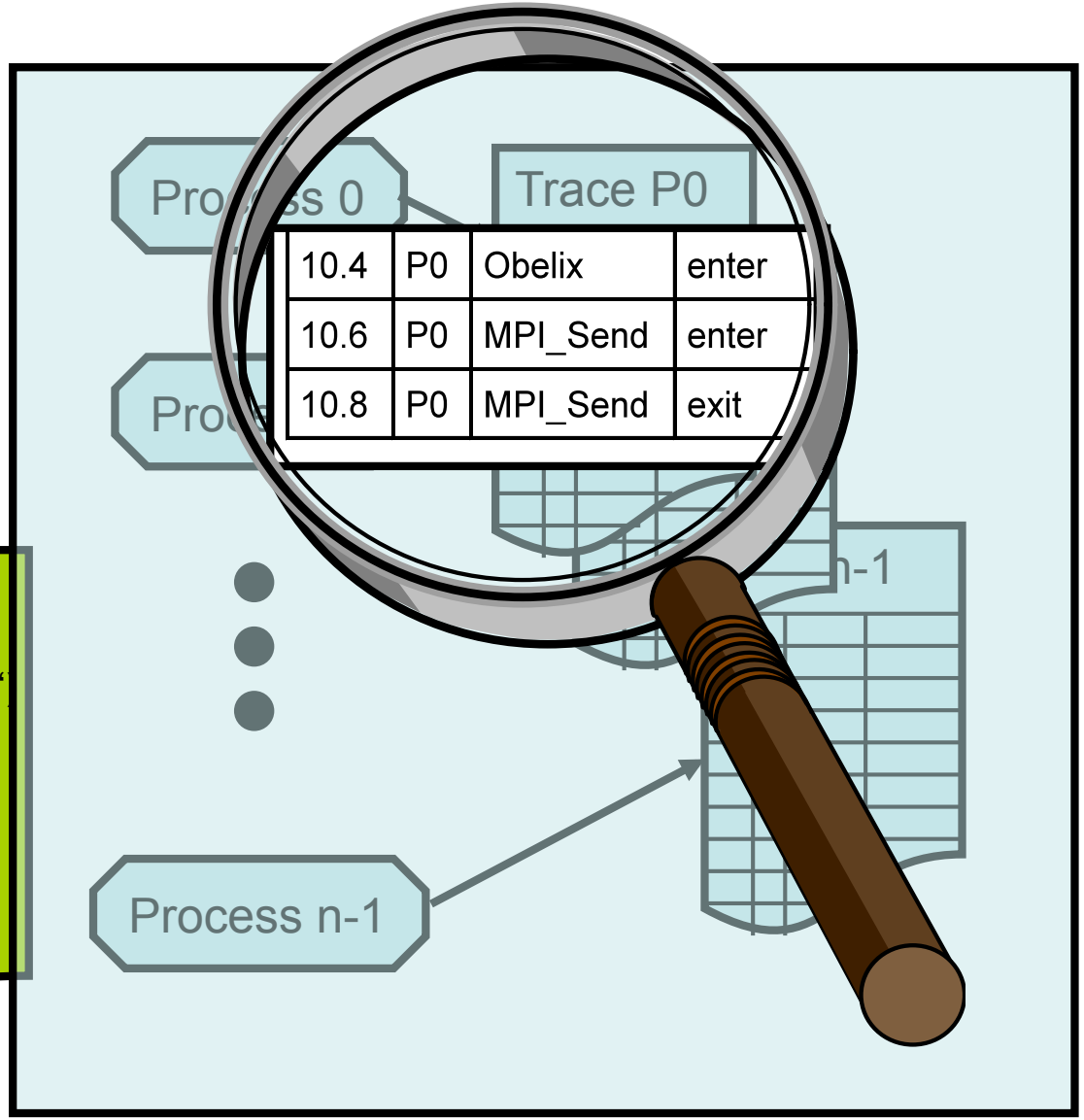
- Source code instrumentation
  - done by the compiler, source-to-source tool, or manually
    - + portability
    - + link back to source code easy
    - • re-compile necessary when instrumentation is changed
    - • difficult to instrument mixed-code applications
    - • cannot instrument system or 3rd party libraries or executables
- Object code instrumentation
  - „patching“ the executable to insert hooks (like a debugger)
    - • inverse pros/cons
  - Offline
  - Online

Tracing

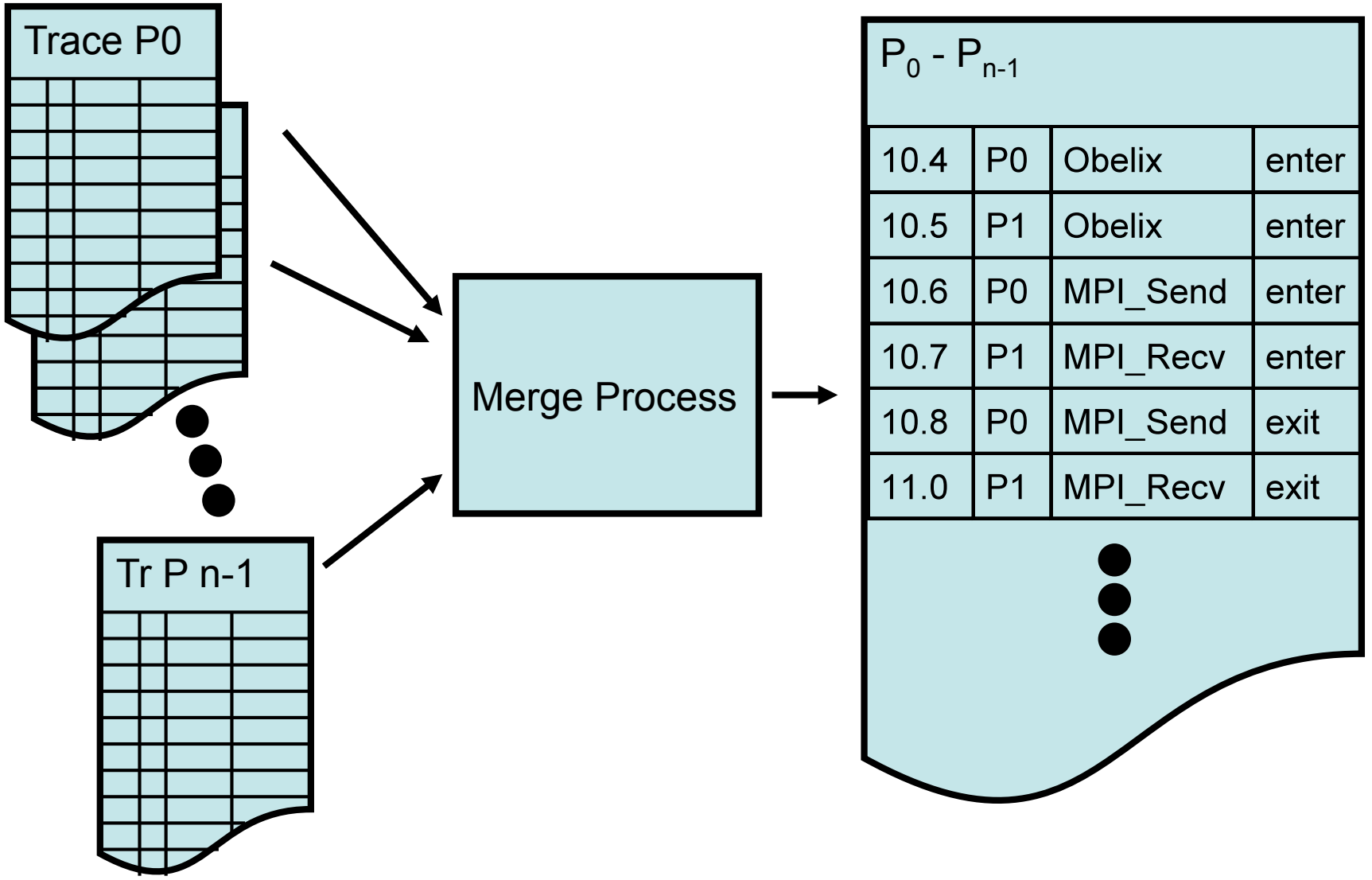
```
...
Function Obelix (...)
 call monitor(“Obelix“, “enter“)
...
 call monitor(“Obelix“,“exit“)
end Obelix
...
```

MPI Library

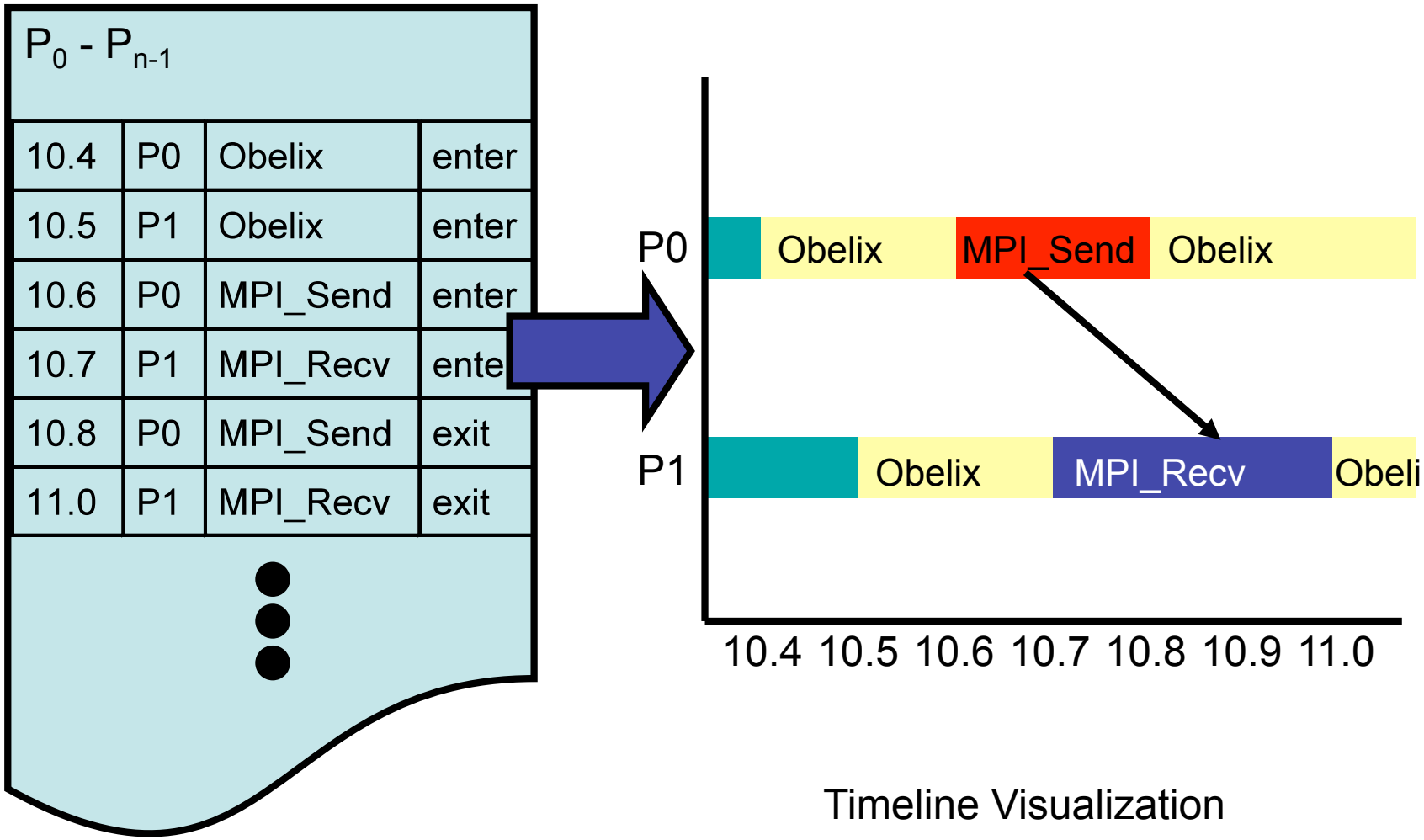
```
Function MPI_send (...)
 call monitor(“MPI_send“, “enter“)
 ...
 call PMPI_send(...)

 call monitor(“MPI_send“,“exit“)
end Obelix
...
```



Merging


Trace P0
Tr P n-1
Merge Process
P_0 - P_{n-1}
10.4 P0 Obelix enter
10.5 P1 Obelix enter
10.6 P0 MPI_Send enter
10.7 P1 MPI_Recv enter
10.8 P0 MPI_Send exit
11.0 P1 MPI_Recv exit

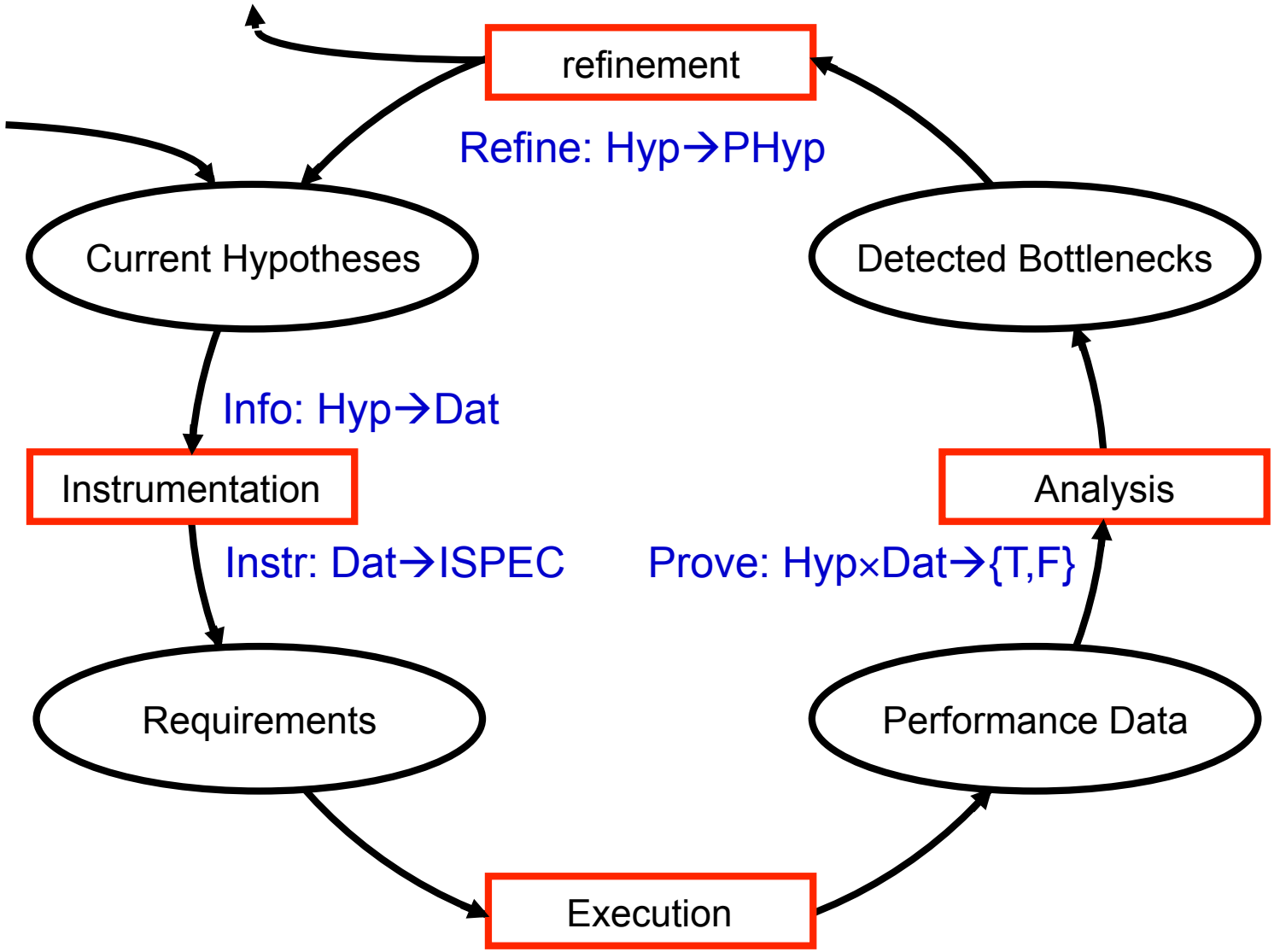
Visualization of Dynamic Behaviour



Profiling vs Tracing

- Profiling
  - recording summary information (time, #calls,#misses...)
  - about program entities (functions, objects, basic blocks)
  - very good for quick, low cost overview
  - points out potential bottlenecks
  - implemented through sampling or instrumentation
  - moderate amount of performance data
- Tracing
  - recording information about events
  - trace record typically consists of timestamp, processid, ...
  - output is a trace file with trace records sorted by time
  - can be used to reconstruct the dynamic behavior
  - creates huge amounts of data
  - needs selective instrumentation

1. Motivation
2. Performance analysis basics
3. Measurement techniques
4. Analysis techniques
5. The tools


refinement
Refine: Hyp→PHyp
Current Hypotheses
Detected Bottlenecks
Info: Hyp→Dat
Instrumentation
Analysis
Instr: Dat→ISPEC
Prove: Hyp×Dat→{T,F}
Requirements
Performance Data
Execution

Common Performance Problems with MPI Programs

- Single node performance
  - Excessive number of 2nd-level cache misses
  - Low number of issued instructions
- IO
  - High data volume
  - Sequential IO due to IO subsystem or sequentialization in the program
- Excessive communication
  - Frequent communication
  - High data volume

Common Performance Problems with MPI

- Frequent synchronization
  - All-to-all operations
  - Barrier operations
- Load balancing
  - Wrong data decomposition
  - Dynamically changing load

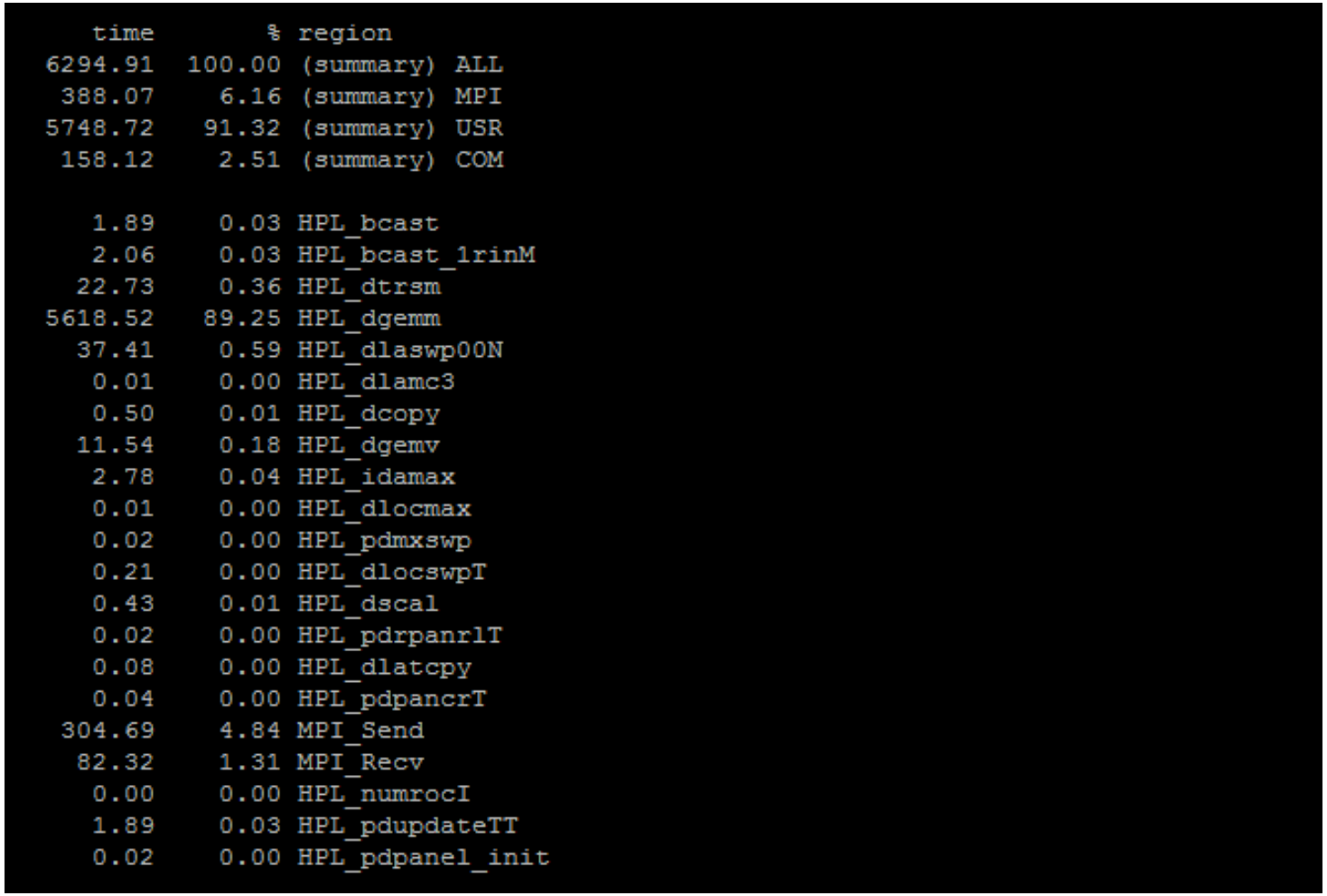
Common Performance Problems with SM

- Single node performance
  - ...
- IO
  - ...
- Excessive communication
  - Large number of remote memory accesses
  - False sharing
  - False data mapping
- Frequent synchronization
  - Implicit synchronization of parallel constructs
  - Barriers, locks, ...
- Load balancing
  - Uneven scheduling of parallel loops
  - Uneven work in parallel sections

Analysis Techniques

- Offline vs online analysis
  - Offline: first generate data then analyse
  - Online: generate and analyze data while application is running
  - Online requires automation➔limited to standard bottlenecks
  - Offline suffers more from size of measurement information
- Three techniques to support user in analysis
  - Source-level presentation of performance data
  - Graphical visualization
  - Ranking of high-level performance properties

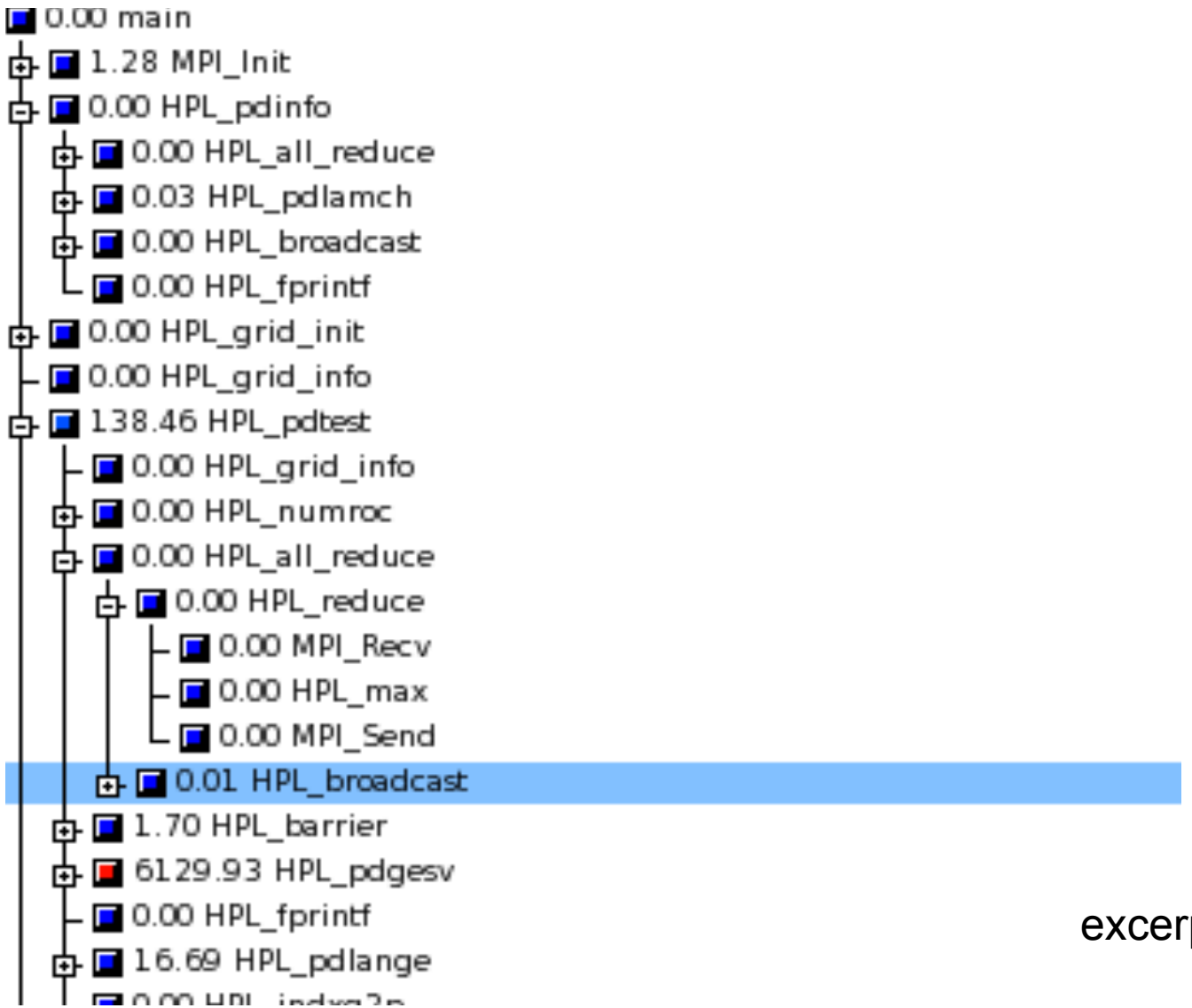
Analysis Techniques: Flat Profile



```
    time       % region
 6294.91  100.00 (summary) ALL
  388.07    6.16 (summary) MPI
 5748.72   91.32 (summary) USR
  158.12    2.51 (summary) COM

    1.89    0.03 HPL_bcast
    2.06    0.03 HPL_bcast_1rinM
   22.73    0.36 HPL_dtrsm
 5618.52   89.25 HPL_dgemm
   37.41    0.59 HPL_dlaswp00N
    0.01    0.00 HPL_dlamc3
    0.50    0.01 HPL_dcopy
   11.54    0.18 HPL_dgemv
    2.78    0.04 HPL_idamax
    0.01    0.00 HPL_dlocmax
    0.02    0.00 HPL_pdmxswp
    0.21    0.00 HPL_dlocswpT
    0.43    0.01 HPL_dscal
    0.02    0.00 HPL_pdrpanrlT
    0.08    0.00 HPL_dlatcpy
    0.04    0.00 HPL_pdpancrT
  304.69    4.84 MPI_Send
   82.32    1.31 MPI_Recv
    0.00    0.00 HPL_numrocI
    1.89    0.03 HPL_pdupdateTT
    0.02    0.00 HPL_pdpanel_init
```

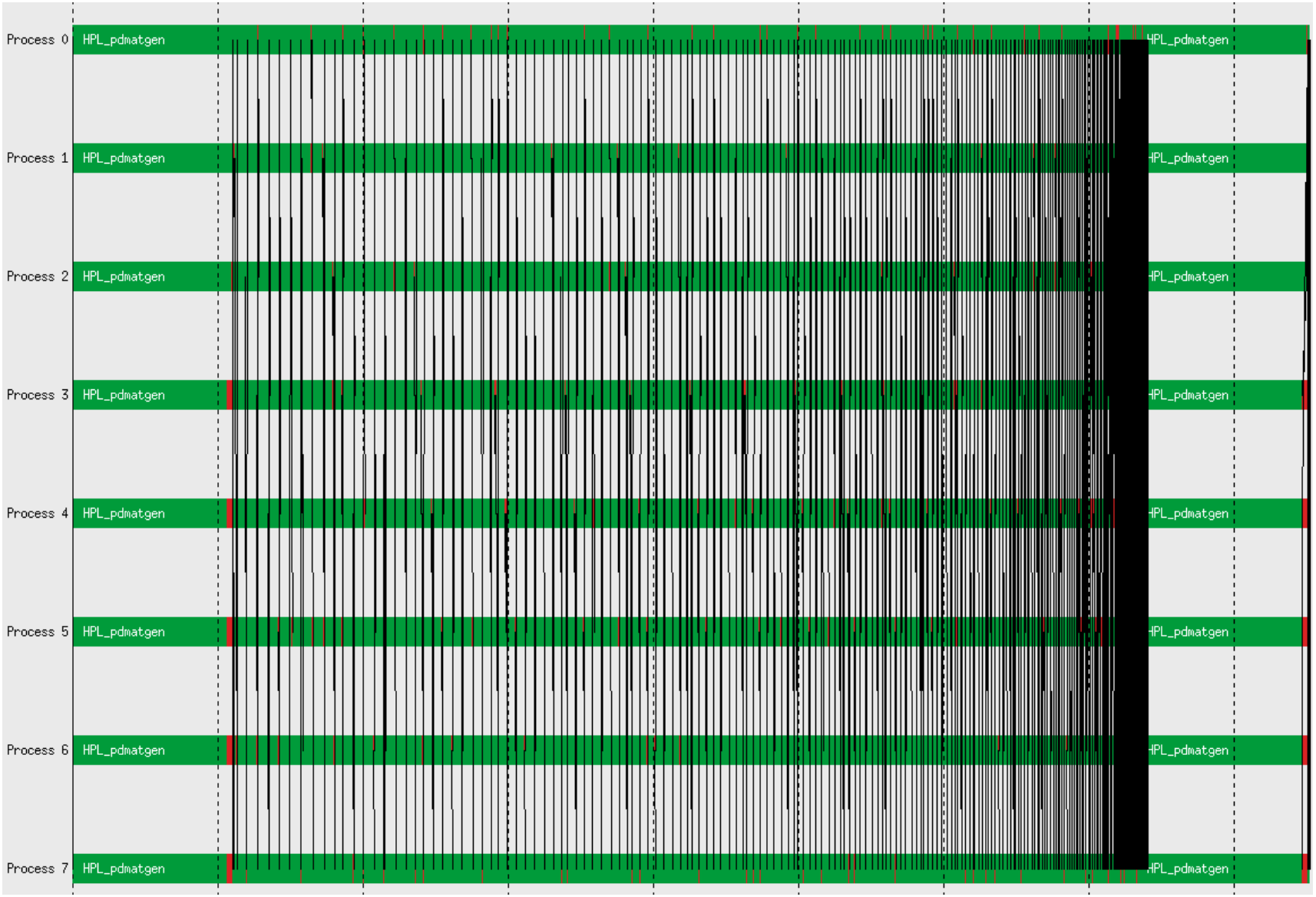
excerpt from scalasca

Analysis Techniques: Call-path profile



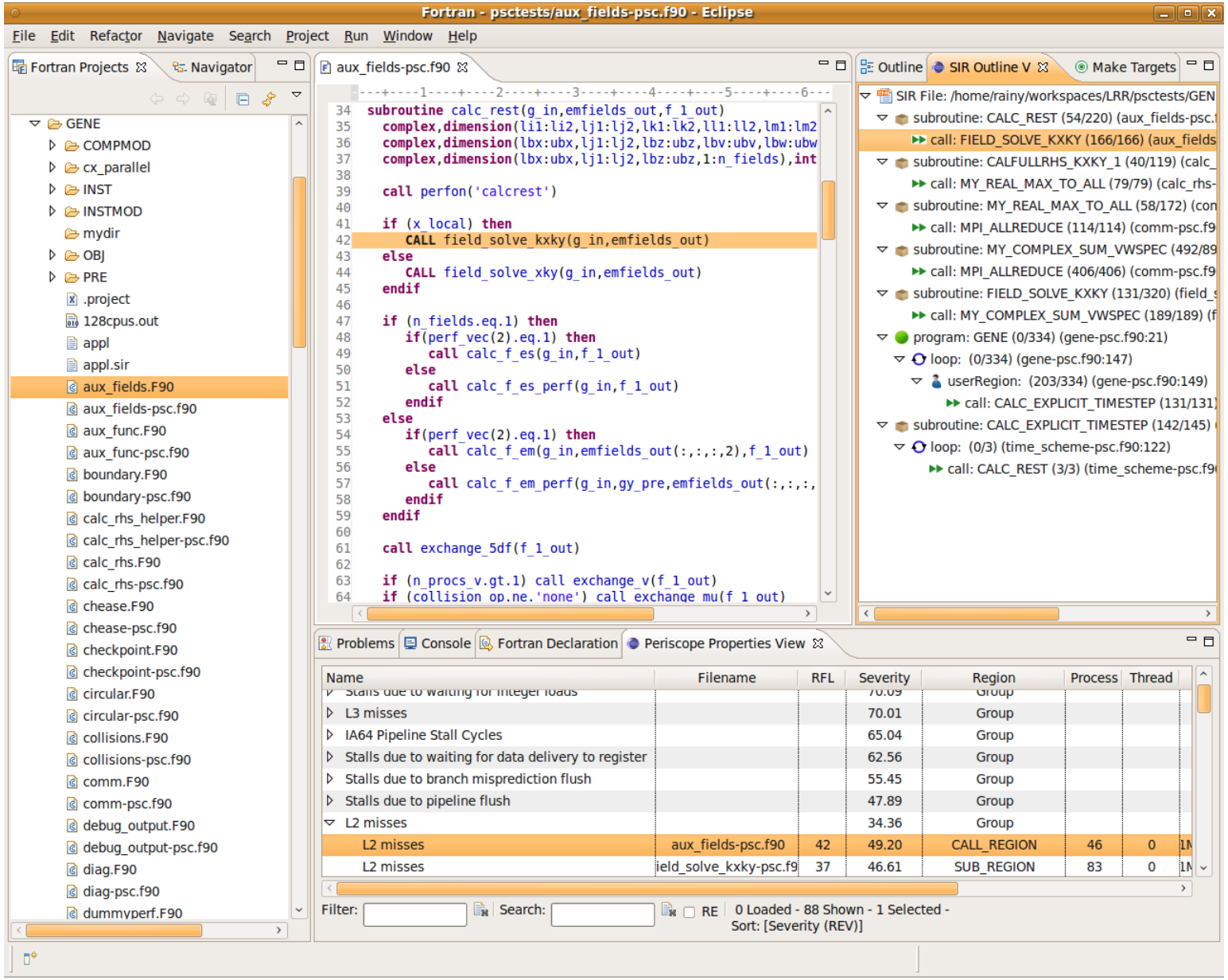
excerpt from scalasca

Analysis Techniques: Timeline



excerpt from vampir

Analysis Techniques: Properties on Source Level



More later at this workshop

1. Motivation
2. Performance analysis basics
3. Measurement techniques
4. Analysis techniques
5. The tools

The Tools

- Callgrind / Kcachegrind
  - Cache analysis via simple cache simulation
- Periscope
  - Automatic detection of performance properties
  - Profile data
  - Online
- Scalasca
  - Call-path profiling & tracing library
  - Automatic trace analysis
- VAMPIR
  - Visual trace analysis
- IBM HPC Toolkit
  - Profiling and tracing

Overview

1. Motivation
2. Performance analysis basics
3. Measurement techniques
4. Analysis techniques
5. The tools