Performance Analysis and Tuning of Parallel Programs: Resources and Tools

PART1

Introduction and Overview

Michael Gerndt (Technische Universität München)

PART2

Resources and Tools

Bernd Mohr (Forschungszentrum Jülich)

PART3

Automatic Performance Analysis with Paradyn

Barton Miller (University of Wisconsin-Madison)

Optimization Goals

- Execution Time
  - Get the answer as fast as possible.
- Throughput
  - Get as many happy users as possible.
- Utilization
  - Get the best exploitation of the invested money.

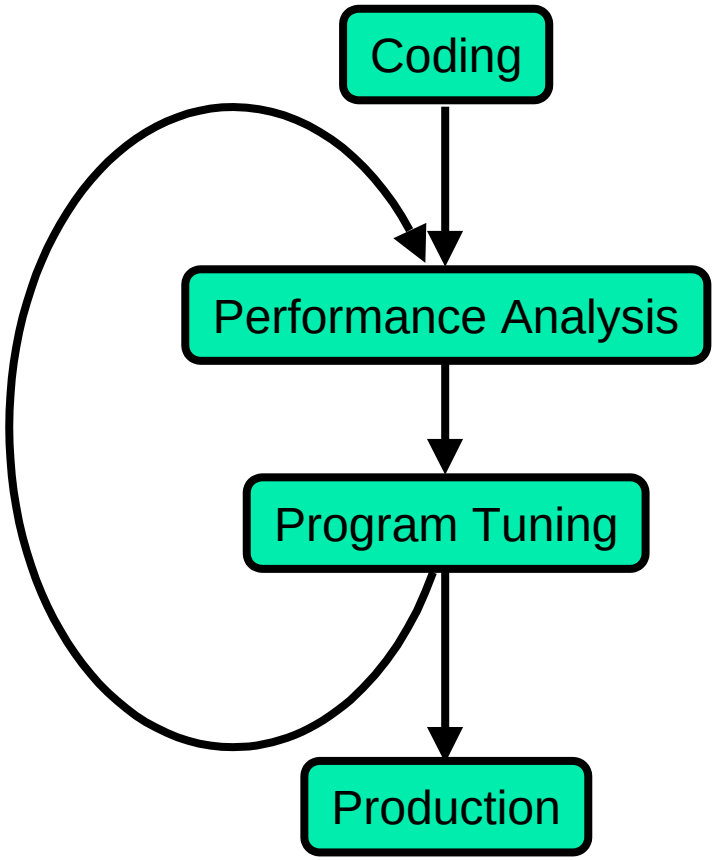
Performance of Application or Computer System

"How well is the goal achieved"

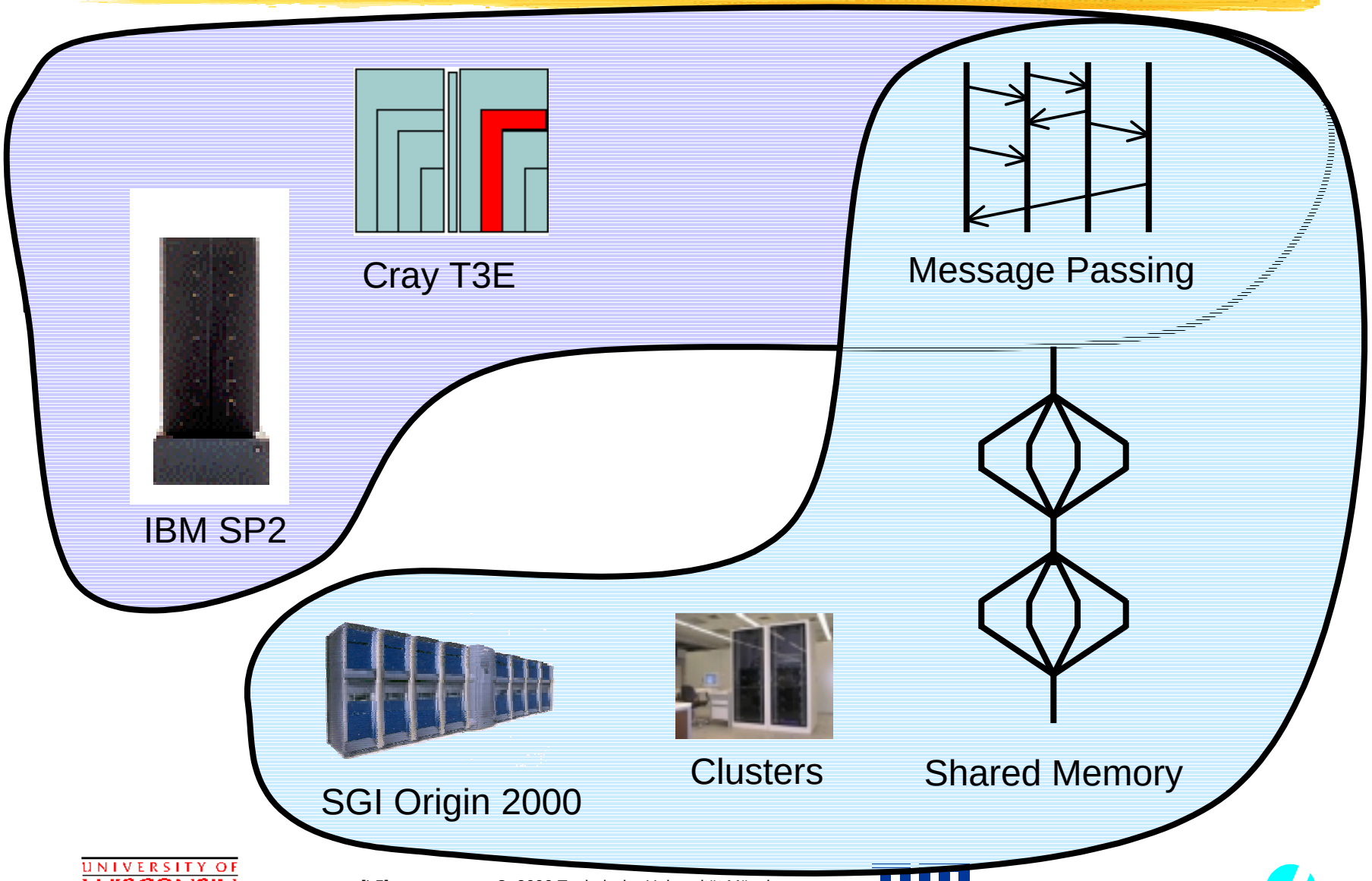
- Performance Analysis and Tuning is based on
  - measurements of performance data
  - evaluation of performance data
  - code or policy changes based on analysis results

Scenario: Parallel Computing

- Optimization goal: minimal execution time



Parallel Systems and Programming Models

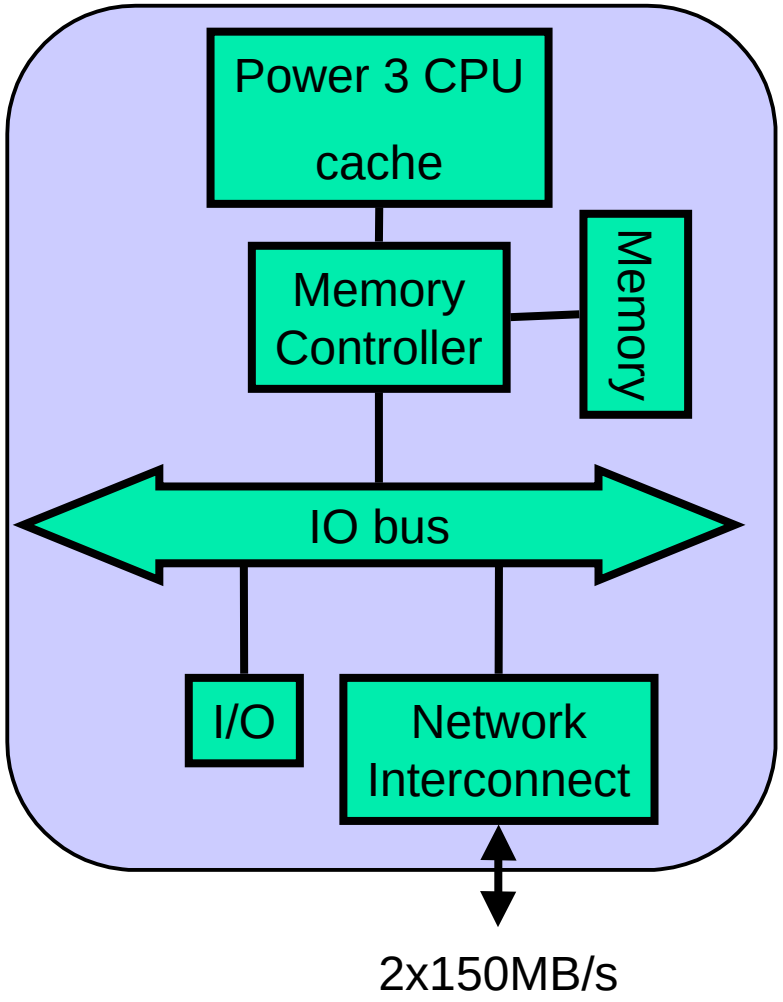


Contents

- Part 1 - Introduction and Overview
  - Parallel Systems
  - Programming Models
  - Common Performance Problems
  - Performance Measurement and Analysis Techniques
  - Performance Tuning
- Part 2 - Resources and Tools
  - Cray T3E, IBM SP2, SGI O2K Vendor Tools
  - 3rd Party Tools
  - Research Tools
  - Fall-back Tools
- Part 3 - Automatic Performance Analysis with Paradyn
  - Sample Analysis Session with Paradyn
  - Paradyn technologies and Architecture
  - On-going Research and Developments

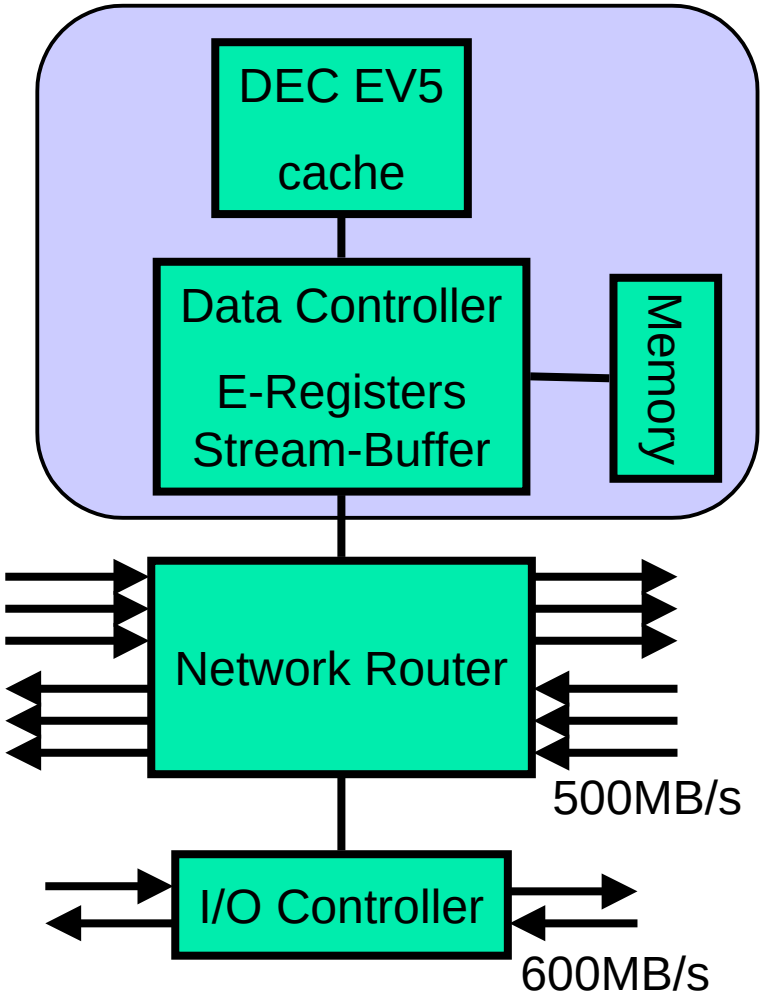
IBM SP2

- message passing system
- cluster of workstations
- 200 MHz Power 3 CPU
  - Peak 800 MFLOPS
  - 4-16 MB 2nd-level cache
  - sustained memory bandwidth 1.6 GB/s
- multistage crossbar switch
- MPI
  - latency 21.7 µsec
  - bandwidth 139 MB/s
- I/O hardware distributed among the nodes



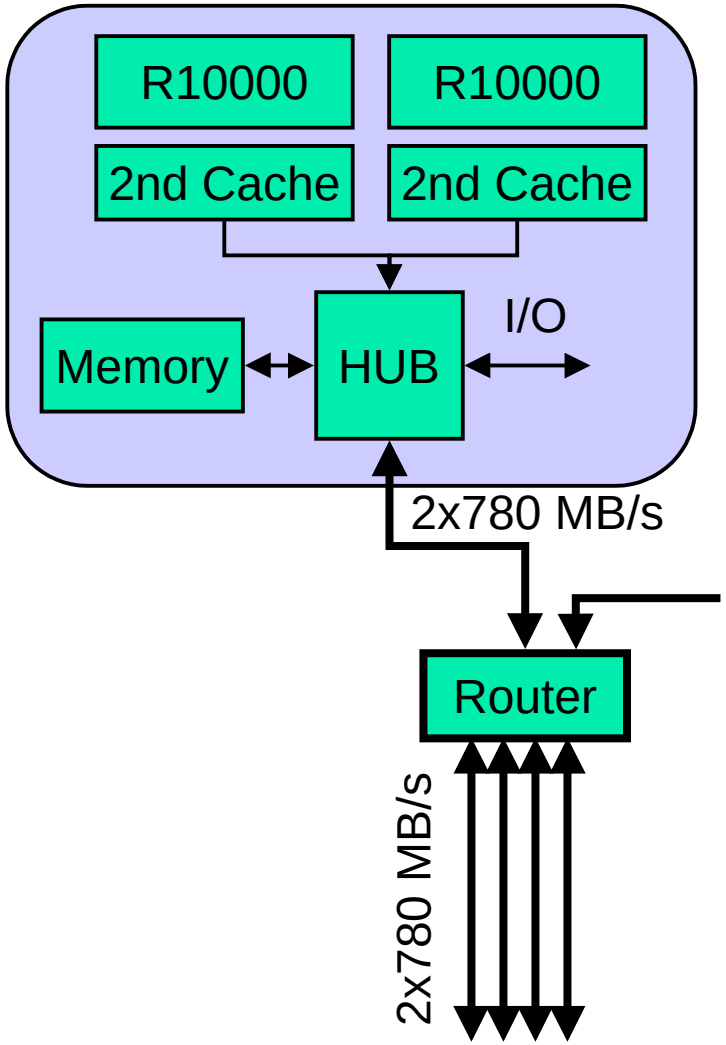
Cray T3E

- remote memory access system
- single system image
- 600 MHz DEC Alpha
  - peak performance 1200 MFLOPS
  - 2nd-level data cache 96 KB
  - memory bandwidth 600 MB/s
- 3D torus network
- MPI
  - latency 17 µsec
  - bandwidth 300 MB/s
- shmem
  - latency 4 µsec
  - bandwidth 400 MB/s
- SCI-based I/O network



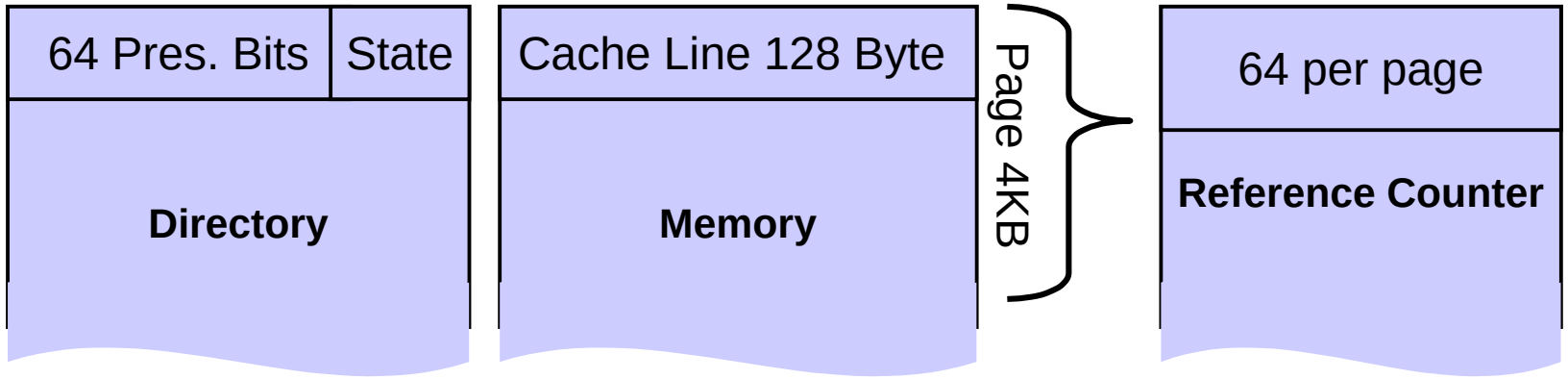
SGI Origin 2000

- cc-NUMA system
- single system image
- 250 MHz MIPS R10000
  - peak performance 500 MFLOPS
  - 2nd-level data cache 4-8 MB
  - memory bandwidth 670 MB/s
- 4D hypercube
- MPI
  - latency 16 µsec
  - bandwidth 100 MB/s
- remote memory access
  - latency 497 nsec
  - bandwidth 600 MB/s
- I/O hardware distributed among nodes but global resource

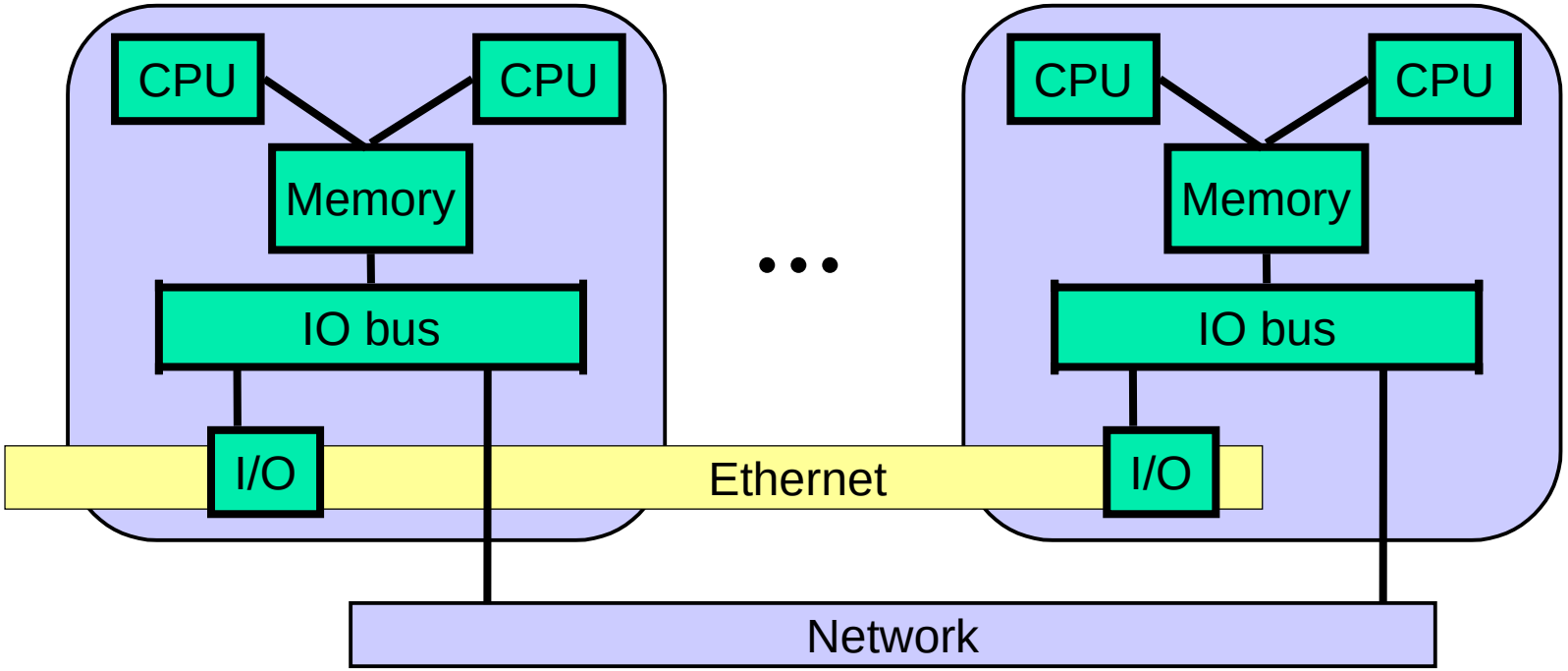


ccNUMA Implementation on SGI Origin 2000

- Access to remote data
  - read or write copy is transferred into the cache
  - coherence of copies and home location is maintained through directory
  - allocation of pages to node memories is by first touch
  - migration is possible based on reference counters

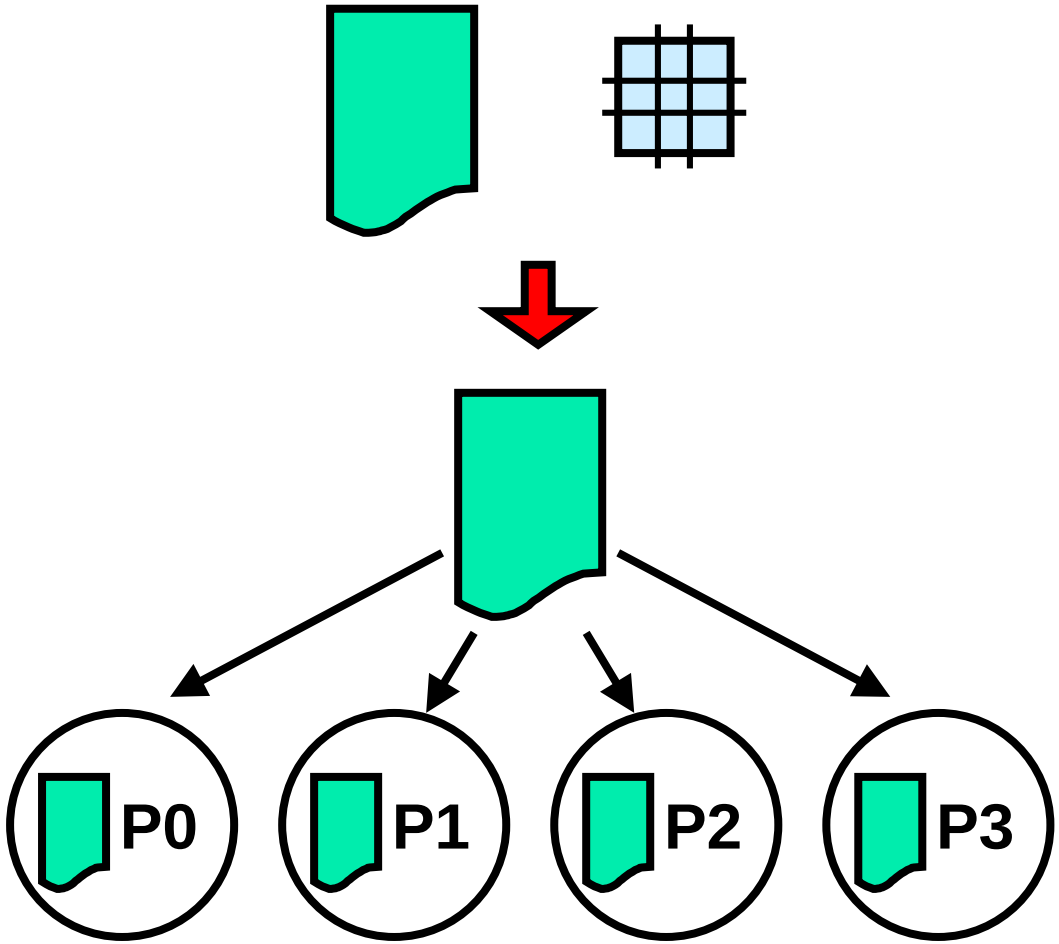


Clusters



- Hierarchical architecture: shared memory in a node, message passing across nodes
- PC-based nodes or workstation-based nodes
- Networks: Myrinet, Scalable Coherent Interface, Gigabit-Ethernet

Single Program Multiple Data (SPMD)



sequential program
+ domain decomposition

node program
+ message passing

program replicated

MPI (Message Passing Interface)

- MPI is the standard programming interface
  - MPI 1.0 in 1994
  - MPI 2.0 in 1997
- Library interface (Fortran, C, C++)
- It includes
  - point-to-point communication
  - collective communication
  - barrier synchronization
  - one-sided communication (MPI 2.0)
  - parallel I/O (MPI 2.0)
  - process creation (MPI 2.0)

MPI Example

Node 0

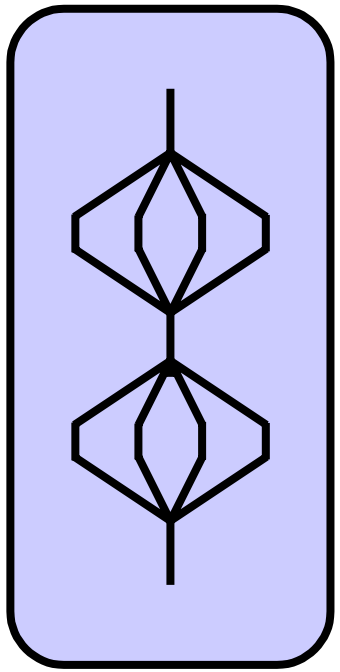
```
//local computation
for(i=0;i<10;i++) a(i)= f(i);

MPI_send(a(1), 10, MPI_INT, Node1, 20, comm)

//reduction
for (i=0;i<10;i++) s=s+a(i);
MPI_reduce(s, s1, 1, MPI_INT, SUM, Node1, comm)
```

Node 1

```
...

MPI_recv(b(1), 10, MPI_INT, Node0, 20, comm, status)

...
MPI_reduce(s, s1, 1, MPI_INT, SUM, Node1,comm)

printf("Global sum: %d",s1)
```

OpenMP: Directive-based SM Parallelization

- OpenMP is a standard shared memory programming interface (1997)
- Directives for Fortran 77 and C/C++
- Fork-join model resulting in a global program
- It includes:
  - Parallel loops
  - Parallel sections
  - Parallel regions
  - Shared and private data
  - Synchronization primitives
    - barrier
    - critical region



OpenMP: Example

```
!$OMP parallel do schedule(static, 10)
      do i=1,100
        a(i)=f(i)
      enddo
!$OMP end parallel do

!$OMP parallel do reduction(+:a),
!$OMP*             schedule(dynamic,10)
      do i=1,100
        s=s+a(i)
      enddo
!$OMP end parallel do

      write(*,*) 'Global sum:', s
```

Hybrid Programming

- Hierarchical parallel architectures are gaining more importance.
- Uniform programming interface is possible but might not be most efficient.
  - MPI requires optimized library for intranode communication.
  - OpenMP requires global address space implemented in software.
- Hybrid programming allows to optimize both levels of the hierarchy.
  - MPI across nodes
  - OpenMP within a node
- In addition to increased programming complexity it requires additional performance-relevant design decisions prior to coding.

Common Performance Problems with MPI

- 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

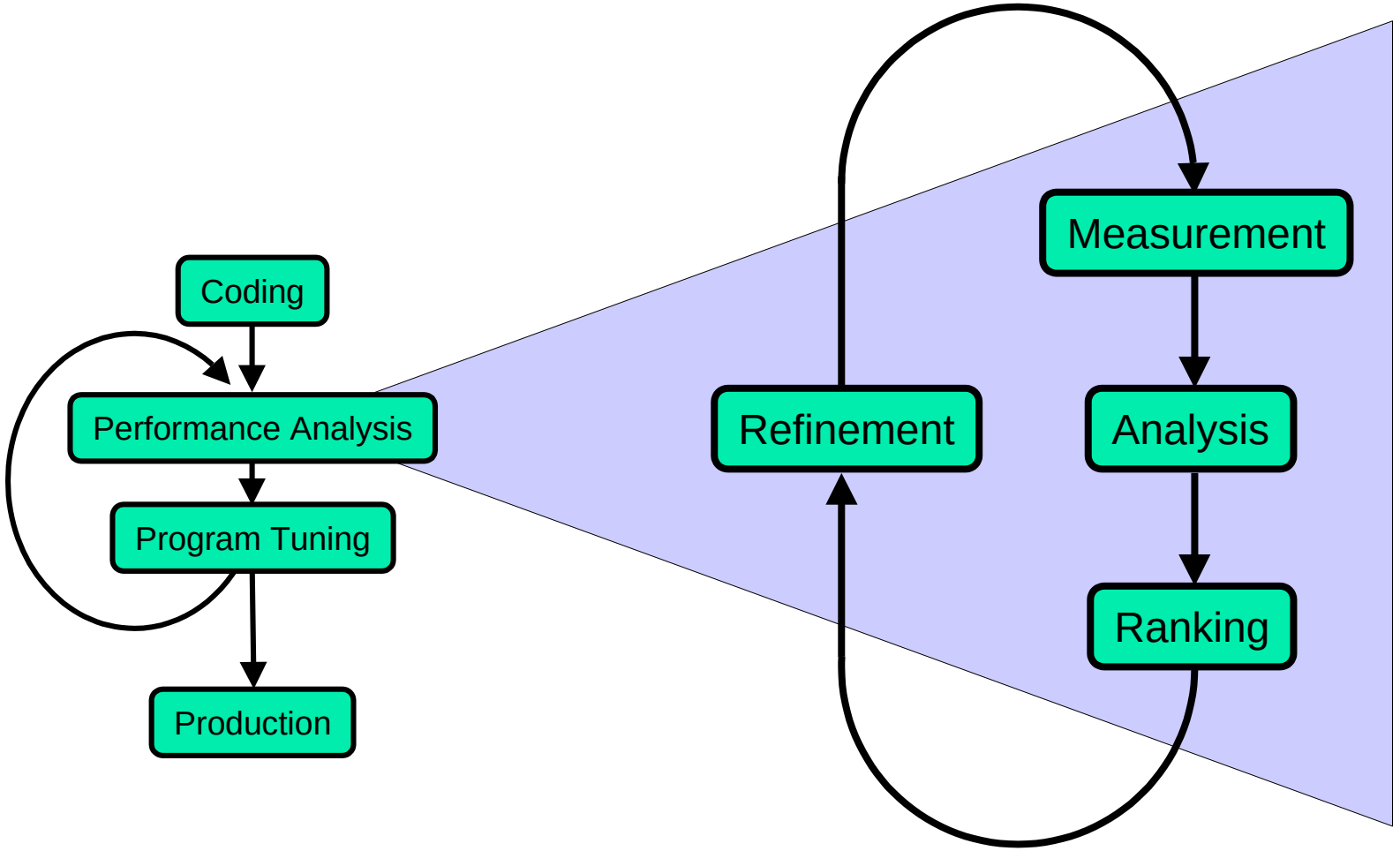
Common Performance Problems with MPI

- Frequent synchronization
  - Reduction 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

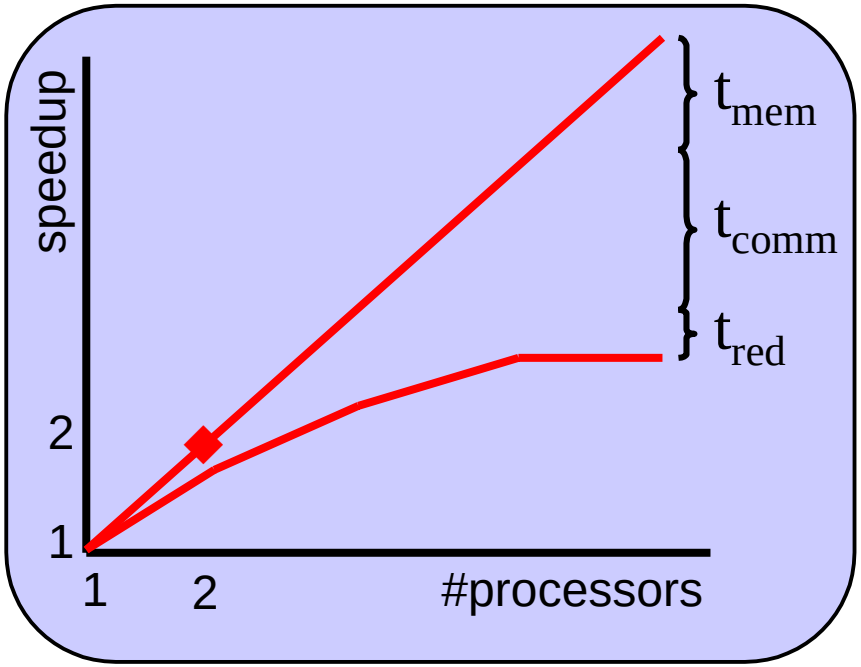
Performance Analysis Process



Overhead Analysis

- How to decide whether a code performs well:
  - Comparison of measured MFLOPS 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}
    - ...

Other Performance Metrics

Scaled speedup: Problem size grows with machine size

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

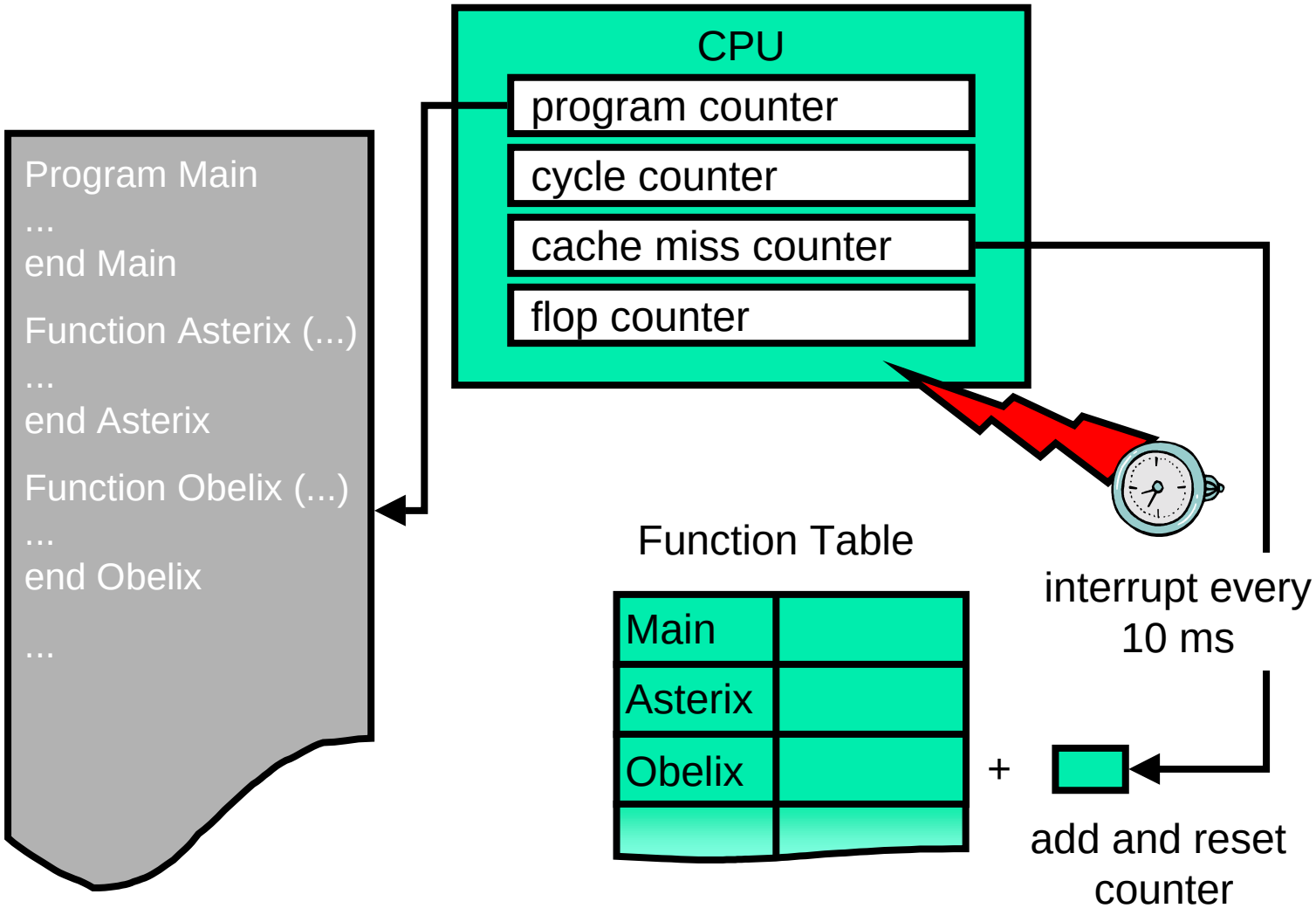
Parallel efficiency: Percentage of ideal speedup

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

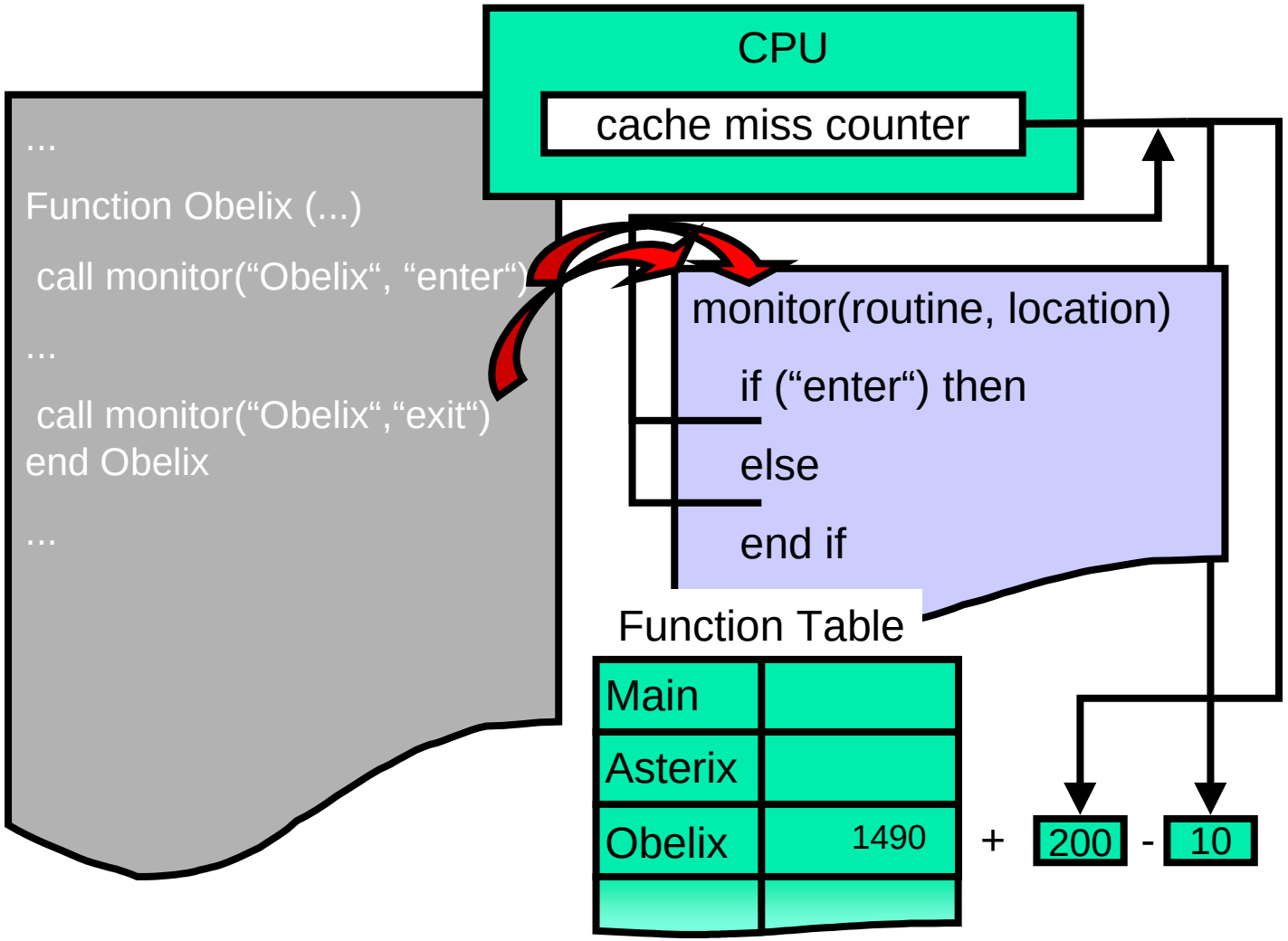
Performance Measurement Techniques

- 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: Precise measurement
- Tracing: Recording performance data of individual events

Sampling



Instrumentation and Monitoring



Instrumentation Techniques

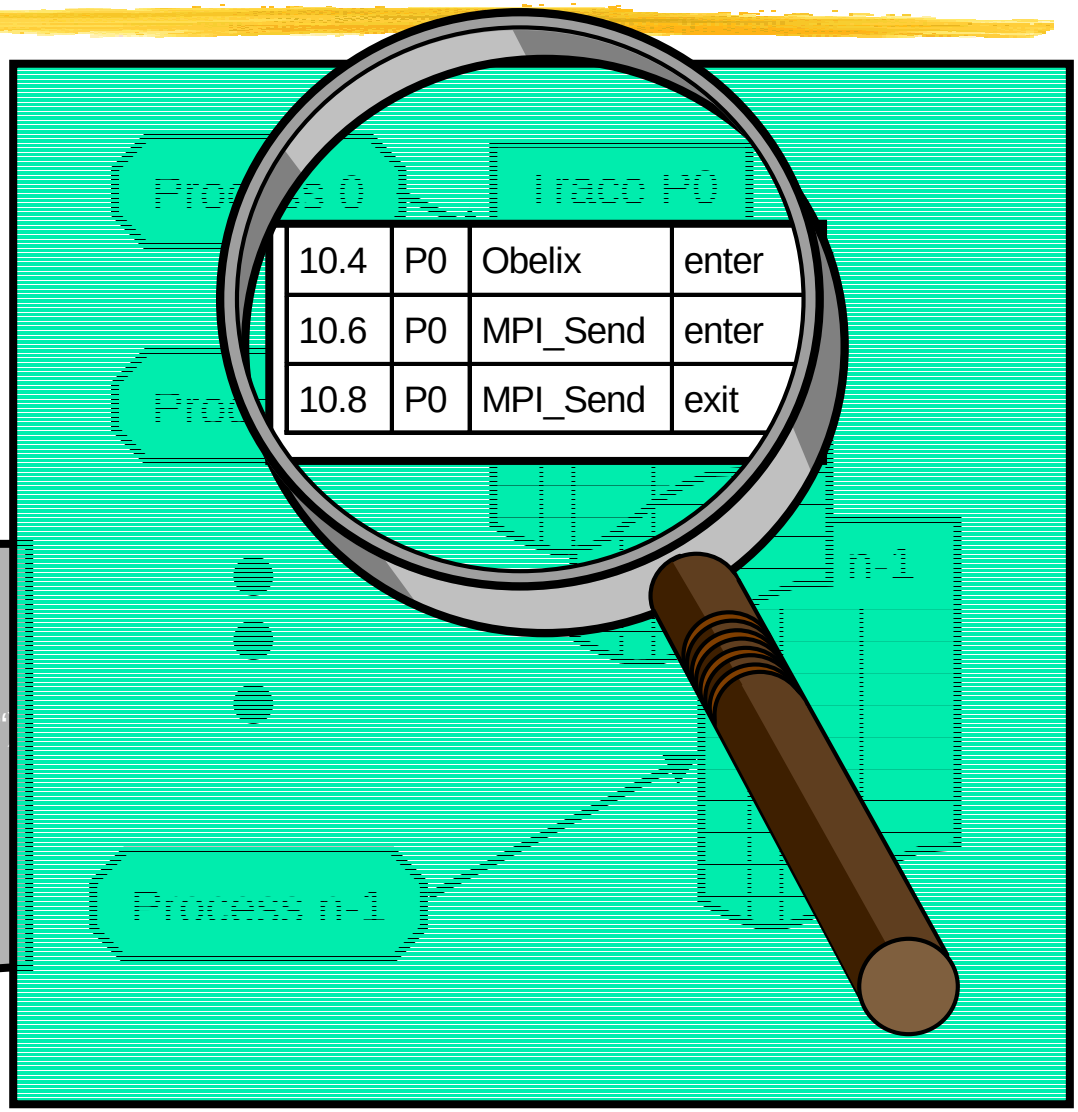
- Source code instrumentation
  - done by the compiler, source-to-source tool, or manually
    - + portability
    - + link back to source code easy
    - – re-compile necessary when instrumentation is changed
    - – difficult to instrument mixed-code applications
    - – cannot instrument system or 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
...
```

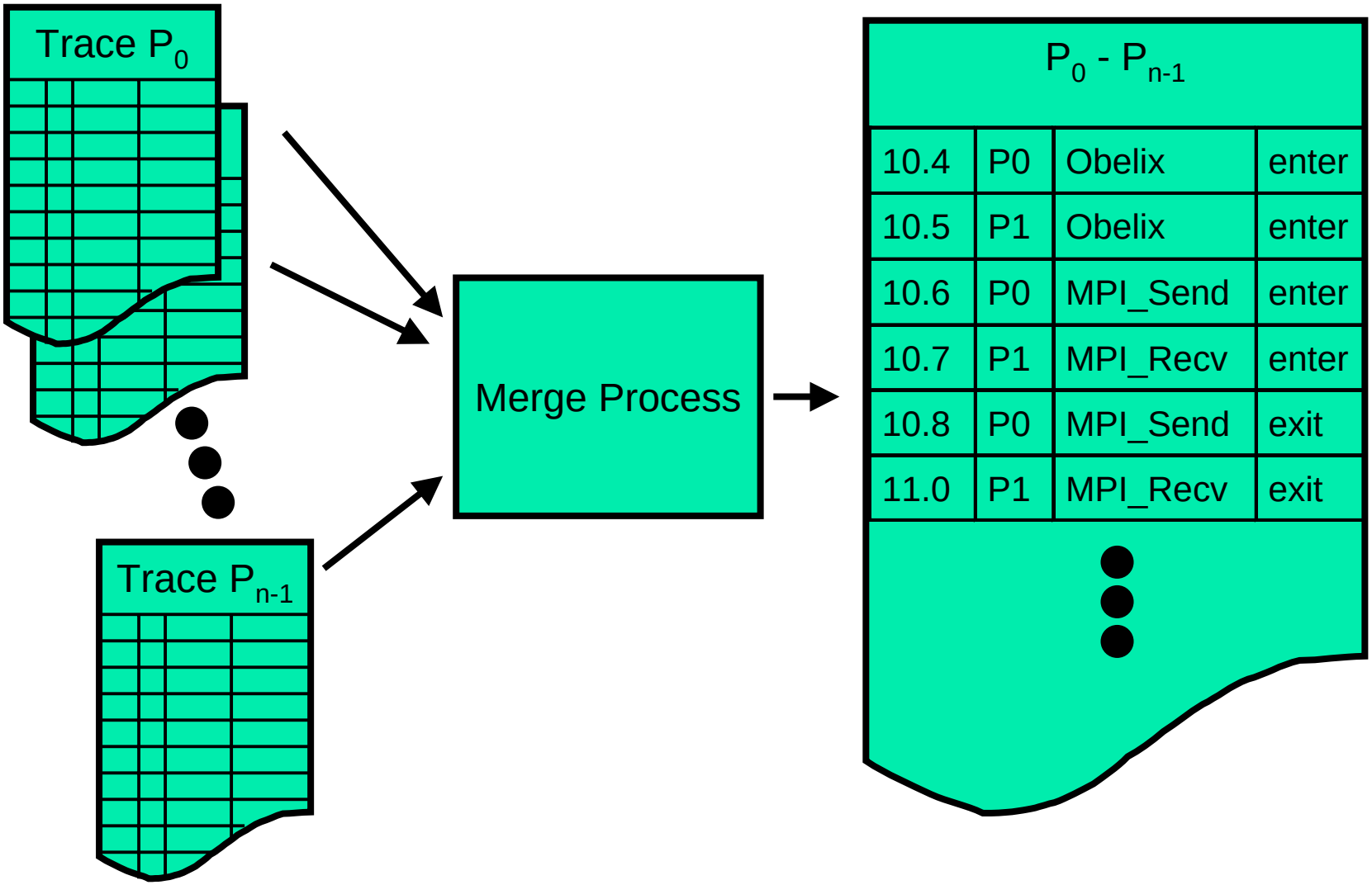
MPI Library

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

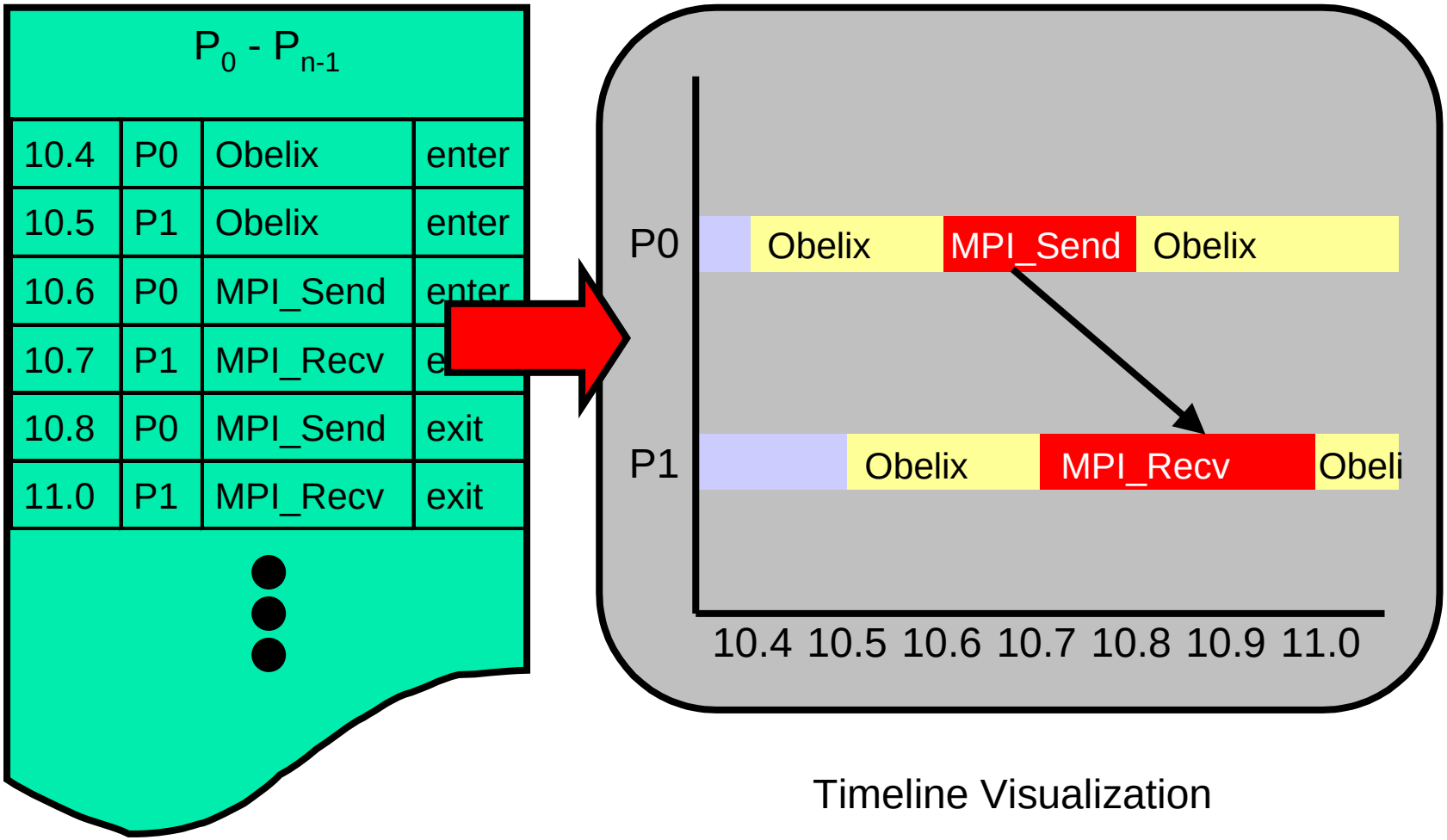


10.4	P0	Obelix	enter
10.6	P0	MPI_Send	enter
10.8	P0	MPI_Send	exit

Merging



Visualization of Dynamic Behavior

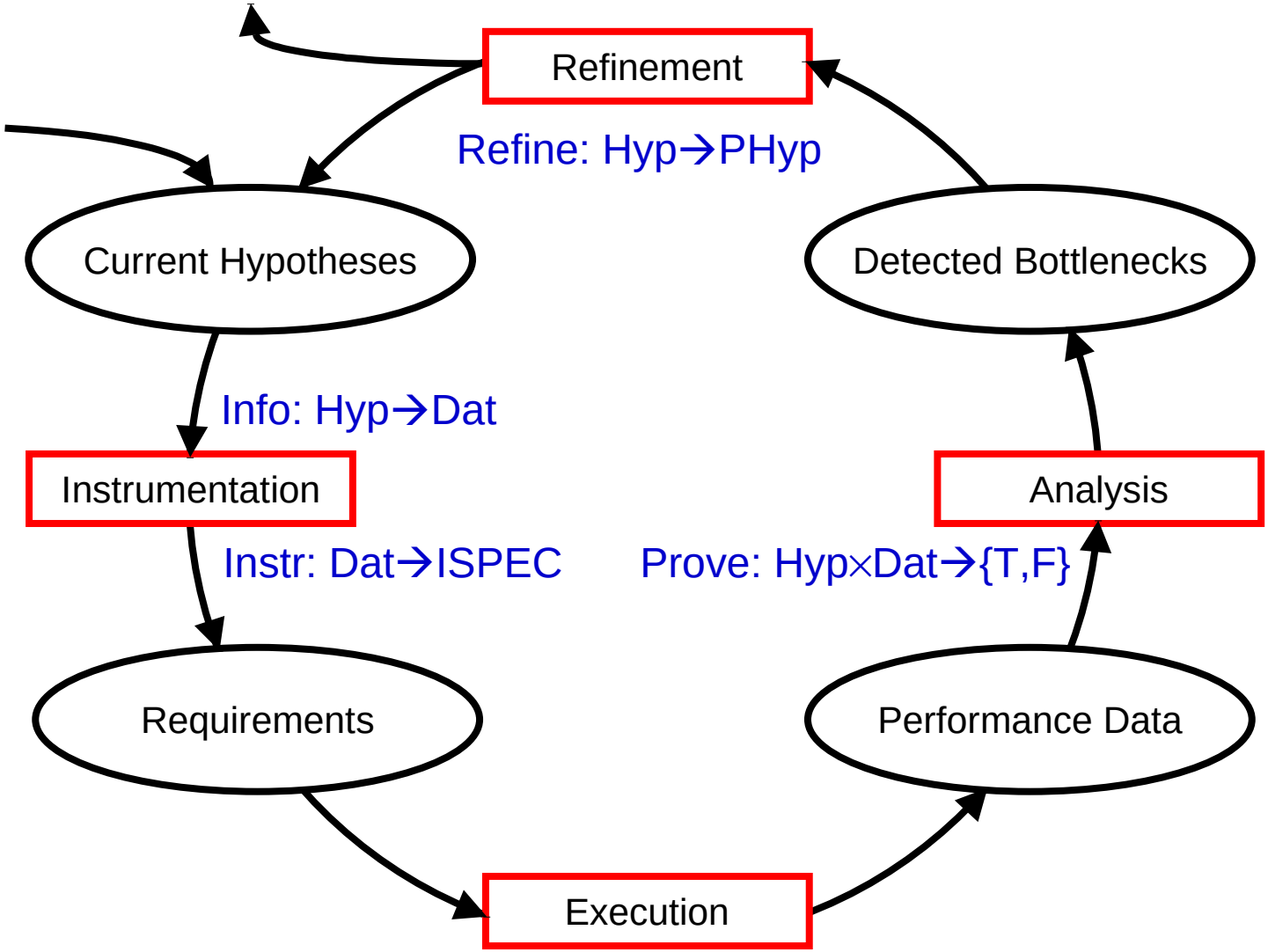


Timeline Visualization

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

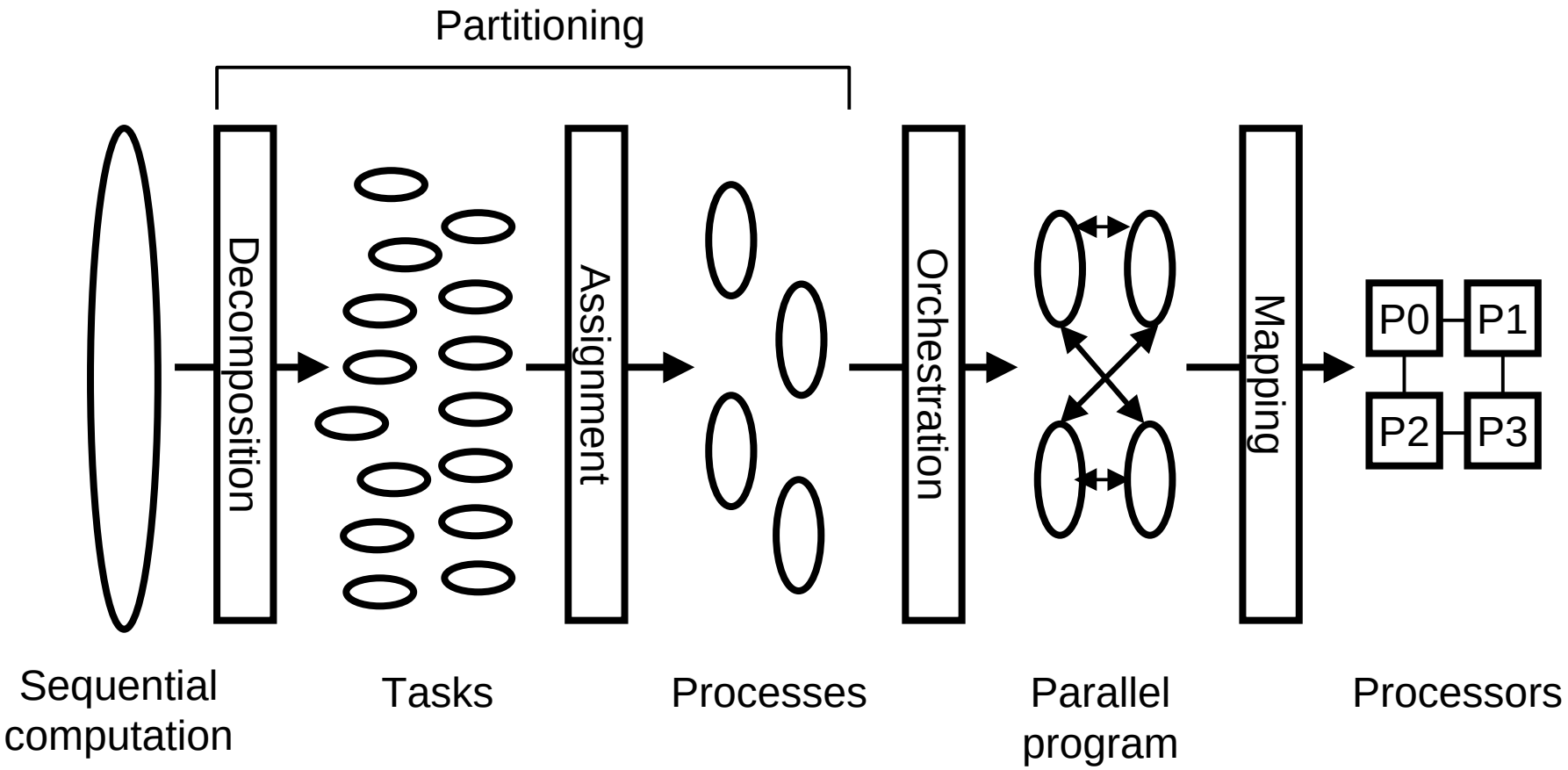
Performance Analysis



Performance Prediction and Benchmarking

- Performance analysis determines the performance on a given machine.
- Performance prediction allows to evaluate programs for a hypothetical 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.

Steps in the Parallelization Process



Parallel Computer Architecture, D. Culler & J.P. Singh

Performance Tuning

- Identification of performance bottlenecks
- Level of program tuning
  - Partitioning
    - Load balancing
    - Inherent communication
    - Extra work
    - Synchronization
  - Orchestration
    - Extra work
    - Artifactual communication
    - Synchronization
  - Mapping
    - Communication delay and contention

Load balance

- Load balance and reduction of synchronization wait time
  - Equalizing work assigned to processes
  - Reducing serialization due to mutual exclusion and dependences

- Tuning has four aspects:
  - Amount parallelism
  - Distribution strategy (static or dynamic)
  - Granularity
  - Synchronization costs

Load Balance: Amount of Parallelism

- More precise dependence analysis might unveil more parallelism.
- Algorithmic changes
- Data parallelism + functional parallelism
  - Different calculations
  - Pipelining

Load Balance: Distribution Strategy

- Static or predetermined
  - Small management overhead
  - Requires good prediction of runtime behavior
- Dynamic
  - Semistatic or repartitioning: requiring good prediction for phase
  - Dynamic tasking adapts to unpredictable work or system environment properties, e.g. self scheduling for parallel loops.
  - Higher management overhead
  - Potentially increasing communication and compromising data locality.

Load Balance: Granularity

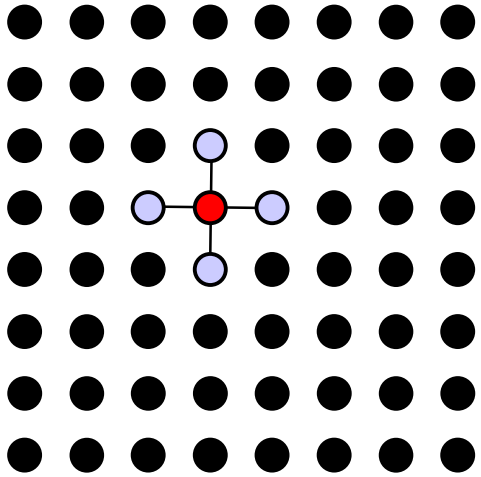
- Finer task granularity
  - Offers more potential to balance load.
- Coarser task granularity
  - Reduces contention for task queue
  - Reduces management overhead
  - Might increase data locality

Inherent Communication

- Reduce the communication-to-computation ratio
  - Assign communicating tasks to same process
  - Partitioning based on domain decomposition
    - Choose a partition that increases surface-to-volume ratio
    - Strategies:
      - Static by inspection of code
      - Static by inspection of input data set
      - Semistatically with periodic repartitioning
      - Statically or semistatically with dynamic task stealing

Example: Equation Solver Kernel

- Solver kernel for a simple partial differential equation.
- Finite difference method
- Grid (n+2) × (n+2)
- Fixed boundaries
- Interior point are recomputed
  - Mean value of five points in stencil
  - In place computation
  - Gauss-Seidel method
    - New values of upper and left point
    - Old values of lower and right point
  - Termination if difference between old and new value is below threshold for all points



```
A[i,j]=0.2*(A[i,j]+
       A[i-1,j]+A[i,j-1]+
       A[i,j+1]+A[i+1,j]
```

Assignment (1/2)

- Tasks are assigned to processes
- Row-based decomposition
  - Block assignment

$$iteration\ i \rightarrow \left\lfloor \frac{i}{p} \right\rfloor$$

  - Cyclic assignment

$$iteration\ i \rightarrow i \bmod p$$

  - Communication-to-computation ratio
    - Surface-to-volume ratio
    - Block $\dfrac{2*n}{n*\frac{n}{p}} = 2*\dfrac{p}{n}$ Cyclic $\dfrac{2*n*\frac{n}{p}}{n*\frac{n}{p}} = 2$

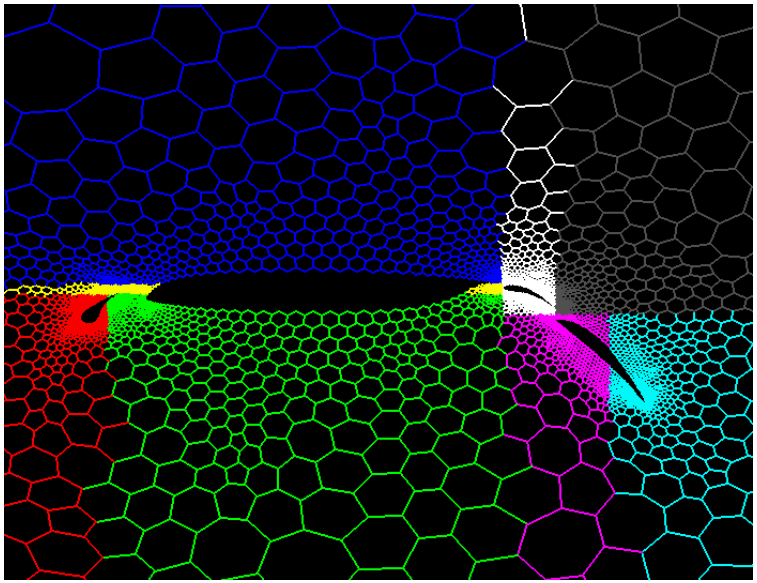
Assignment (2/2)

- Point-based decomposition
  - Block-Block assignment

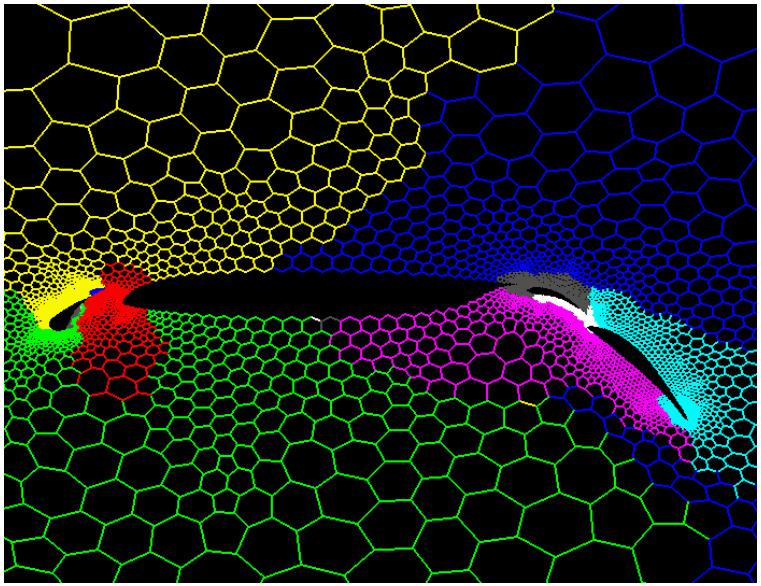
$$iteration\ (i,j) \rightarrow \left(\left\lfloor \frac{i}{\sqrt{p}} \right\rfloor, \left\lfloor \frac{j}{\sqrt{p}} \right\rfloor \right)$$

  - Communication-to-computation ratio

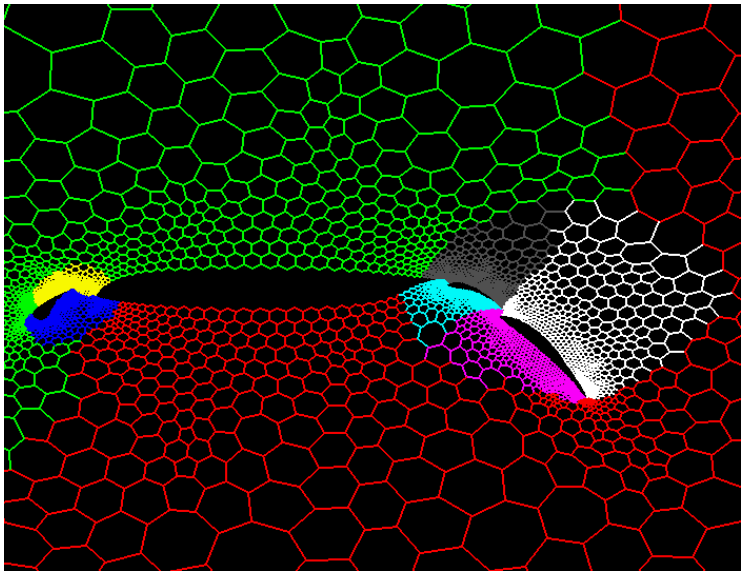
$$\frac{4\frac{n}{\sqrt{p}}}{\frac{n^2}{p}} = 4\frac{\sqrt{p}}{n}$$



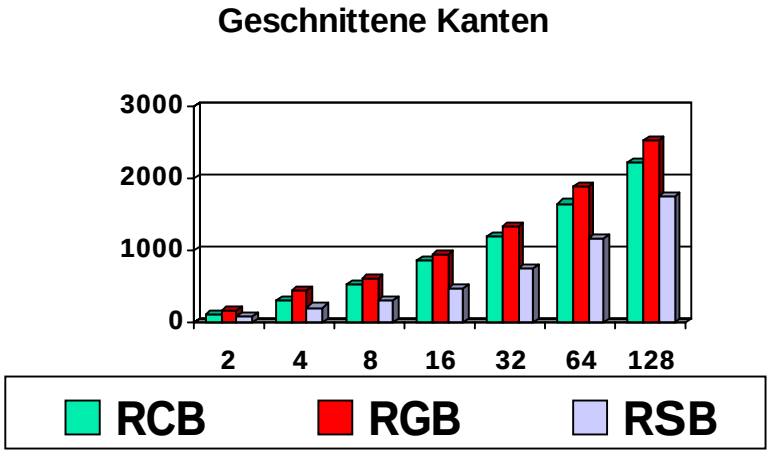
Recursive Coordinate Bisection



Recursive Graph Bisection



Recursive Spectral Bisection



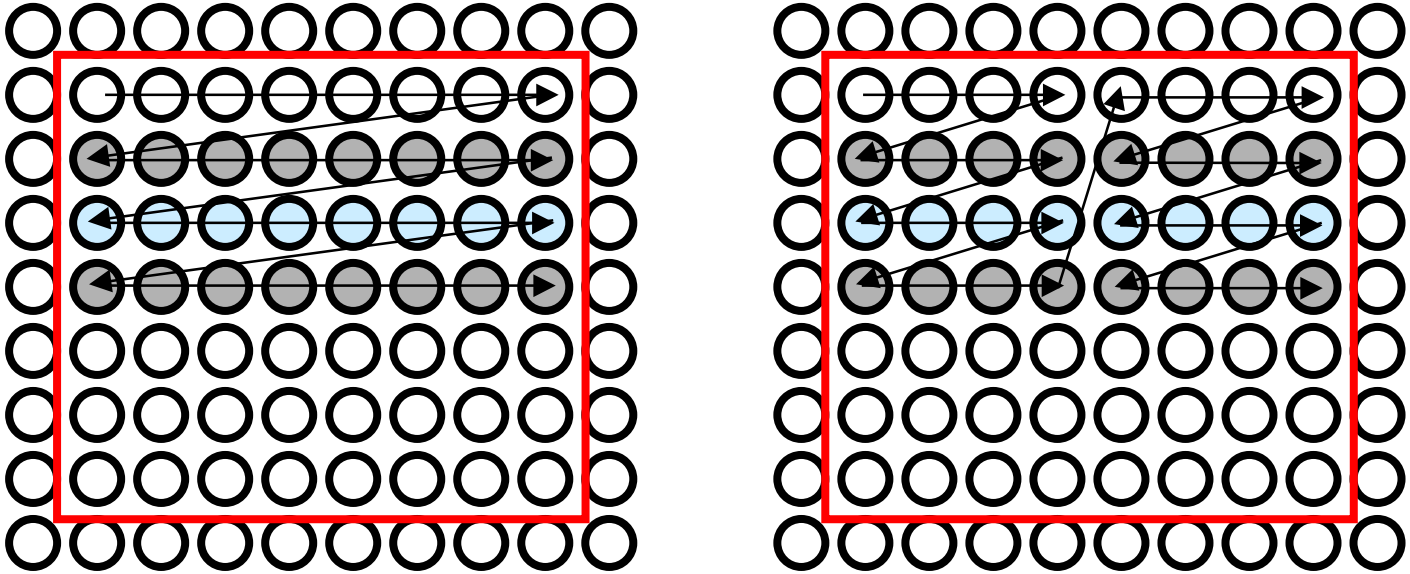
[Simon 91]

Artifactual Communication

- Poor allocation of data
  - Improve distribution or increase replication
- Unnecessary data in a transfer
  - Sending more information than required
    - Trade-off overhead for computation of precise data set vs communication overhead
  - Implicit transfer of unnecessary data, e.g. cache blocks
    - Restructure program to increase spatial locality
- Redundant communication of data
  - Transfer of changed data although not required, e.g. update protocols
  - Transfer same data multiple times
- Finite replication capacity
  - In cache and in main memory
- False sharing

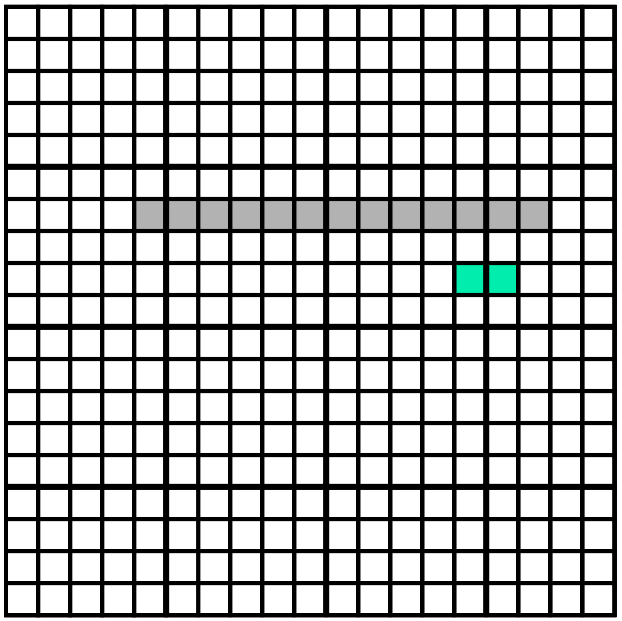
Improving Temporal Locality

- Artifactual communication is explicit in message passing
- Artifactual communication is implicit in shared memory
  - Improving temporal and spatial locality
  - Loop transformations, such as blocking

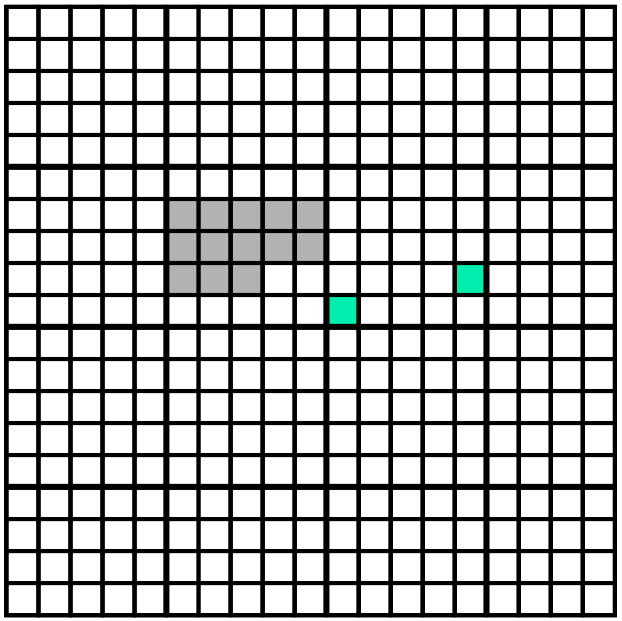


Improving Spatial Locality

- Data structure transformations can improve spatial locality
  - Memory allocation in node memories on page basis
    - Block partitioning does not take it into account
  - Cache lines might cross segment boundaries



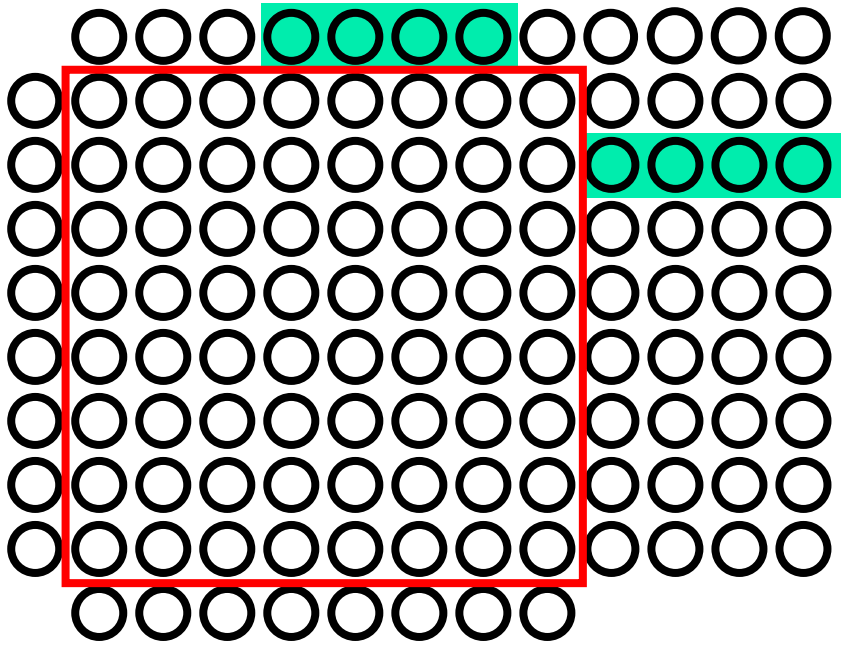
two-dimensional allocation



four-dimensional allocation

Trade-off in Communication Tuning

- Reducing inherent communication ⇒ square segments
- Reducing artifactual communication ⇒ row decomposition



Extra Work

- Management overhead for partitioning
  - Trade-off less efficient but less costly techniques
- Redundant computation of data values
  - Trade-off communication vs. computation
- Orchestration overhead
  - Fork-join overhead
    - Increase granularity of parallelism
  - Packing and unpacking data
    - Avoid packing by sending additional data

Reducing Synchronization

- Elimination of synchronization
  - Assign dependent tasks to same process
- Relaxing synchronization
  - Replacing barrier synchronization by point-to-point synchronization
- Separate locks for separate data items
- Reducing size of critical sections
- Reducing frequency of critical sections

Structuring Communication to Reduce Cost

- Overhead for initiating or processing a message
  - Fusion of small messages
- Network delay in switches
  - Select a mapping that minimizes the number of hops
- Contention
  - Network resources
    - Optimize mapping
  - Endpoint contention
    - Apply different orchestration, e.g. tree-based algorithm
    - Stagger communication in time
- Overall transmission time
  - Overlapping communication with computation
    - Start communication as early as possible
    - Multithreading
    - Prefetching

Performance Analysis and Program Tuning

- Identification of reasons of performance bottlenecks is limited by the information available.
- User knowledge about the program algorithm can improve performance diagnosis considerably.
- Performance tuning of parallel programs is extremely difficult due to
  - Dependence on machine characteristics
  - Global effects of local transformations
  - Trade-off between conflicting tuning goals

Performance Analysis and Tuning of Parallel Programs: Resources and Tools

PART1

Introduction and Overview

Michael Gerndt (Technische Universität München)

PART2

Resources and Tools

Bernd Mohr (Forschungszentrum Jülich)

PART3

Automatic Performance Analysis with Paradyn

Barton Miller (University of Wisconsin-Madison)

Performance Tuning: an Old Problem!

[Intentionally left blank]

Performance Tuning: an Even Older Problem

"The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible."

Charles Babbage
1791 - 1871

Contents

- Practical Performance Analysis and Tuning
- Debugging
  - Totalview
- CRAY T3E Tools
  - Apprentice
  - PAT
- IBM SP Tools
  - Xprofiler
  - VT
  - DPCL
  - Future Tools
- SGI Origin Tools
  - Perfex
  - SpeedShop
  - prof / cvperf
- 3rd Party Tools
  - DEEP
  - Vampir
  - Vampirtrace
  - Dimemas
  - GuideView
- Research Tools
  - Upshot / Nupshot / Jumpshot
  - MPICL / Paragraph
  - AIMS / NTV
  - Dynaprof
  - TAU
  - Other
- Fall-back Tools
  - Timer
  - HW Counter (PCL, PAPI)

Practical Performance Analysis and Tuning

- Peak and benchmark (Linpack, NAS) performance has nothing to do with 'real' performance
  - ⇒ The performance of your program is important
- 50% of 1 Gflop is better than 20% of 2GFlop
  - ⇒ the sustained fraction of peak performance is crucial
- Successful tuning is combination of
  - right algorithm and libraries
  - compiler flags and pragmas / directives (Learn and use them)
  - THINKING
- Measurement is better than reasoning / intuition (= guessing)
  - to determine performance problems
  - to validate tuning decisions / optimizations (after each step!)

Practical Performance Analysis and Tuning

- It is easier to optimize a slow correct program

  than to debug a fast incorrect one

  ⇒ Debugging before Tuning

  ⇒ Nobody really cares how fast you can compute the wrong answer

- The 80/20 rule
  - Program spents 80% time in 20% of code
  - Programmer spends 20% effort to get 80% of the total speedup possible in the code

  ⇒ Know when to stop!

- Don't optimize what doesn't matter

  ⇒ Make the common case fast

Practical Performance Analysis and Tuning

- Strategies for Speed
  - Use a better algorithm or data structure
    - watch out for algorithms with quadratic behavior $O(n^2)$!
  - Use optimized libraries (BLAS, Cray's libsci, IBM ESSL, ...)
  - Enable compiler optimizations (through options and pragmas)
  - Tune the code
    - Collect common subexpressions
    - Replace expensive operations by cheap ones
    - Write a special-purpose allocator
    - Minimize I/O and system calls
    - Unroll or eliminate loops
    - Cache frequently used values
    - Buffer input and output
    - Use approximate values
    - inline often used functions
- Space Efficiency
  - Save space by using the smallest possible data type
  - Don't store what you can easily re-compute

Practical Performance Analysis and Tuning

- Typical Procedure
  - Do I have a performance problem at all?
    - time / HW counter measurements
    - speedup and scalability measurements
  - What is the main bottleneck (calculation / communication / synchronization) ?
    - flat profiling (sampling / prof)
  - Where is the main bottleneck?
    - call graph profiling (gprof)
    - detailed (basic block) profiling
  - Where is the bottleneck exactly and why is it there?
    - tracing of selected parts to keep trace files manageable
  - Does my code has scalability problems?
    - profile code for typical small and large processor count
    - compare profiles function-by-function

Limiting Trace File Size

- Use smallest number of processors possible
- Use smallest input data set (e.g. number of timesteps)
- Limit trace file size
  - by environment variables / API functions / configuration files?
- Trace only selected parts by using control functions to switch tracing on/off
  - select important parts only
  - for iterative programs: trace some iterations only
- Trace only important events
  - no MPI administrative functions and/or MPI collective functions
  - only user function in call tree level 1 or 2
  - never functions called in (inner) loops

NOTE: Make sure to collect complete message traffic!

- For non-blocking (`MPI_I*`) include `MPI_Wait*` + `MPI_Test*`

Performance Optimization and Tuning Tutorials

- Designing and Building Parallel Programs (Ian Foster, ANL)
  http://www-unix.mcs.anl.gov/dbpp/
- ASCI Blue Training Sessions (LLNL)
  http://www.llnl.gov/computing/tutorials/workshops/workshop/
  (→ Select Tutorials)
  - MPI Performance Topics
  - Performance Analysis Tools
  - Optimization Topics
- Tuning MPI Applications for Peak Performance (ANL)
  http://www-unix.mcs.anl.gov/mpi/tutorial/perf/
- Optimization and Tuning of Uniprocessors, SMPs and MPPs, including OpenMP and MPI (Philip Mucci and Kevin London)
  http://www.cs.utk.edu/~mucci/MPPopt.html
- Performance Tuning (MHPCC)
  http://www.mhpcc.edu/doc/perf.html
- Look for SC98, SC99, and SC00 tutorials!

Debugging: Totalview

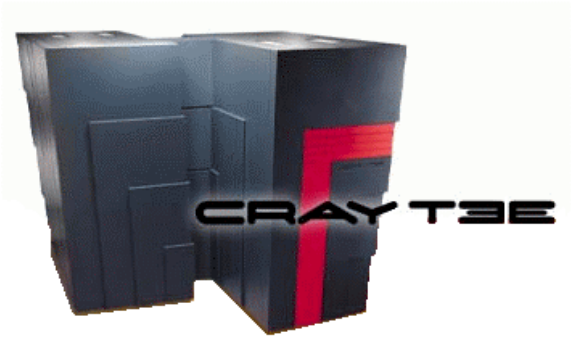
- Commercial product of ETNUS Inc.
- UNIX Symbolic Debugger
  for C, C++, f77, f90, PGI HPF, assembler programs
- „Quasi“ standard for debugger
- Selected as DOE “ASCI debugger”
- Special, non-traditional features
  - Multi-process and multi-threaded
  - C++ support (templates, inheritance, inline functions)
  - F90 support (user types, pointers, modules)
  - 1D + 2D Array Data visualization
  - parallel debugging
    - MPI (Vendor+MPICH: automatic attach, message queues) and PVM
    - OpenMP (Vendor, PGI, KAI)
    - HPF (HPF source level, distributed array access, distribution display)
    - pthreads (AIX, IRIX)

Totalview: Availability

- Supported platforms
  - IBM RS/6000 and SP
  - SGI workstations and Origin
  - Compaq Alpha workstations and Clusters
  - Sun Solaris Sparc + x86, Sun SunOS
  - HP HP-UX
  - Linux Intel x86 and Alpha
- Also available from vendor or 3rd party for
  - Cray T3E and PVP
  - Fujitsu VPP
  - QSW CS2
  - NEC SX4 + Cenju
  - Hitachi SR22xx
- Plans for NT
- http://www.etnus.com/products/totalview/

CRAY T3E: Performance Tools

- time / HW counter measurements
  - Performance Analysis Tool (PAT)
- flat profiling (sampling)
  - PAT
- call graph profiling
  - PAT
- detailed (basic block) profiling
  - Apprentice
- tracing
  - PAT / [Vampir]



PAT

- Features
  - Profiling: execution time profile for functions (sampling)
  - Call Site Reporting: call graph profiling (number of calls, execution time) of selected functions
  - HW Counter Measurement: e.g. Number of FLOPS
  - [Object-code instrumentation for tracing with Vampir]
- Advantages (especially compared to Apprentice)
  - simple, low overhead ⇨ for quick overview!
  - Does not need re-compilation, only re-linking
  - can analyze system and 3rd party libraries
- Disadvantages
  - can only analyze program as a whole
  - no (useable) GUI

PAT: Usage

- Prepare user application by linking with PAT library (`-lpat`) and PAT linker command file (`pat.cld`)

```
t3e% f90 *.o -o myprog -lpat pat.cld        # Fortran
t3e% cc *.o -o myprog -lpat pat.cld         # ANSI C
t3e% CC *.o -o myprog -lpat pat.cld         # C++
```

- If necessary, select T3E HW performance counter through environment variable $PAT_SEL

```
t3e% export PAT_SEL=FPOPS       # floating-point (default)
t3e% export PAT_SEL=INTOPS      # integer instructions
t3e% export PAT_SEL=LOADS       # load instructions
t3e% export PAT_SEL=STORES      # store instructions
```

- If necessary, change sampling rate (default 10000)

```
t3e% export PAT_SAMPLING_RATE=1000
```

[Value is in microseconds]

PAT: Performance Overview

- Execute application as usual (creates file: `myprog.pif`)

  t3e% mpprun -n ## myprog *-myoptions myargs*

  Start PAT

  t3e% pat -L myprog myprog.pif

- Execution time overview

  => time

```
Elapsed Time              1.129 sec   16 PEs
User    Time (ave)        2.090 sec   98%
System  Time (ave)        0.027 sec    1%
```

- HW performance counter report

  => perfcntr

```
Performance counters for FpOps (Values in MILLIONS)
PE  cycles operations  ops/sec  dmisses  misses/sec
 0  104.16     19.33     83.52     5.17     22.32
 1  194.00     20.19     46.84     7.01     16.27
 2  193.39     20.19     46.99     7.01     16.32
    ...
```

PAT: Profiling

- Function profiling report (sampling report):

  => profile

```
                                Percent    90% Conf. Interval
  VELO                            52%             1.4
  _shmem_long_wait                 7%             0.7
  CURR                             6%             0.7
  barrier                          5%             0.6
  MPI_RECV                         5%             0.6
  TEMP                             3%             0.5
  ...
```

- Instrument for “Call Site Reporting”

  => instcall VELO

  => quit

  generates “new a.out”

- Execute again:

  t3e% mpprun -n ## a.out *-myoptions myargs*

PAT: Profiling

- Execute PAT again for “Call Site” report:

```
t3e% pat -L a.out a.out.pif
  Interactive mode. Valid commands:
  csi hist instcall perfcntr profile quit time ...

  => csi
  Name  PE  called from  #calls  total time avg./call
  VELO:
        PE:0
                  CX3D      4     165.1102    41.2775
        PE:1
                  CX3D      4     166.7334    41.6834
        PE:2
                  CX3D      4     166.6664    41.6666
        PE:3
                  CX3D      4     165.3808    41.3452
        ...
```

Apprentice

- Tool for detailed profiling on basic block level
- Usage
  - Instrumentation by Cray compiler

    ```
    t3e% f90 -eA myprog.f -lapp
    t3e% cc -happrentice myprog.c -lapp
    ```

  - Executing program generates app.rif
  - Analyze performance

    ```
    t3e% apprentice app.rif
    ```

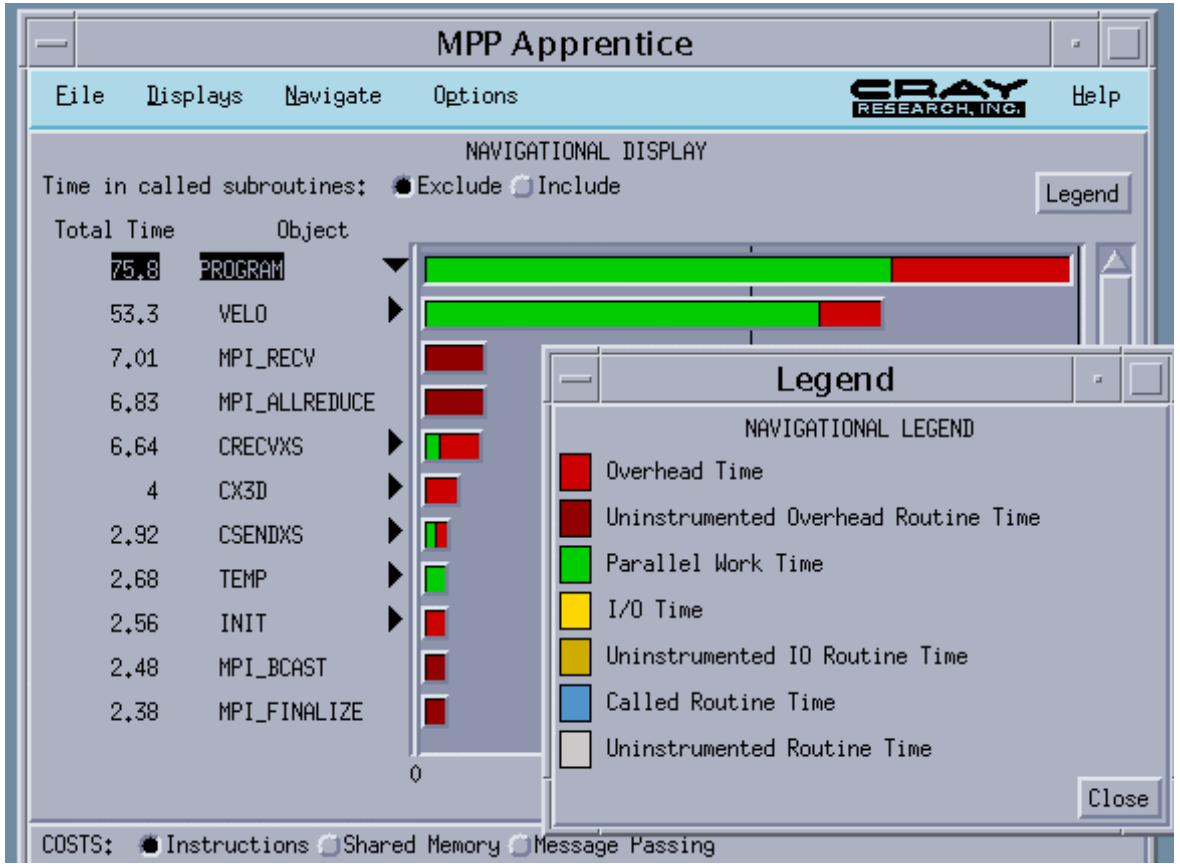
- Features
  - Provides summary data (sum over all PE and total execution time)
  - Provides
    - execution time and block counts (measured)
    - number of floating-point, integer, load, store instructions (calculated)
  - Automatically corrects measurement overhead

Apprentice: Navigation Window

- Shows time profile for
  - program
  - functions
  - basic blocks

  sorted by usage

- Items can be "expanded" by clicking on black triangle

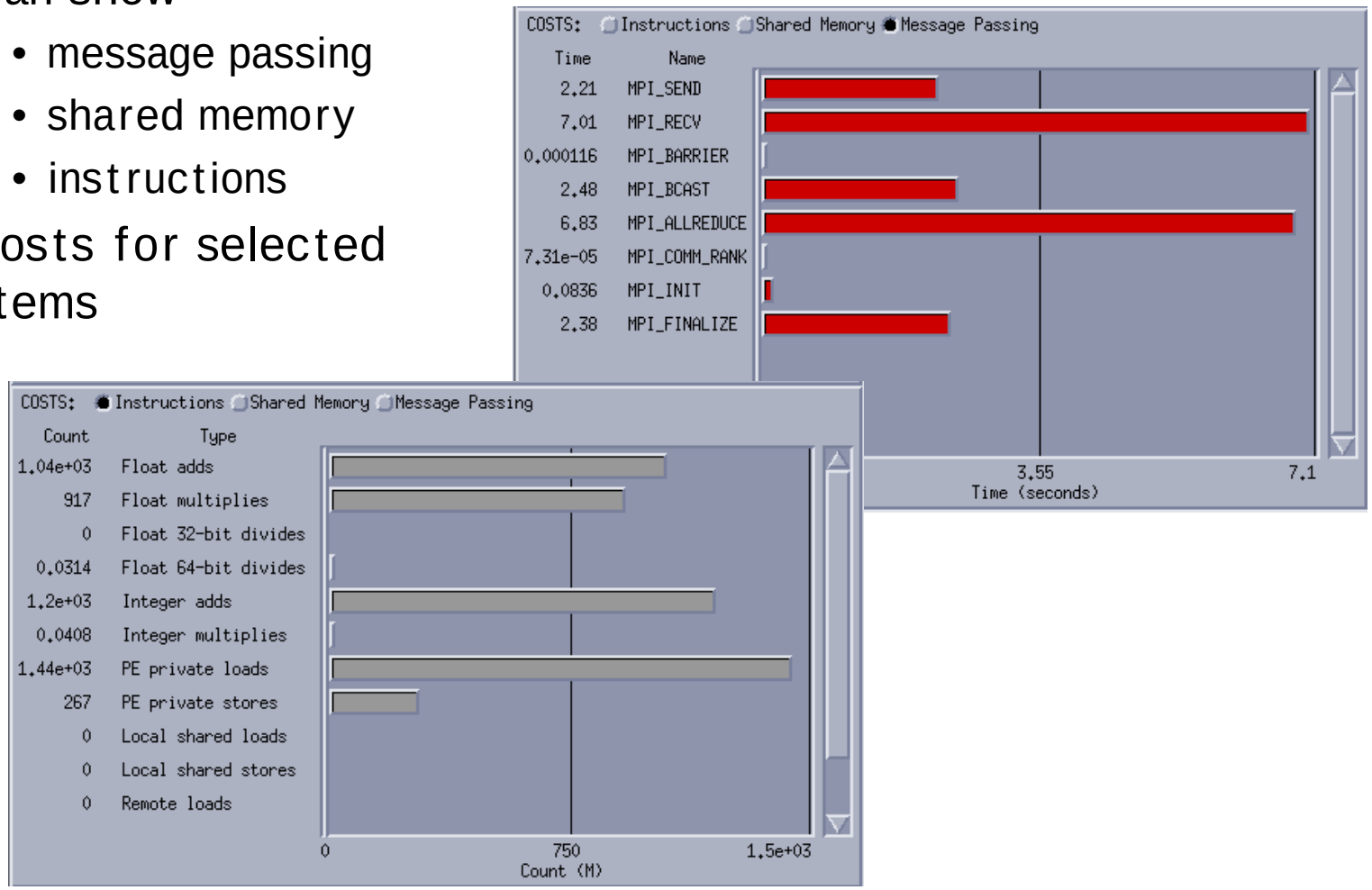


- Can show time including or excluding the time used by subregions

Apprentice: Costs Window

- Can show
  - message passing
  - shared memory
  - instructions

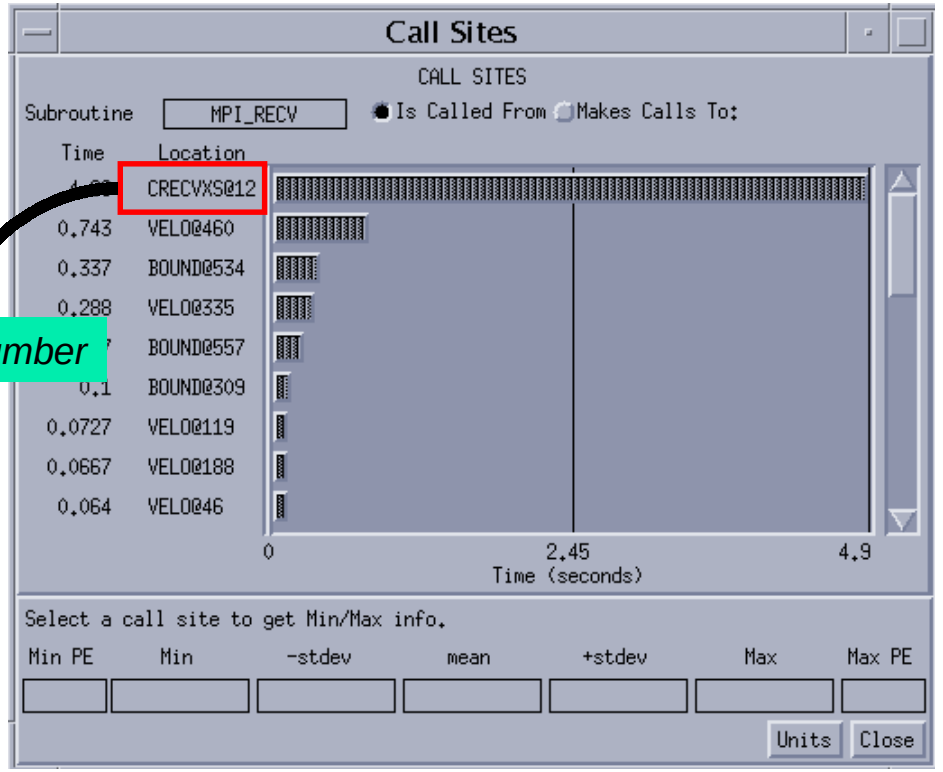
  costs for selected items



Apprentice: Finding Communication Bottleneck

- Compare values for 1st line (program) in navigation window:
  - Overhead (red) is communication time
  - parallel work (green) is calculation part
- Limiting MPI call ⇨ 1st MPI call in navigation window
- Select it, then
  - Menu Displays
  - Item Call sites
  - Option Is Called From

subroutine@linenumber

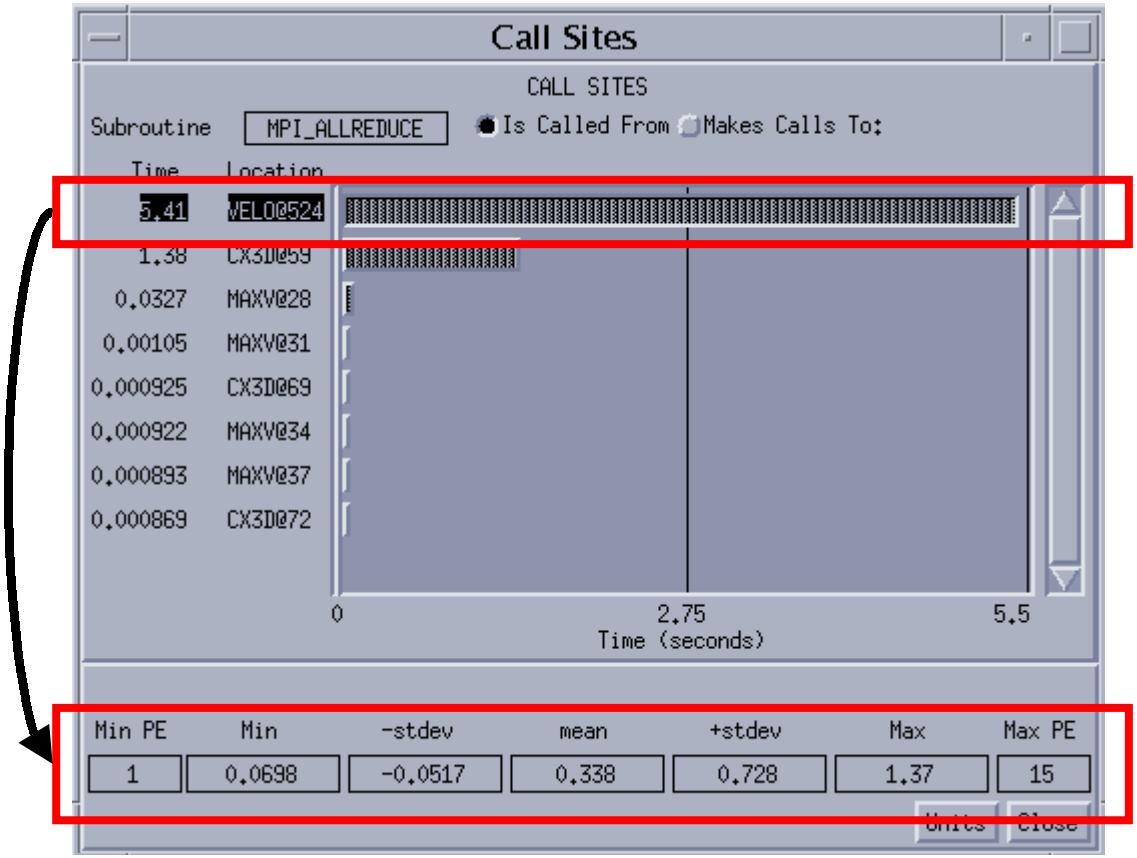


- To show source code
  - expand calling routine in navigation window
  - select call site

Apprentice: Finding Load Imbalance

- Select barrier or MPI collective communication routine in navigation window, then
  - Menu `Displays`
  - `Item` `Call sites`
  - Option `Is Called From`
- Select call site to see
  - minimum
  - mean
  - maximum

  execution time



T3E: Tracing

- There is no Cray tracing tool ⇨ use Vampir
- Necessary trace instrumentation can be done in two ways
  - Object code instrumentation with PAT
    - collaboration between Cray and Forschungszentrum Jülich
    - supports MPI-, PVM-, and SHMEM programs
    - additional manual source code instrumentation for selective tracing
  - Commercial Vampirtrace tracing library
    - supports MPI and SHMEM programs
    - additional manual source code instrumentation for recording of user subroutines and selective tracing
- See

  `http://www.fz-juelich.de/zam/RD/coop/cray/crayperf.html`

  for up-to-date description and additional tips and hints

PAT: Instrumentation for Tracing

- Prepare user application by linking with PAT library (`-lpat`), message passing wrapper library (`-lwrapper`) and PAT linker command file (`pat.cld`)

```
t3e% cc *.o -o myprog -lpat -lwrapper pat.cld
```

- Instrument for tracing

```
=> insttrace CX3D VELO barrier CURR MPI_Recv TEMP...
=> quit
```

- For real programs unusable
  better: create list of functions to instrument
  - one function per line
  - Note!!! names have to be specified as linker symbol names (i.e. Fortran: all capital letters; C++: “mangled names”)
  - ready-to-use lists of message passing routines at website
- Then use -B option for PAT to instrument executable

```
t3e% pat -L -B MYLIST myprog
```

PAT: Trace Generation

- If necessary, parameters for trace recording can be specified through environment variables

```
t3e% export PAT_PROFILE_WHILE_TRACING=OFF  # default: on
t3e% export PAT_NUM_TRACE_ENTRIES=20000    # 8192
t3e% export PAT_TRACE_LIMIT=20000          # unlimited
t3e% export PAT_TRACE_COLLECTIVE=1         # 0
```

- Execute program again (to generate pif file)

```
t3e% mpprun -n ## a.out -myoptions myargs
```

- Convert PAT output into trace format of Vampir [pif2bpv available at website]

```
t3e% pif2bpv a.out.pif mytrace.bpv
```

- Analyze trace file with Vampir

```
t3e% vampir mytrace.bpv
```

PAT: Trace File Size Reduction

- Manually insert functions (defined in `pat.h` and `pat.fh`)

- for selective tracing

```
C/C++                 Fortran
TRACE_OFF();          PIF$TRACEOFF()        # Turn tracing off
TRACE_ON();           PIF$TRACEON()         # Turn tracing on
```

- for trace size limitation

```
TRACE_LIMIT(li);  PIF$TRACELIMIT(li)   # Set limit to li
                                       # trace records
                                       # -1 ⇨ unlimited
```

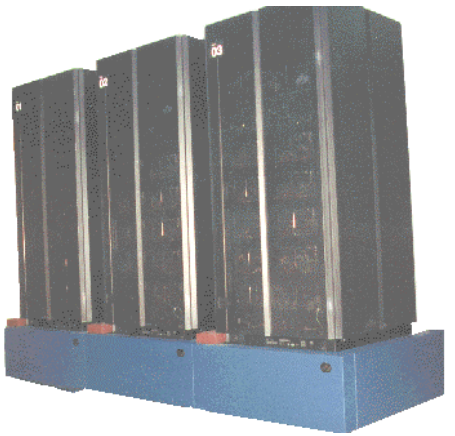
Pat stops tracing after recording *li* records

CRAY T3E: URLs

- Cray Research Online Software Publications Library
  `http://www.cray.com/swpubs/`
  - CRAY T3E C and C++ Optimization Guide, 004-2178-002
  - CRAY T3E Fortran Optimization Guide, 004-2518-002
  - Introducing the MPP Apprentice Tool, IN-2511 3.0
- Cray T3E vendor Information
  `http://www.cray.com/products/systems/crayt3e/`
- White Papers
  `http://www.cray.com/products/systems/crayt3e/papers.html`
- Forschungszentrum Jülich Cray Performance Tools Page
  `http://www.fz-juelich.de/zam/RD/coop/cray/crayperf.html`
- Various T3E documents and talks about optimization
  `http://www.fz-juelich.de/zam/docs/craydoc/t3e/`

IBM SP: Performance Tools

- flat profiling (sampling)
  - [prof]
  - gprof
- call graph profiling
  - gprof / Xprofiler
- detailed (source line) profiling
  - Xprofiler
- tracing
  - VT



Xprofiler

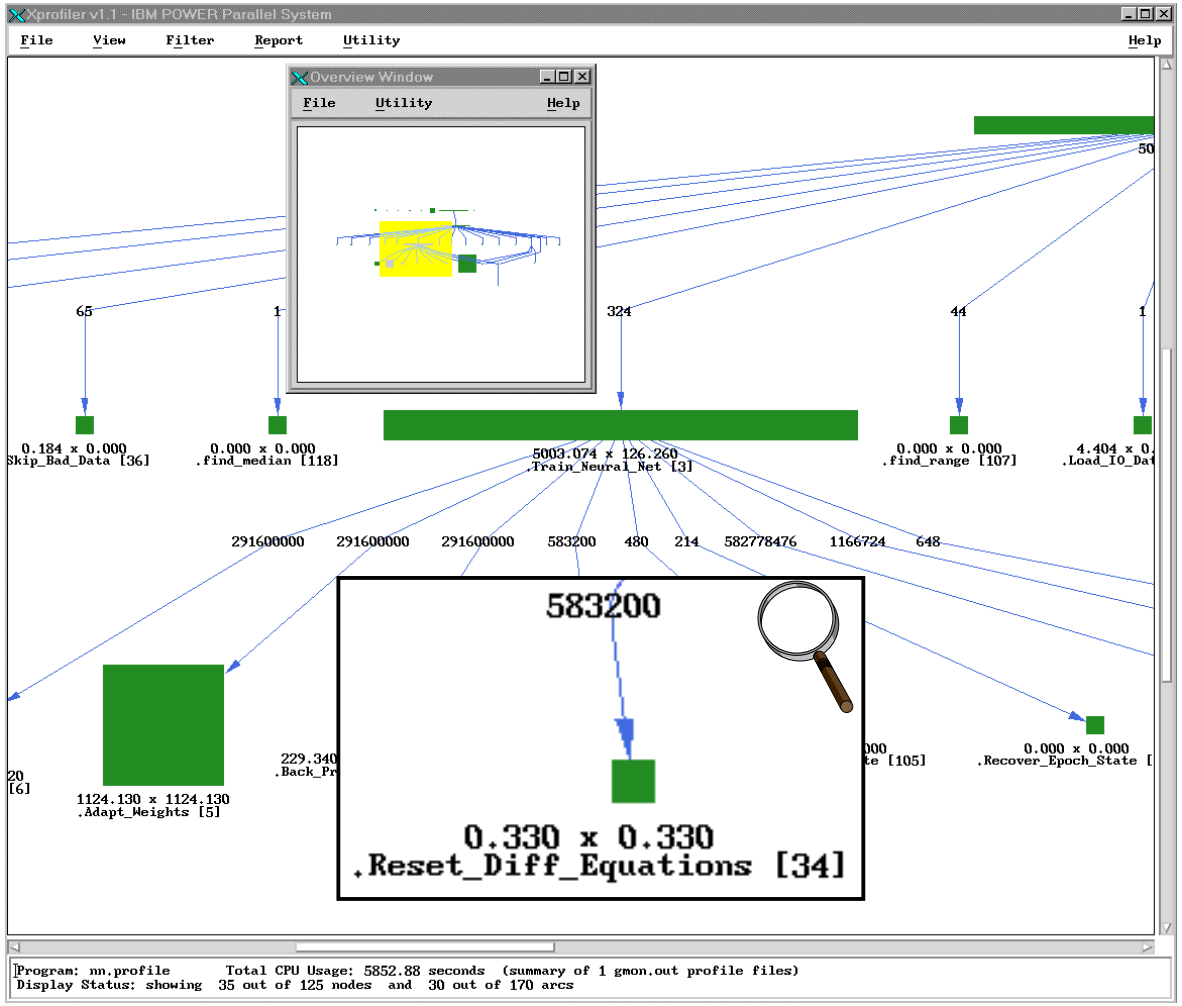
- Gprof: time sampled call graph profiling
- Xprofiler: graphical analysis interface for gprof outputs
- Usage
  - Compile and link with "-g -pg" flags and optimization
  - Run the code; generates gmon.out.*N* files (N = 0 .. P-1)
  - Analyze with gprof or xprofiler

```
sp% gprof gmon.out.* > gprof.report
sp% xprofiler myprog gmon.out.*
```

- Features
  - graphical call graph performance display
    - execution time bottleneck location
    - call sequence identification
  - gprof reports
  - source line profiling

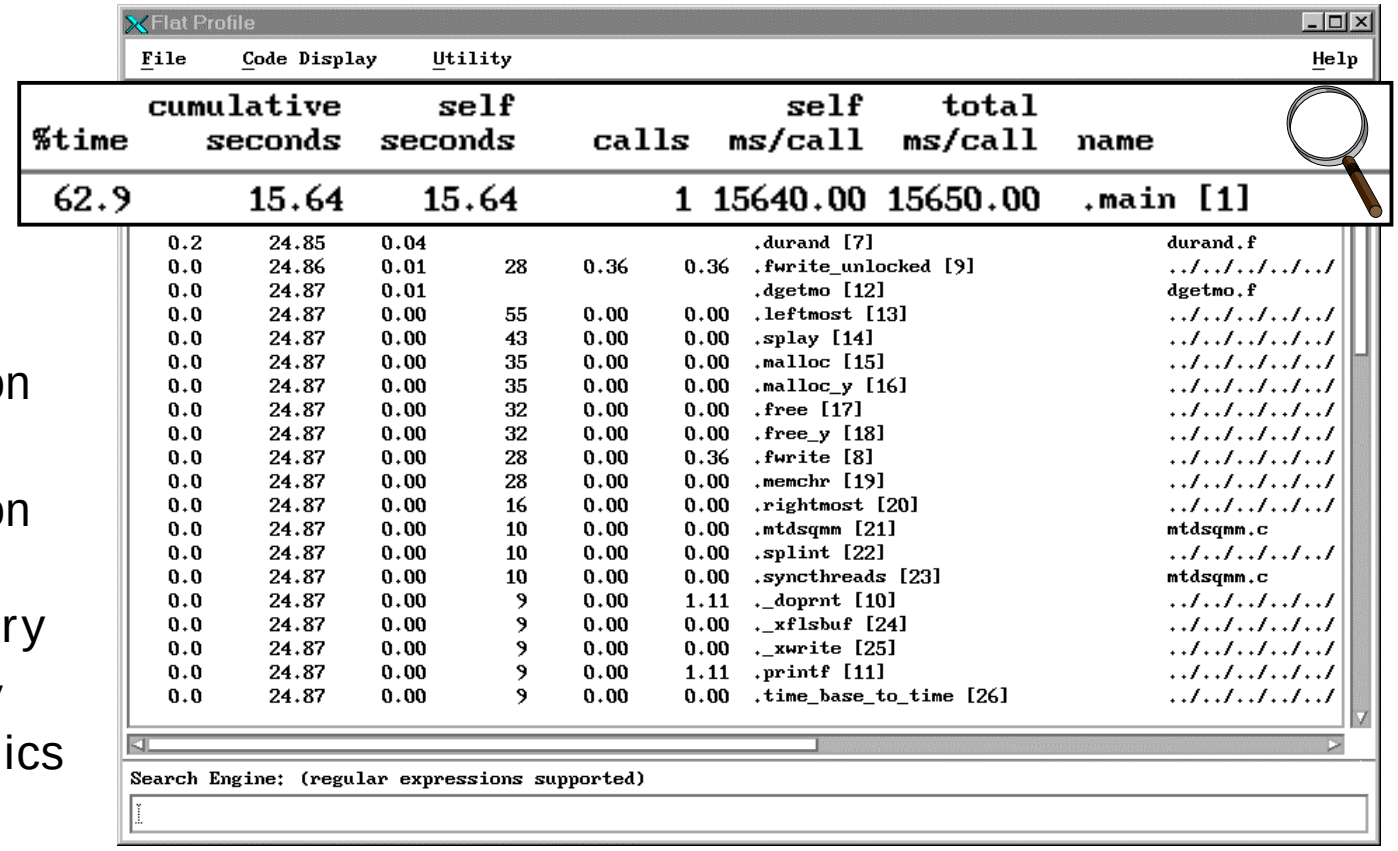
Xprofiler: Main Display

- width of a bar
  ∝ time including called routines
- height of a bar
  ∝ time excluding called routines
- call arrows labeled with number of calls
- Overview window for easy navigation (View → Overview)



Xprofiler: Gprof Reports

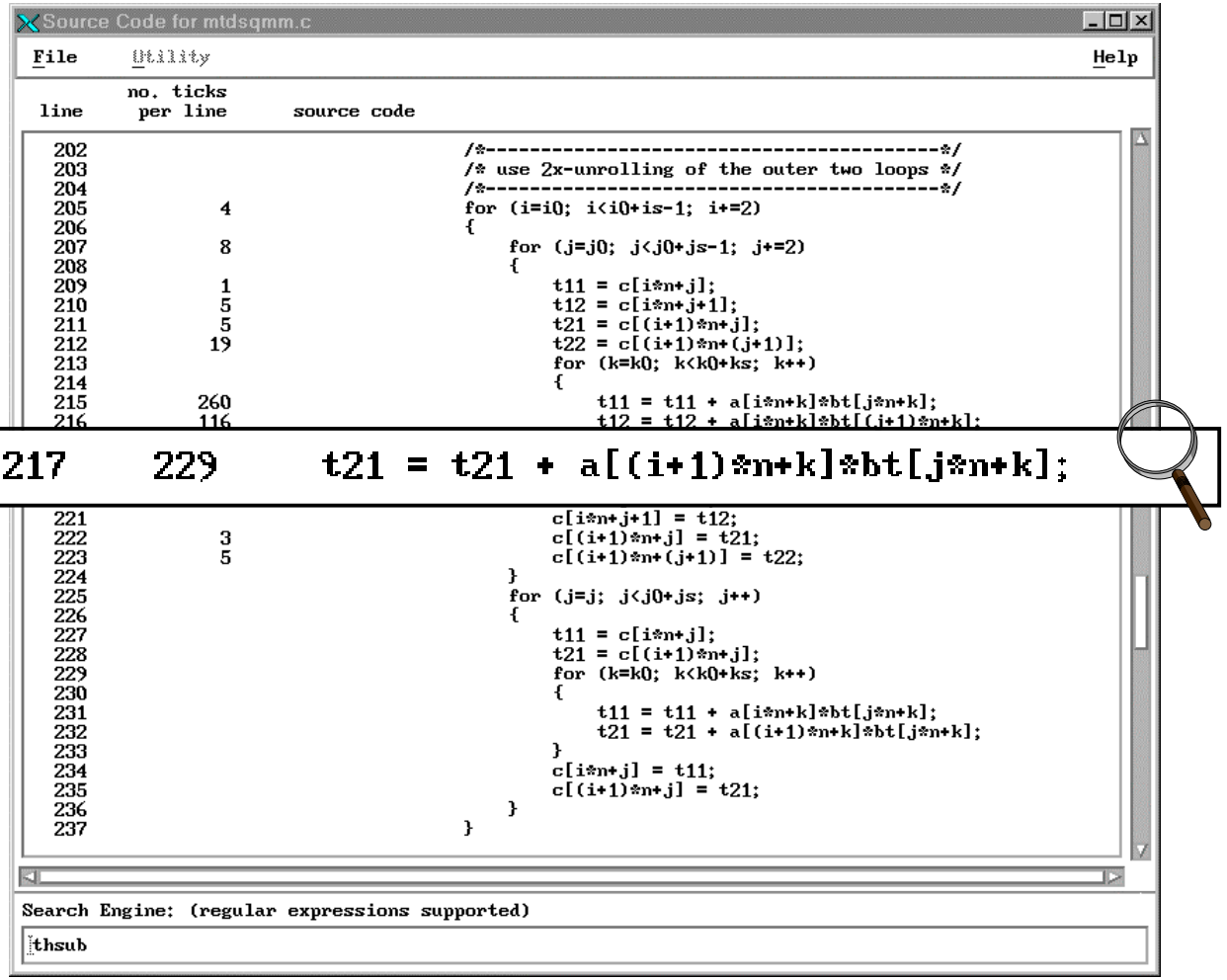
- Menu Report provides usual gprof reports plus some extra ones
  - Flat Profile
  - Call Graph Profile
  - Function Index
  - Function Call Summary
  - Library Statistics



Xprofiler: Source Code Window

- Source code window displays source code with time profile (in ticks=.01 sec)

- Access
  - select function in main display
    → context menu
  - select function in flat profile
    → Code Display
    → Show Source Code



Xprofiler: Tips and Hints

- Simplest when gmon.out.*, executable, and source code are in one directory
- By default, call tree in main display is “clustered”
  - Menu `Filter` → Item `Uncluster Functions`
  - Menu `Filter` → Item `Hide All Library Calls`
- Libraries must match across systems!
  - on measurement SP nodes
  - on workstation used for display!
- Must sample realistic problem (sampling rate is 1/100 sec)
- PVM programs require special hostfile (specify different wd’s otherwise gmon.out files overwrite each other)

Xprofiler: Sample PVM Hostfile for Profiling

```
#host configuration for v08 nodes
#
#executable path for all hosts
*
ep=$PVM_ROOT/bin/$PVM_ARCH:$PVM_ROOT/lib/$PVM_ARCH:
   $HOME/bin:$PVM_ROOT/lib
v08l01 wd=/cfs/profile/01
v08l02 wd=/cfs/profile/02
v08l03 wd=/cfs/profile/03
v08l04 wd=/cfs/profile/04
v08l05 wd=/cfs/profile/05
v08l06 wd=/cfs/profile/06
...
```

Alternative : add a `chdir()` statement to the procedure code.

VT

- MPI message passing visualization tool
- No longer supported starting with Parallel Environment 3.1
- No re-compiling or re-linking necessary!
- Trace visualization through animation
- Does not use AIX tracing library (but own one)

- Trace generation by setting trace level with option `-tlevel`

  sp% `poe myprog -procs ## -tlevel` *`N -myoptions myargs`*

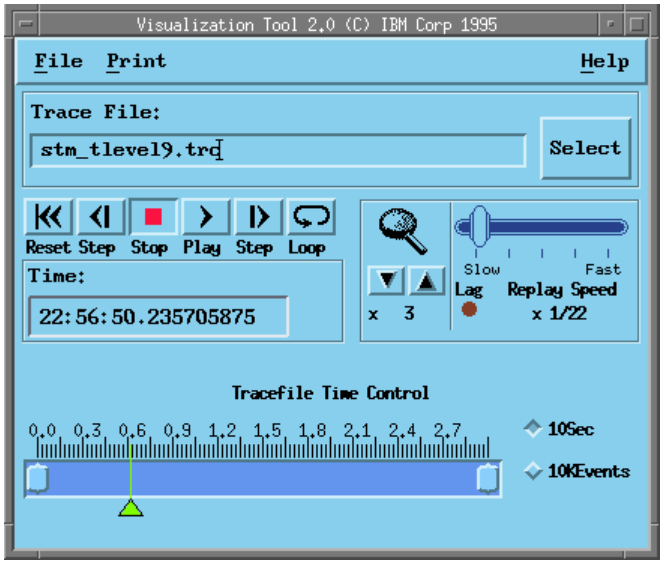
  or environment variable `MP_TRACELEVEL`

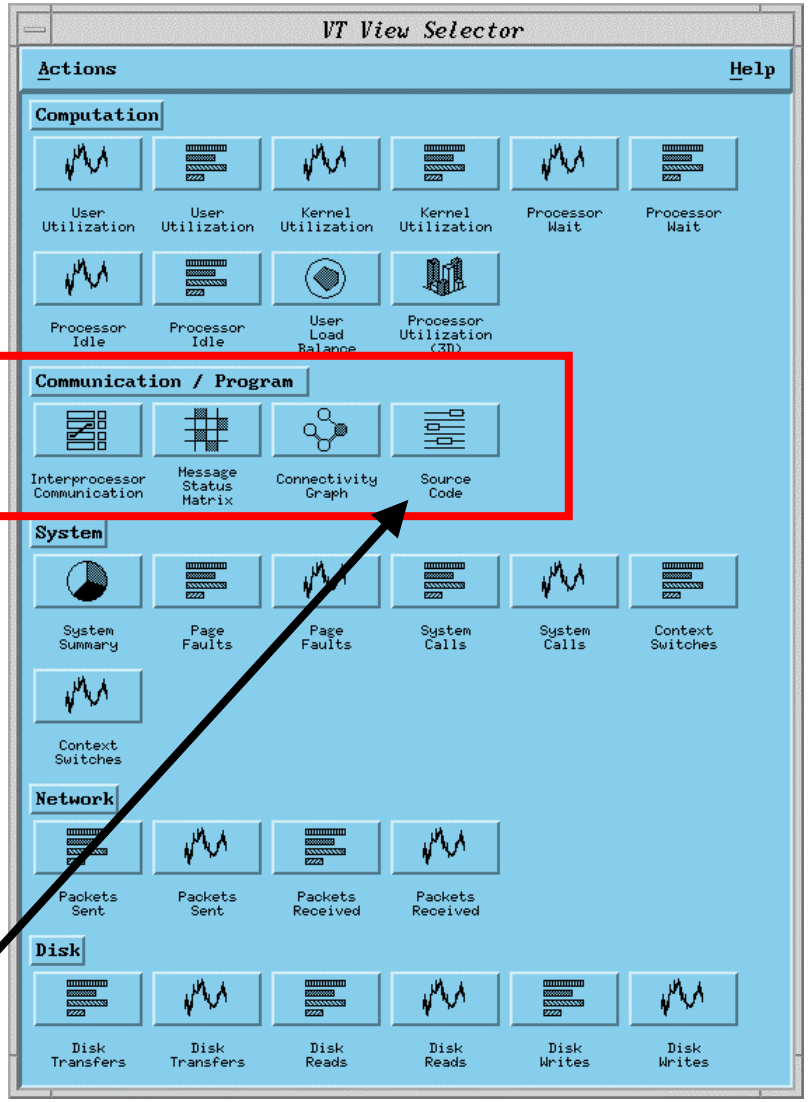
- Events / Trace level

Events	Trace level
application markers	1,2,3,9
kernel statistics	2,9
message passing	3,9

VT Displays

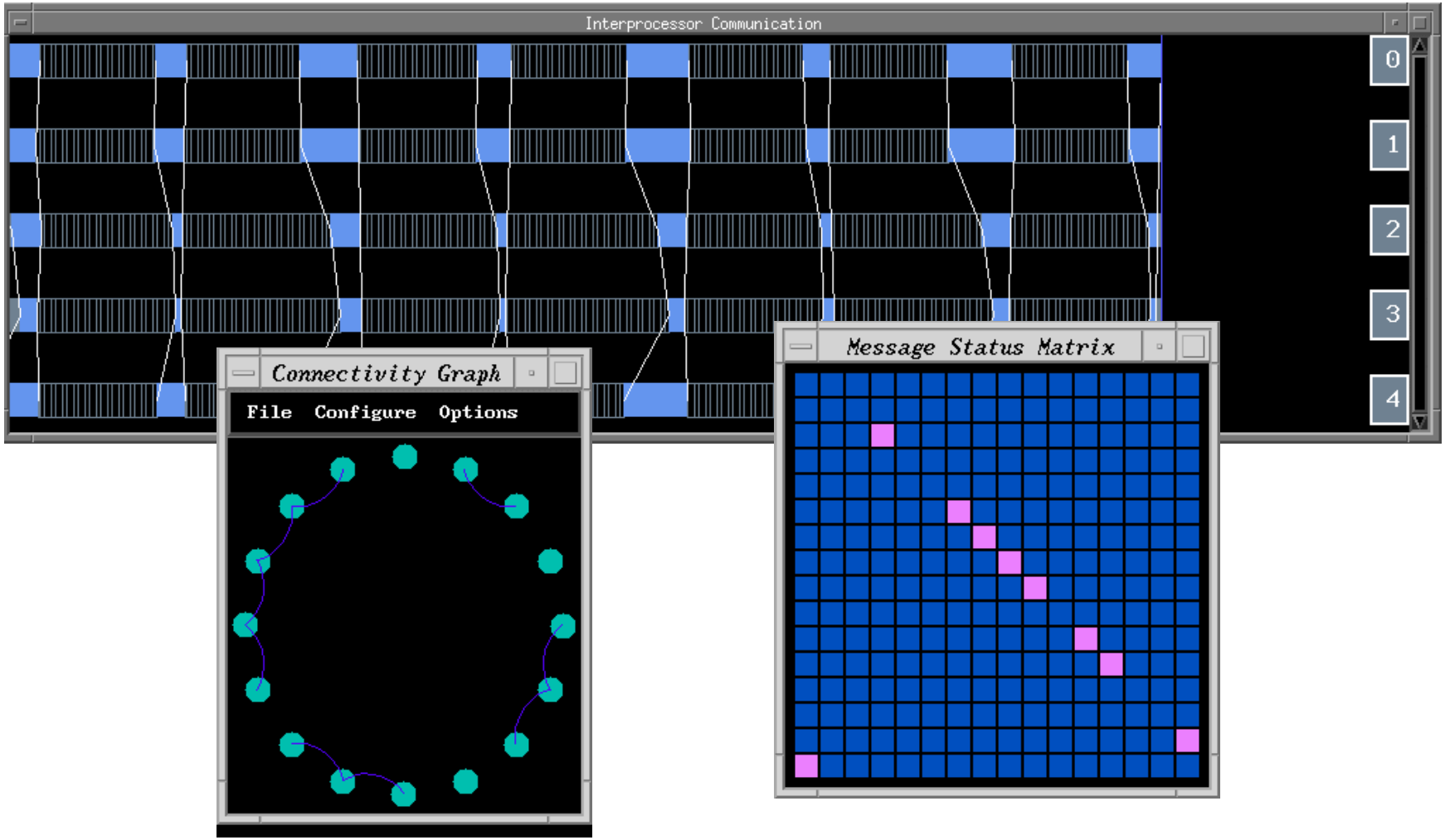
- VT provides a large variety of displays for system and parallel program performance
- Trace Control Window allows control of trace and display animation





- Source display requires -g

VT Communication Displays



VT: Trace Recording Control

- Selective tracing

C/C++	Fortran
`VT_trc_start_c(level);`	`VT_TRC_START(level, ierr)`
`VT_trc_stop_c();`	`VT_TRC_STOP(ierr)`

- Application marker

`mpc_marker(...)`	`MP_MARKER(light, col, str)`

- Trace size limitation
  - Command line options
    - `tbuffsize` *`size`* `# default 1M`
    - `tpermsize` *`size`* `# 10M`
  - Environment variables

```
sp% export MP_TBUFFSIZE=size
sp% export MP_TPERMSIZE=size
```

IBM SP: Future Tools

- Dynamic Probe Class Library (DPCL)
  - dynamic instrumentation of running serial or parallel programs
  - foundation for a new family of tools
  - http://www.ptools.org/projects/dpcl
- Performance Hardware Monitor API (PMAPI)
- Unified Trace Environment (UTE)
  - uses AIX trace facility and standard MPI profiling interface
  - supports MPI event groups (point-to-point, collective, ...)
  - uses Jumpshot for trace visualization
  - http://www.research.ibm.com/people/w/wu/UTE.html
- SvPablo
  - graphical source code instrumentor for C and Fortran
  - graphical performance analysis and presentation tool
  - captures HW + SW statistics per functions / loops / statements

IBM SP: URLs

- RS/6000 SP Resource Center
  `http://www.rs6000.ibm.com/support/sp/resctr/index.html`
- Parallel Environment - Product Information
  `http://www.rs6000.ibm.com/software/sp_products/pe.html`
- Parallel Environment - Online Documentation
  `http://www.rs6000.ibm.com/resource/aix_resource/sp_books/pe/`
  - SA22-7426: PE Operation and Use Vol. 2 (describes xprofiler + vt)
  - SA22-7420: DPCL Programming Guide
  - SA22-7421: DPCL Class Reference
- IBM Redbooks
  `http://www.redbooks.ibm.com`
  - RS/6000 Scientific and Technical Computing: POWER Introduction and Tuning Guide, SG24-5155
  - RS/6000 SP: Practical MPI Programming, SG24-5380
- White Papers and Technical Reports
  `http://www.rs6000.ibm.com/resource/technology/index.html#sp`

IBM SP: URLs

- IBM SP Scientific Computing User Group, SCICOMP
  `http://www.spscicomp.org/`
- Advanced Computing Technology Center (ACTC)
  `http://www.research.ibm.com/actc/`
- POWERPARALLEL User Group
  `http://spud-web.tc.cornell.edu/HyperNews/get/SPUserGroup.html`
- Maui High Perf. Computing Center Performance Tuning Pages
  `http://www.mhpcc.edu/doc/perf.html`

SGI Origin: Performance Tools

- time / HW counter measurements
  - timex, perfex, SpeedShop
- flat profiling (sampling)
  - SpeedShop, Cvperf
- call graph profiling
  - SpeedShop, Cvperf
- detailed (basic block) profiling
  - SpeedShop, Cvperf
- data / memory profiling
  - dprof records memory access information
  - dlook displays placement of memory pages
- tracing
  - SpeedShop / [Vampir]

Timex

- Command line tool for measuring
  - elapsed, user, and system time
  - processor accounting data (see `man timex`)

- Usage

  `irix%` `timex` *`options`* `myprog` *`-myoptions myargs`*

- Example

```
irix% timex myprog
==> setup ...
==> calculation ...

real     1.63
user     0.85
sys      0.58
```

Perfex

- Command line tool for accessing MIPS hardware counters (R10000 + R12000 only)
- Usage

  irix% perfex *options* myprog *-myoptions myargs*
- Features
  - exact count of one or two events (option `-e` *num*)
  - approximate count of all countable events by multiplexing (`-a`)
  - can include analytic output (`-y`)
  - use signals to control start/stop (`-s`)
  - per-thread + total report for multprocessing programs (`-mp`)
  - help (includes list of available counters) (`-h`)
- No re-compilation or re-linking necessary!
- Selective profiling with calls to perfex library

Perfex: MIPS HW Counter

- R10000 HW Counters
  - 0 = Cycles
  - 1 = Issued instructions
  - 2 = Issued loads
  - 3 = Issued stores
  - 4 = Issued store conditionals
  - 5 = Failed store conditionals
  - 6 = Decoded branches
  - 7 = Quadwords written back from secondary cache
  - 8 = Correctable secondary cache data array ECC errors
  - 9 = Primary (L1) instruction cache misses
  - 10 = Secondary (L2) instruction cache misses
  - 11 = Instruction misprediction from secondary cache way prediction table
  - 12 = External interventions
  - 13 = External invalidations
  - 14 = Virtual coherency conditions
  - 15 = Graduated instructions
  - 16 = Cycles
  - 17 = Graduated instructions
  - 18 = Graduated loads
  - 19 = Graduated stores
  - 20 = Graduated store conditionals
  - 21 = Graduated floating point instructions
  - 22 = Quadwords written back from primary data cache
  - 23 = TLB misses
  - 24 = Mispredicted branches

Perfex: MIPS HW Counter

- R10000 HW Counters (cont.)
  - 25 = Primary (L1) data cache misses
  - 26 = Secondary (L2) data cache misses
  - 27 = Data misprediction from secondary cache way prediction table
  - 28 = External intervention hits in secondary cache (L2)
  - 29 = External invalidation hits in secondary cache
  - 30 = Store/prefetch exclusive to clean block in secondary cache
  - 31 = Store/prefetch exclusive to shared block in secondary cache
- R12000 HW Counter Differences to R10000
  - 1 = Decoded instructions
  - 2 = Decoded loads
  - 3 = Decoded stores
  - 4 = Miss handling table occupancy
  - 6 = Resolved conditional branches
  - 14 = Always 0
  - 16 = Executed prefetch instructions
  - 17 = Prefetch primary data cache misses

Perfex Example: Getting MFlops

```
irix% perfex -y -e 21 myprog
  ==> setup ...
  ==> calculation ...
  Summary for execution of myprog
                    Based on 195 MHz IP27
                        MIPS R10000 CPU
                        Counter  Typical Minimum Maximum
  Event Counter Name     Value   Time(s) Time(s) Time(s)
  ================================================================
   0 Cycles.......... 5158743137 26.455  26.455  26.455
-e 21 Floating point.. 3651000162 18.723   9.361 973.600

  Statistics
  ================================================================
  Floating point instructions/cycle.........   0.707731
  MFLOPS (average per process)..............  138.007459
```

SpeedShop

- Program profiling with three different methods
  - statistical sampling based on different time bases
  - counting (based on dynamic instrumentation with pixie)
  - exception traces (fpe, mpi, I/O)
- No re-compilation or re-linking necessary!
- Supports
  - unstripped executables, shared libraries
  - C, C++, f77, f90, Ada, Power Fortran, Power C
  - sproc/fork parallelism, pthreads, MPI, PVM, shmem, OpenMP
  - lots of different experiments (see man speedshop, ssrun, prof)
- Usage
  - run experiments on the executable: `ssrun -exp [-workshop]`
  - generates file *executable.experiment.pid*
  - examine performance data: `prof/cvperf ssrun.outfile`
- Selective profiling with calls to ssapi library

SpeedShop: Sampling Experiments

- Sampling based on
  - actual elapsed time (-totaltime, -usertime, -pcsamp, -cy_hwc)
    - finds the code where the program spends the most time
    - use to get an overview of the program and to find major trouble spots
  - instruction counts (-gi_hwc, -gfp_hwc)
    - finds the code that actually performs the most instructions
    - Use to find the code that could benefit most from a more efficient algorithm
  - data access (-dc_hwc, -sc_hwc, -tlb_hwc)
    - finds the code that has to wait for its data to be brought in from another level of the memory hierarchy
    - Use these to find memory access problems
  - code access (-ic_hwc, -isc_hwc)
    - finds the code that has to be fetched from memory when it is called
    - Use these to pinpoint functions that could be reorganized for better locality, or to see when automatic inlining has gone too far

SpeedShop: Other Experiments

- ideal
  - provides exact measurements of executed basic blocks
  - reports cycle count (ideal time) ⇨ ignores cache misses, overlap...
  - execution slowdown: factor 3 to 6
- io
  - traces calls to UNIX i/o system calls e.g. `read`, `write`, ...
- mpi
  - traces calls to MPI point-to-point, MPI_Barrier, and MPI_Bcast
  - generates file viewable with cvperf
  - ssfilter can produce Vampir traces from "-mpi" output (beta)
- fpe
  - floating-point exceptions trapped in HW, but ignored by default
  - collects data on all floating-point exceptions
  - exception free code allows for higher level of optimization

SpeedShop: Performance Analysis

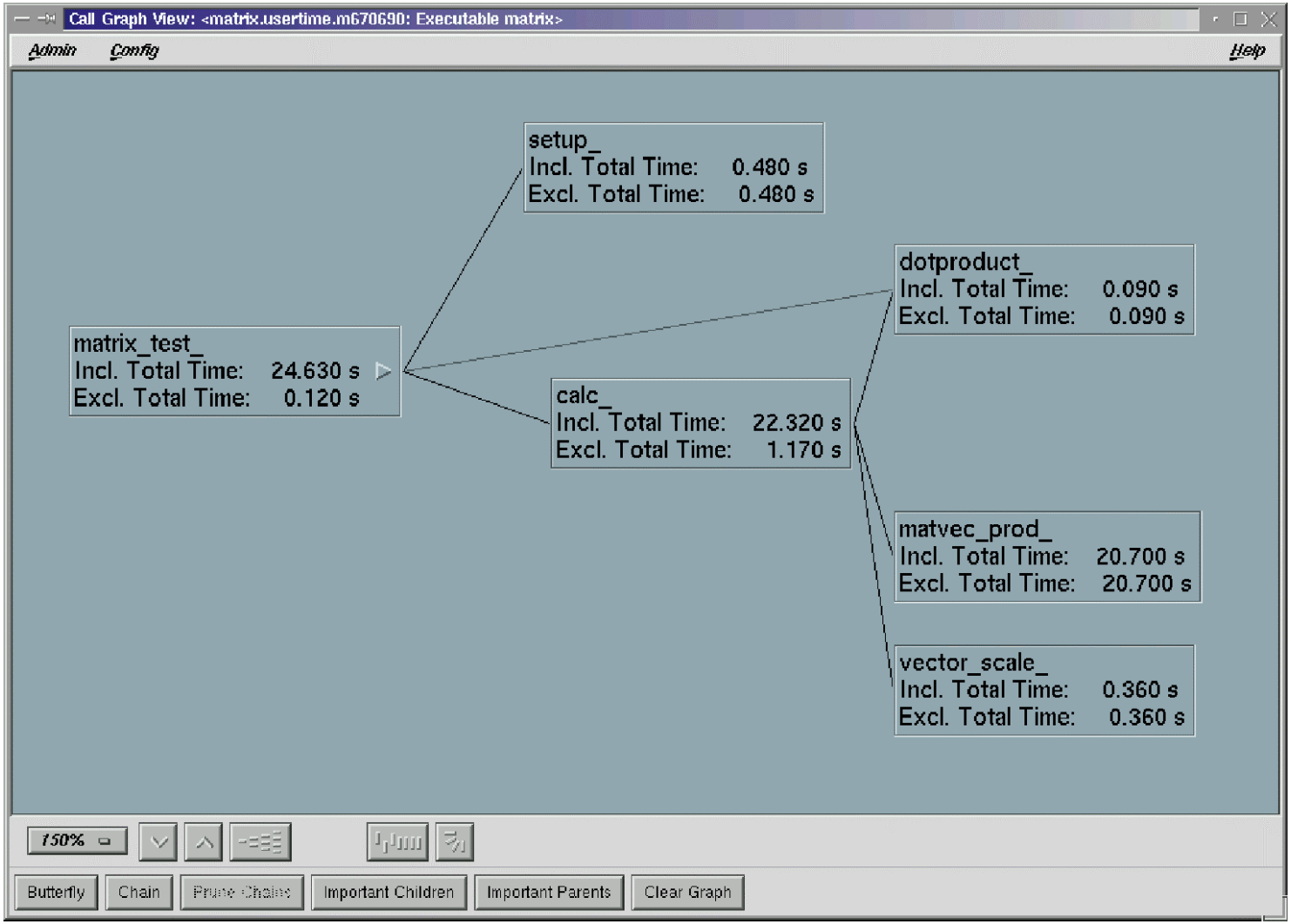
- Prof
  - command line performance analyzer
  - important options
    - heavy — reports most heavily used lines in each function
    - quit n or n% — truncates after the first n lines / n%
    - butterfly — call graph profiling (only for some experiments)
    - inclusive — sort by inclusive data (default: exclusive)
    - usage — also print system statistics report
  - compiler + linker feedback file generation

- Cvperf
  - Visual Workshop Performance Analyzer
  - GUI front-end
  - performance experiment and option specification
  - performance analysis specification and performance views

SpeedShop Example: usertime Sampling

```
irix% ssrun -usertime myprog
irix% prof myprog.usertime.m95674
  ...
  ------------------------------------------------
  function list, in descending order by exclusive time
  ------------------------------------------------
  [index] excl.s excl.%   cum%  incl.s  inc.% procedure
      [5] 10.410  71.8%  71.8%  10.410  71.8% MATVEC
      [6]  2.400  16.6%  88.4%   2.400  16.6% SETUP
      [7]  0.900   6.2%  94.6%   0.900   6.2% SCALE
      [8]  0.780   5.4% 100.0%   0.780   5.4% DOTPROD
      [1]  0.000   0.0% 100.0%  14.490 100.0% __start
      [2]  0.000   0.0% 100.0%  14.490 100.0% main
      [3]  0.000   0.0% 100.0%  14.490 100.0% MAT_VEC_PR
      [4]  0.000   0.0% 100.0%  12.060  83.2% CALC
  ...
```

Cvperf: Call Graph View



Analyzing Parallel Applications: perfex

- OpenMP, pthreads, sproc:
  - use `-mp` option
  - outputs information collected for each thread followed by all summed up
  - watch
    - load imbalance (counter 21: floating point instructions)
    - excessive synchronization (counter 4: store conditionals)
    - false sharing (counter 31, shared cache blocks)
- MPI:
  - **`irix%`** `mpirun -np # perfex -operf.out` *`myprog myargs`*

Analyzing Parallel Applications: speedshop

- OpenMP, pthreads, sproc:
  - data collection for each thread automatically
- MPI:

  irix% mpirun -np # ssrun -exp *myprog myargs*
- Analyze files with prof or cvperf

- NEW for speedshop Version > 1.4.1:
  - Run ideal experiment

    irix% ssrun -ideal *myprog myargs*
  - Combine speedshop experiment files into one file

    irix% ssaggregate -e *myprog.ideal.** -o *myprog.total*
  - Report programming model (OpenMP, MPI, pthread) specific overheads with prof

    irix% prof -overhead *myprog.total*

Speedshop Example: Overhead Report

```
...
... Normal ideal experiment report here
...

-----------------------------------------------------
OpenMP Report
-----------------------------------------------------
       Parallelization Overhead: 00.000%
                  Load Imbalance: 00.076%
       Insufficient Parallelism: 69.085%
                   Barrier Loss: 00.002%
              Synchronization Loss: 00.000%
  Other Model-specific Overhead: 00.000%
```

- Reported categories depend on programming model

SGI Origin: URLs

- SGI Origin2000 and Origin3000 Pages
  http://www.sgi.com/origin/
- SGI Online Publication Library
  http://techpubs.sgi.com/library
  - MIPSpro C, C++, Fortran Programmer's Guides
  - MIPSpro Compiling and Performance Tuning Guide
  - Developer Magic: ProDev Workshop MP User's Guide
  - Developer Magic: Performance Analyzer User's Guide
  - SpeedShop User's Guide
  - O2K and Onyx2 Performance Tuning and Optimization Guide
- NCSA: Frequently Used Tools and Methods for Performance Analysis and Tuning on SGI systems
  http://www.ncsa.uiuc.edu/SCD/Perf/Tuning/Tips/

3rd Party: Performance Tools

- Profiling (of MPI) Programs
  - DEEP (PSR)
- Tracing of MPI Programs
  - Vampir (Pallas)
  - Vampirtrace (Pallas)
- Performance Prediction
  - Dimemas (CEPBA)
- Performance Analysis of OpenMP Programs
  - GuideView (KAI)

DEEP

- Commercial product of
  Veridan -- Pacific-Sierra Research
- DEvelopment Environment
  for Parallel programs
  - ⇒ DEEP/MPI
  - ⇒ DEEP/PAPI (Hardware Counter Profiling)
  - DEEP for SMP, OpenMP, HPF, DPC (with VAST)
- Supports Fortran77/90, C programs (C++ soon)
- Integrated, cross-platform, parallel development environment
  at the user's source code level
  - Program Structure Browsing
  - ⇒ Profiling
  - Debugging

- http://www.psrv.com/deep.html

DEEP: PAPI and MPI Support

- PAPI (portable library for accessing hardware perf. counters)
  - User can select counters to be profiled
  - Counts are available at loop and routine level
  - Automatically graphed and included in tables
  - Support for counter multiplexing

- MPI
  - MPI Message load balance
  - MPI call site profiling
  - Collection of wait time in MPI calls
  - MPI summary for each routine in the user program
  - MPI calls are highlighted in call tree and code abstract views

DEEP/MPI: Usage

- Compile program with DEEP profiling driver
  - Just replace calls to `mpi*` compiling scripts with `mpiprof`

```
% mpiprof myprog.f ...
% mpiprof myprog.c ...
```

  - Generates static program information files in deep subdirectory

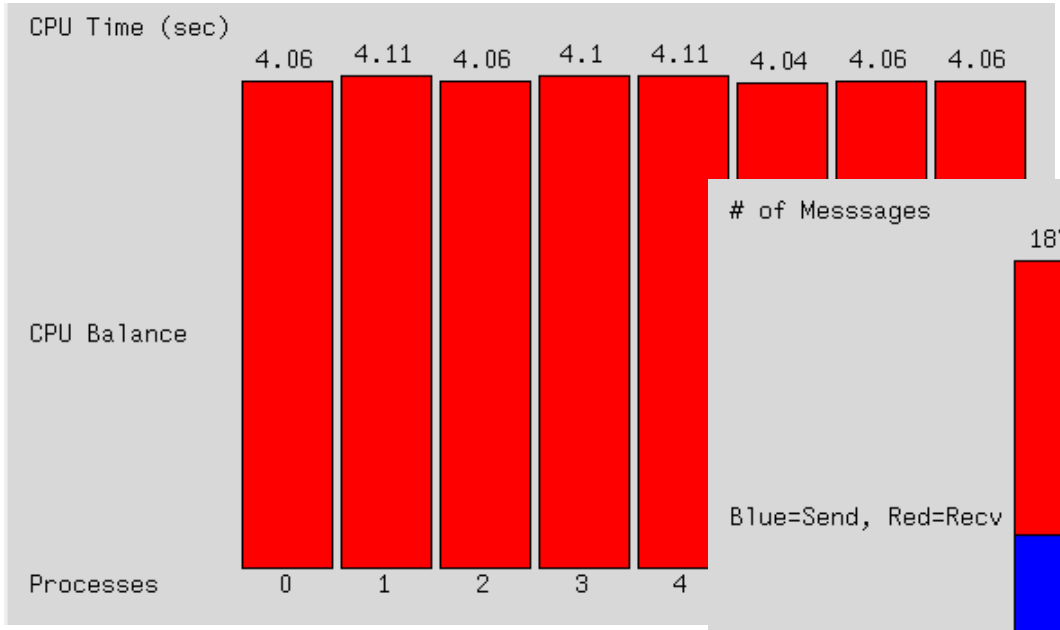
- Run program as usual

```
% mpirun -np # myprog
```

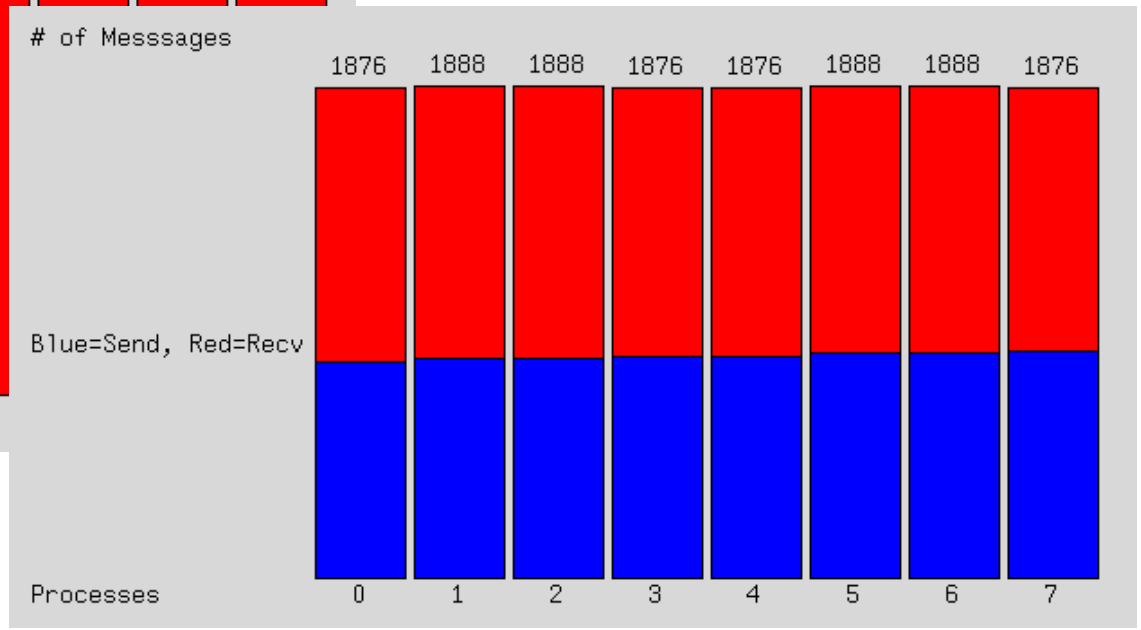
  - Generates dynamic profiling measurement files in deep subdirectory

- Analyze static and dynamic data [possibly on other platform]
  - % deep
    - provide path of source code and DEEP information
    - select run (if needed)
    - view information

DEEP/MPI: Global Balance Charts



- CPU Wallclock time balance

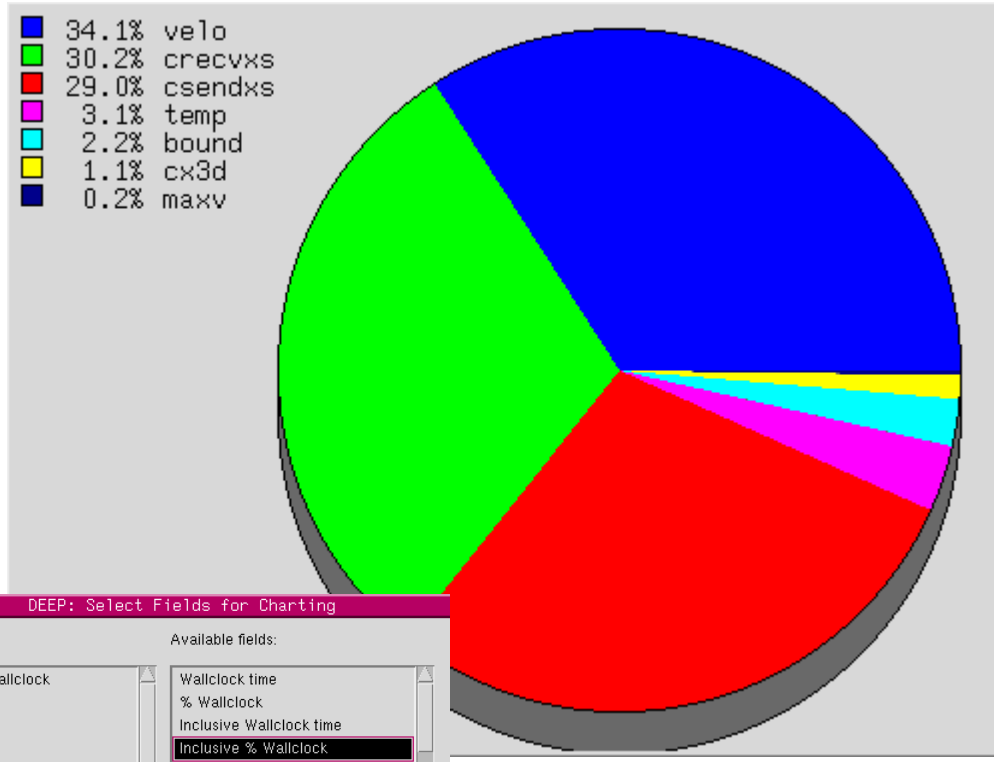


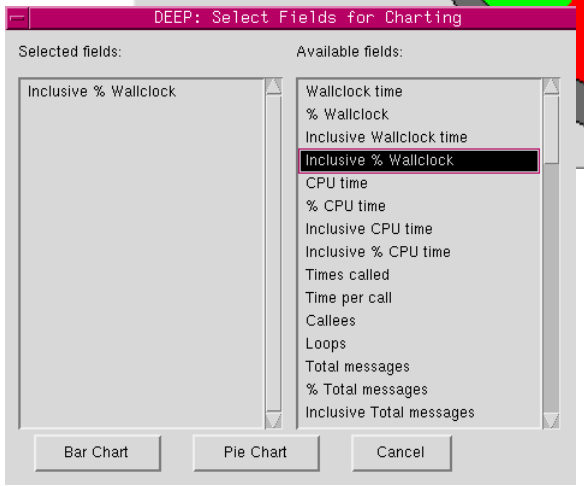
- CPU Message balance

- By default, all other displays show performance data of process 0
- If unbalanced, use process table to select other process

DEEP/MPI: Global Summary Charts

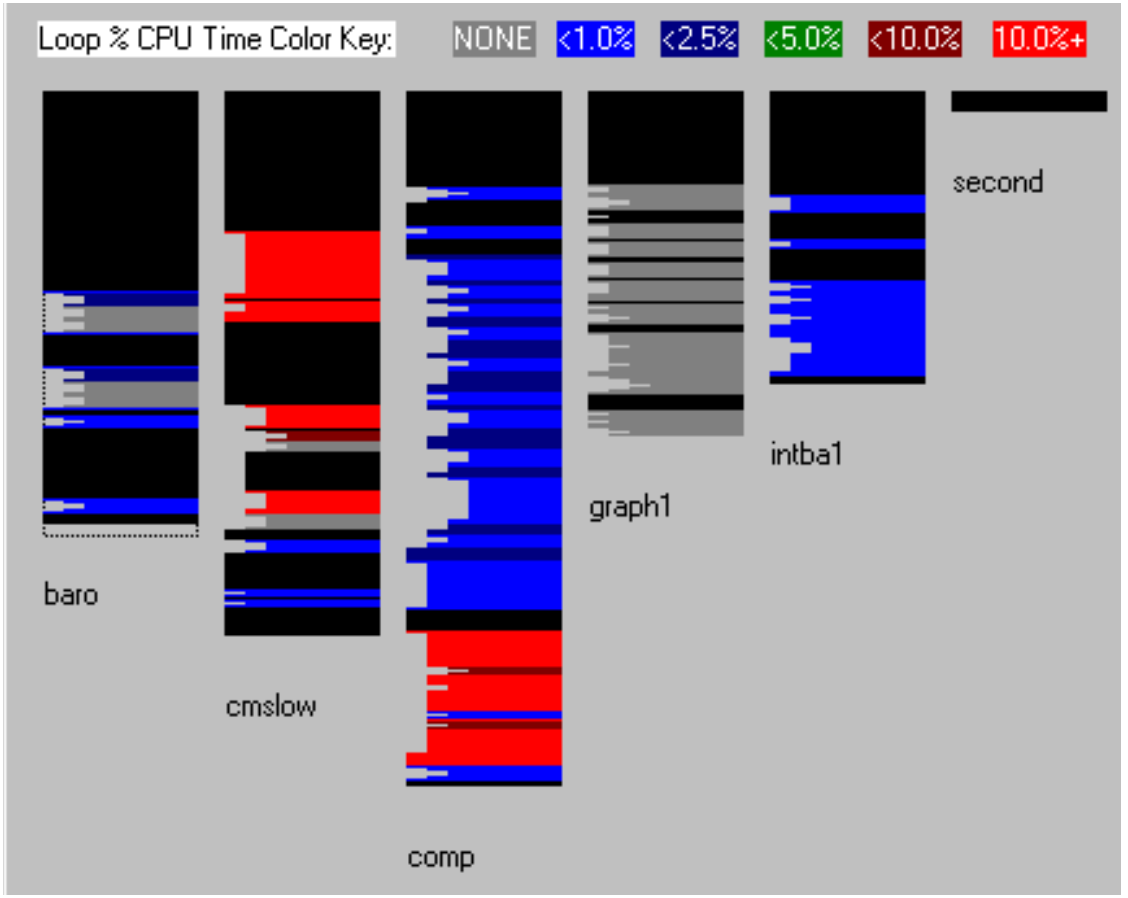
- Predefined:
  - Message counts by procedure
  - Wallclock time by procedure
  - Number of MPI calls by procedure
  - Wallclock time of MPI calls by procedure
- or:
  Custom Charts





DEEP/MPI: Global Loop Time

- Whole program performance overview
- Each procedure represented by rectangle
  - one pixel horizontal line per code line
  - indent shows code structure
- Color code shows CPU time per loop



DEEP/MPI: Global Performance Tables

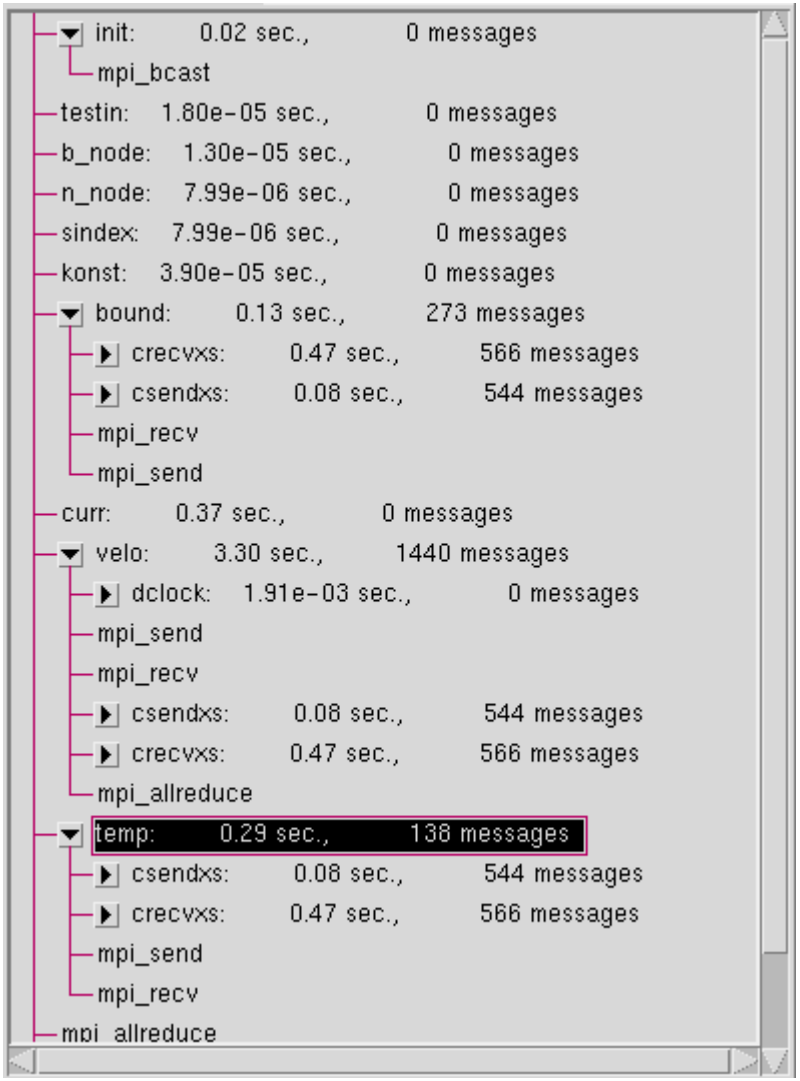
Compile-time | MPI Call Sites | MPI Summary | Loops | Inclusive Run-time | Run-time

Procedure Name	Wallclock time	% Wallclock	CPU time	% CPU time	Times called	Time per call	Callees	Loops
cx3d	5.02	54.92	0.00	0.00	1	5.02	17	
velo	2.85	31.19	2.87	70.69	20	0.14	32	
crecvxs	0.47	5.12	0.45	11.08	566	8.27e-04	0	
curr	0.37	4.08	0.38	9.36	20	0.02	0	
temp	0.25	2.76	0.24	5.91	20	0.01	10	
csendxs	0.08	0.90	0.08	1.97	544	1.51e-04	0	
bound	0.04	0.43	0.03	0.74	21	1.86e-03	24	
init	0.02	0.23	0.00	0.00	1	0.02	0	
maxv	5.30e-03	0.06	0.00	0.00	1	5.30e-03	0	
dclock	1.39e-03	0.02	0.00	0.00	101	1.38e-05	0	

- Compile time, exclusive and inclusive run time, MPI Summary, MPI call sites, and loop statistics for each procedure in the program
- Tables can be sorted on any field
- Table entries can be pruned or highlighted

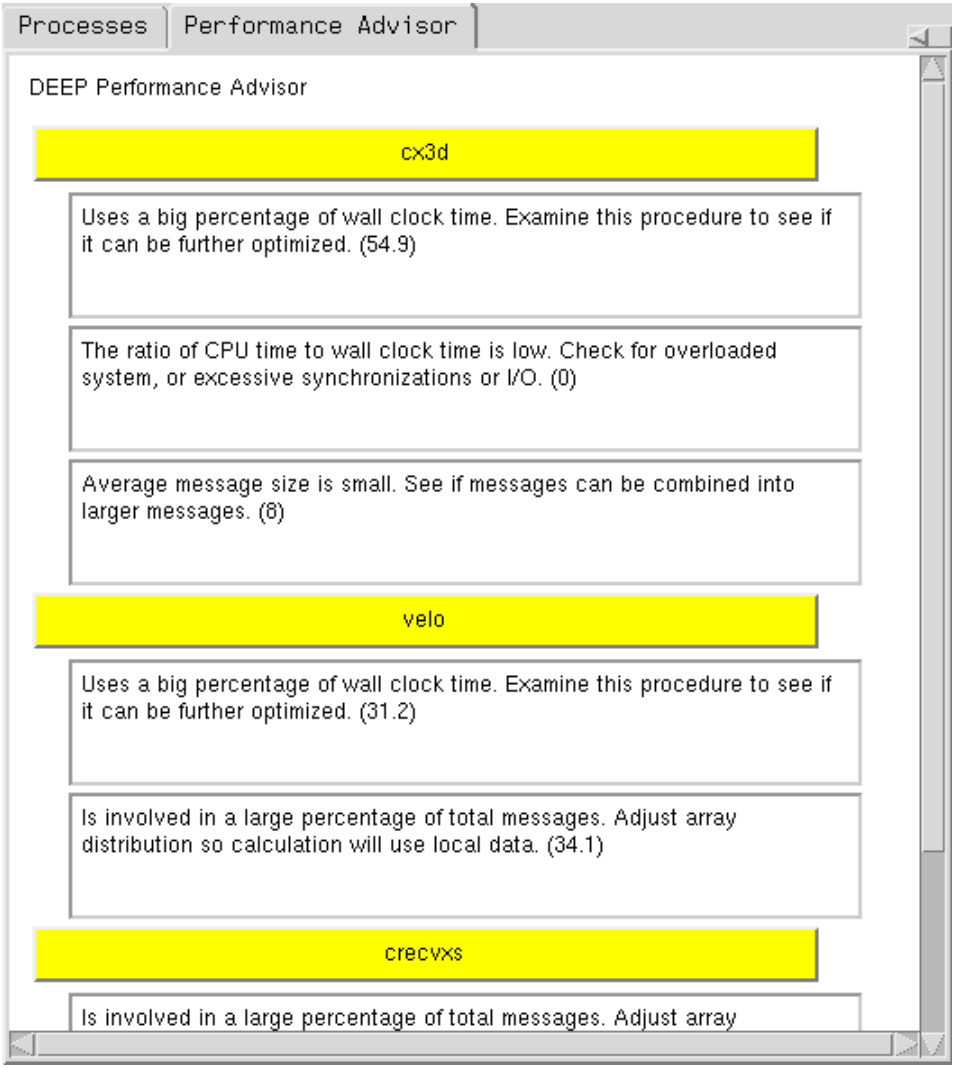
DEEP/MPI: Global Call Tree

- Features
  - any function can be root
  - can show calling tree
  - can show called tree
- Annotated with inclusive
  - wallclock time
  - number of messages
- Can expand / contract tree branches



DEEP/MPI: Performance Advisor

- Point out potential performance problems to the user
- Functions sorted by CPU time usage
- Click on function (yellow) or loop (red) buttons to bring up corresponding information



DEEP/MPI: Procedure Performance

Procedure Performance Displays

- External Call Performance table
- MPI Call Performance table
- Loop performance table

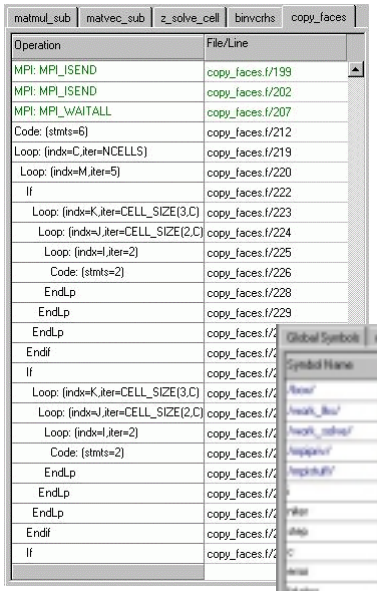
cx3d

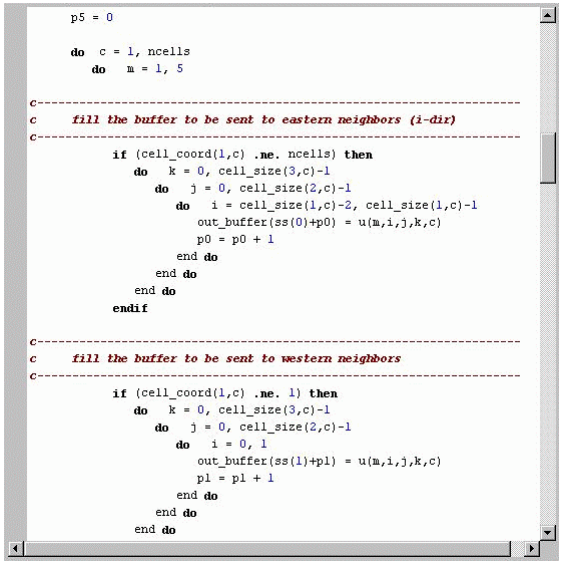
Called Routine	CPU Time	% CPU	Times Called	Calls per Parent	CPU
dclock	0.00e+00	0.0	19	19	
dclock	0.00e+00	0.0	1	1	
init	0.00e+00	0.0	1	1	
testin	0.00e+00	0.0	1	1	
b_node	0.00e+00	0.0	1	1	
n_node	0.00e+00	0.0	1	1	
sindex	0.00e+00	0.0	1	1	
konst	0.00e+00	0.0	1	1	
bound	3.00e-02	0.7	1	1	
curr	3.80e-01	9.4	20	20	
velo	3.28e+00	80.8	20	20	
temp	3.20e-01	7.9	20	20	
bound	5.00e-02	1.2	20	20	
maxv	0.00e+00	0.0	1	1	

DEEP/MPI: Procedure Browsing

Selecting a procedure in most displays updates all source code windows

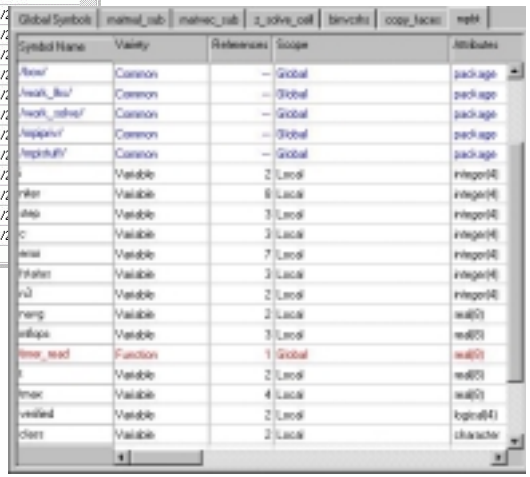
- Source Code Abstract (Basic blocks)





```
      p5 = 0

      do  c = 1, ncells
         do  m = 1, 5

c-----------------------------------------------------------------------
c     fill the buffer to be sent to eastern neighbors (i-dir)
c-----------------------------------------------------------------------
            if (cell_coord(1,c) .ne. ncells) then
               do  k = 0, cell_size(3,c)-1
                  do  j = 0, cell_size(2,c)-1
                     do  i = cell_size(1,c)-2, cell_size(1,c)-1
                        out_buffer(ss(0)+p0) = u(m,i,j,k,c)
                        p0 = p0 + 1
                     end do
                  end do
               end do
            endif

c-----------------------------------------------------------------------
c     fill the buffer to be sent to western neighbors
c-----------------------------------------------------------------------
            if (cell_coord(1,c) .ne. 1) then
               do  k = 0, cell_size(3,c)-1
                  do  j = 0, cell_size(2,c)-1
                     do  i = 0, 1
                        out_buffer(ss(1)+p1) = u(m,i,j,k,c)
                        p1 = p1 + 1
                     end do
                  end do
               end do
```

- Source Code Browser
- Symbol reference browser



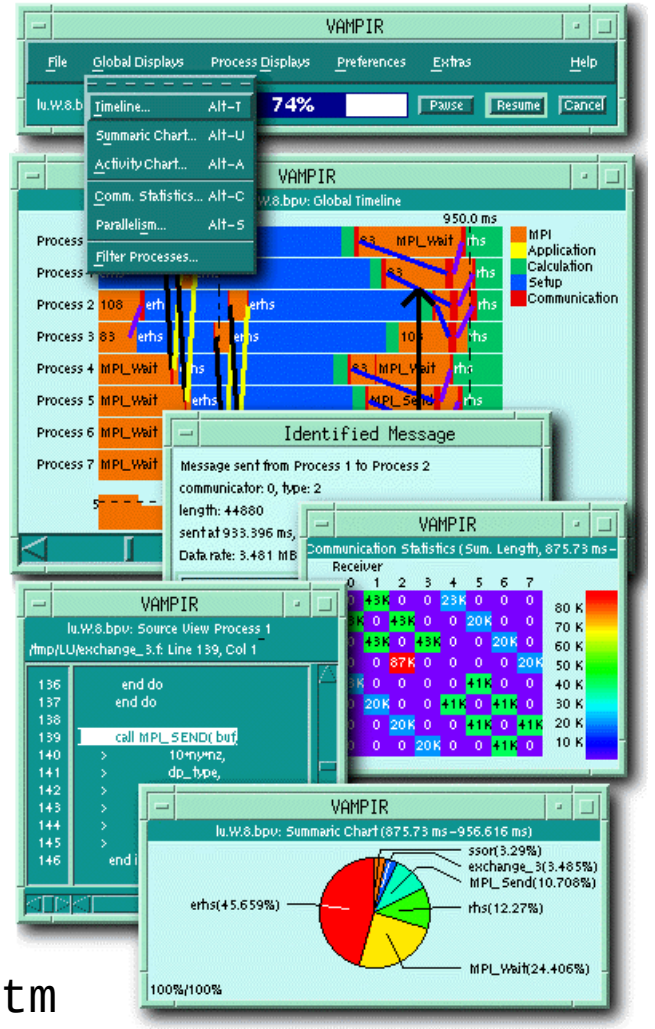
- Symbol Table Browser
  - selecting symbol highlights all appearances in code browser

DEEP: Availability

- DEEP/MPI
  - Linux x86 and Alpha
  - SGI Irix
  - Sun SPARC Solaris
  - IBM AIX
  - Windows NT (Program analysis only)
- DEEP/PAPI
  - Linux x86
- Other platforms on request

Vampir

- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Now main development by Technical University Dresden
- Commercially distributed by PALLAS, Germany



- http://www.pallas.de/pages/vampir.htm

Vampir: General Description

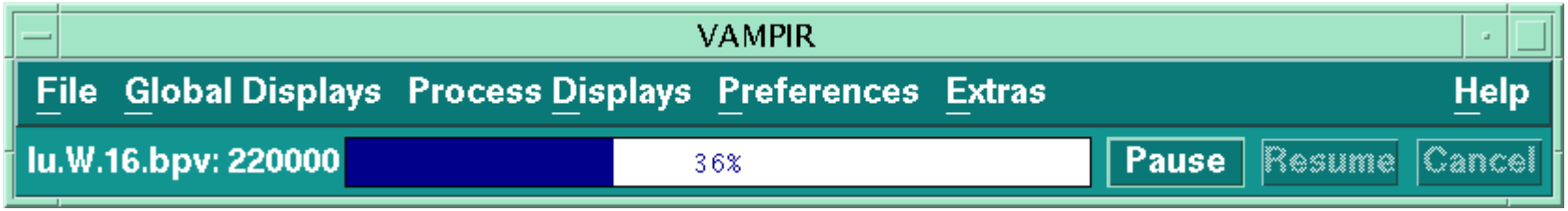
- Offline trace analysis for message passing trace files

- Convenient user–interface / Easy customization
- Scalability in time and processor–space
- Excellent zooming and filtering

- Display and analysis of MPI and application events:
  - user subroutines
  - point–to–point communication
  - collective communication (Version 2.5)
  - MPI–2 I/O operations (Version 2.5)

- All diagrams highly customizable (through context menus)
- Large variety of displays for ANY part of the trace

Vampir: Main window



- Trace file loading can be
  - interrupted at any time
  - resumed
  - started at a specified time offset
- Provides main menu
  - access to global and process local displays
  - preferences
  - help
- Trace file can be re–written (re–grouped symbols)

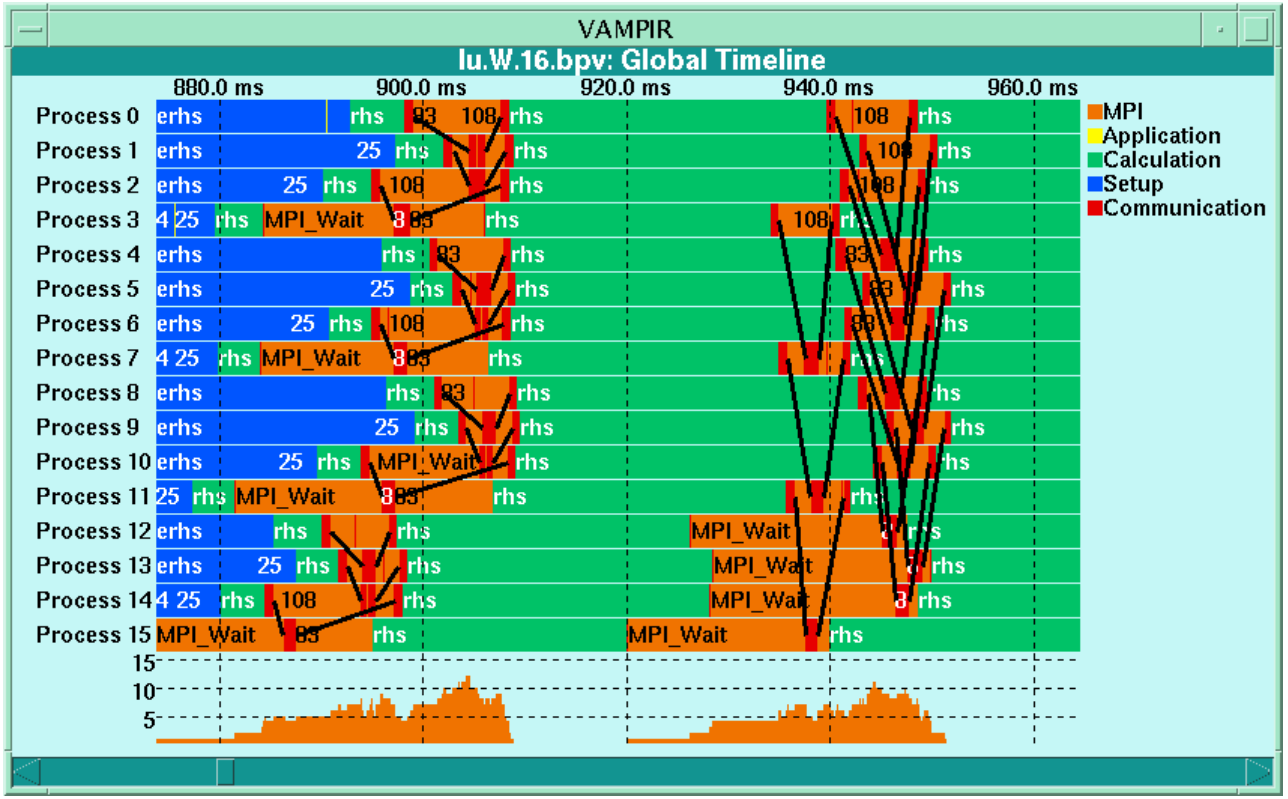
Vampir: Displays

- Global displays show all selected processes
  - Summary Chart: aggregated profiling information
  - Activity Chart: presents per–process profiling information
  - Timeline: detailed application execution over time axis
  - Communication statistics: message statistics for each process pair
  - Global Comm. Statistics: collective operations statistics
  - I/O Statistics: MPI I/O operation statistics
  - Calling Tree: global dynamic calling tree

- Process displays show a single process per window for selected processes
  - Activity Chart
  - Timeline
  - Calling Tree

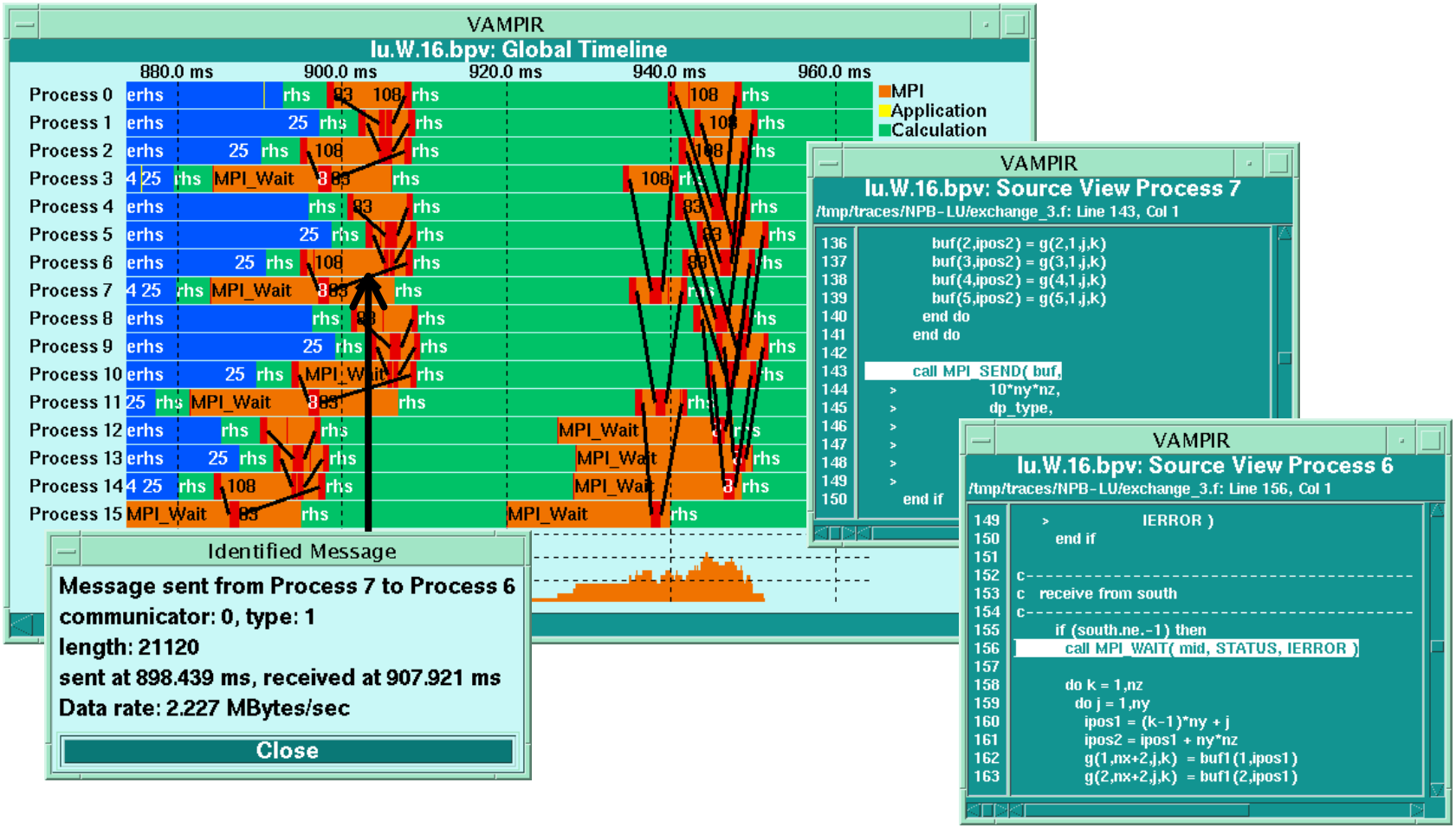
Vampir: Time Line Diagram

- Functions organized into groups
- coloring by group
- Message lines can be colored by tag or size



- Information about states, messages, collective and I/O operations available through clicking on the representation

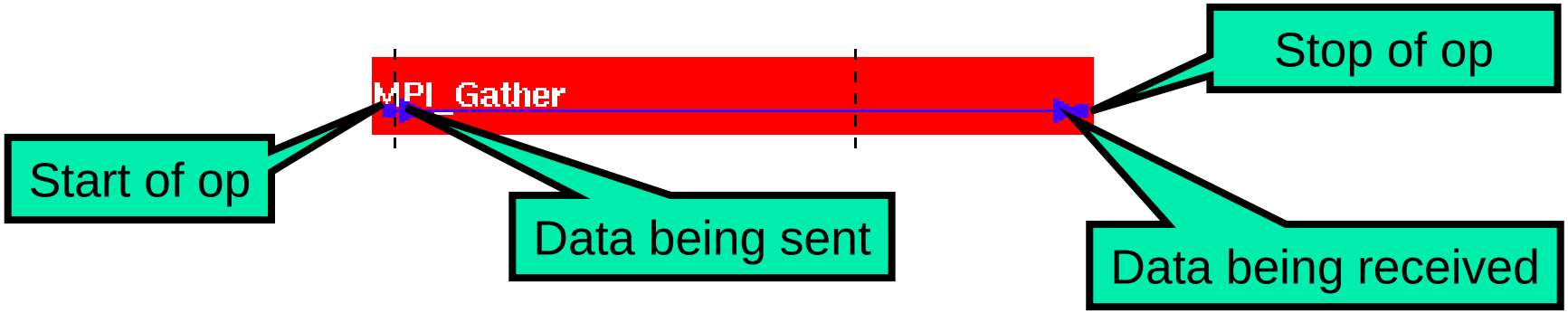
Vampir: Time Line Diagram (Message Info)



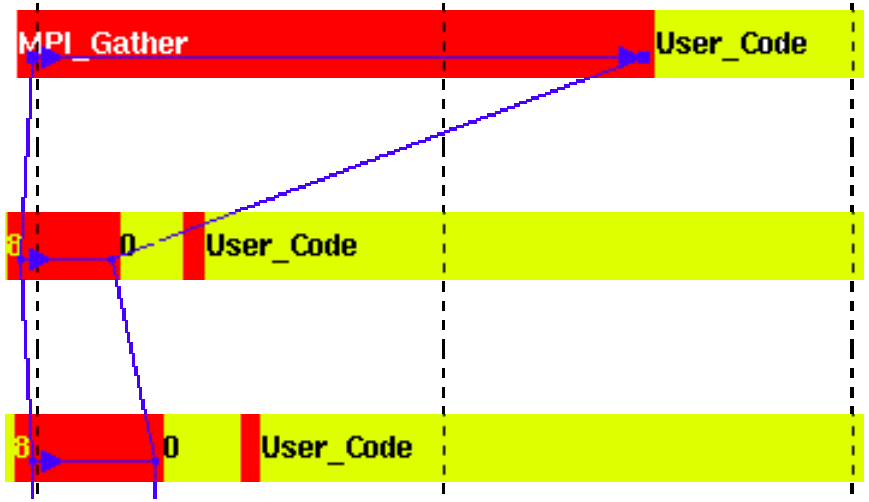
- Source–code references are be displayed if recorded in trace

Vampir: Support for Collective Communication

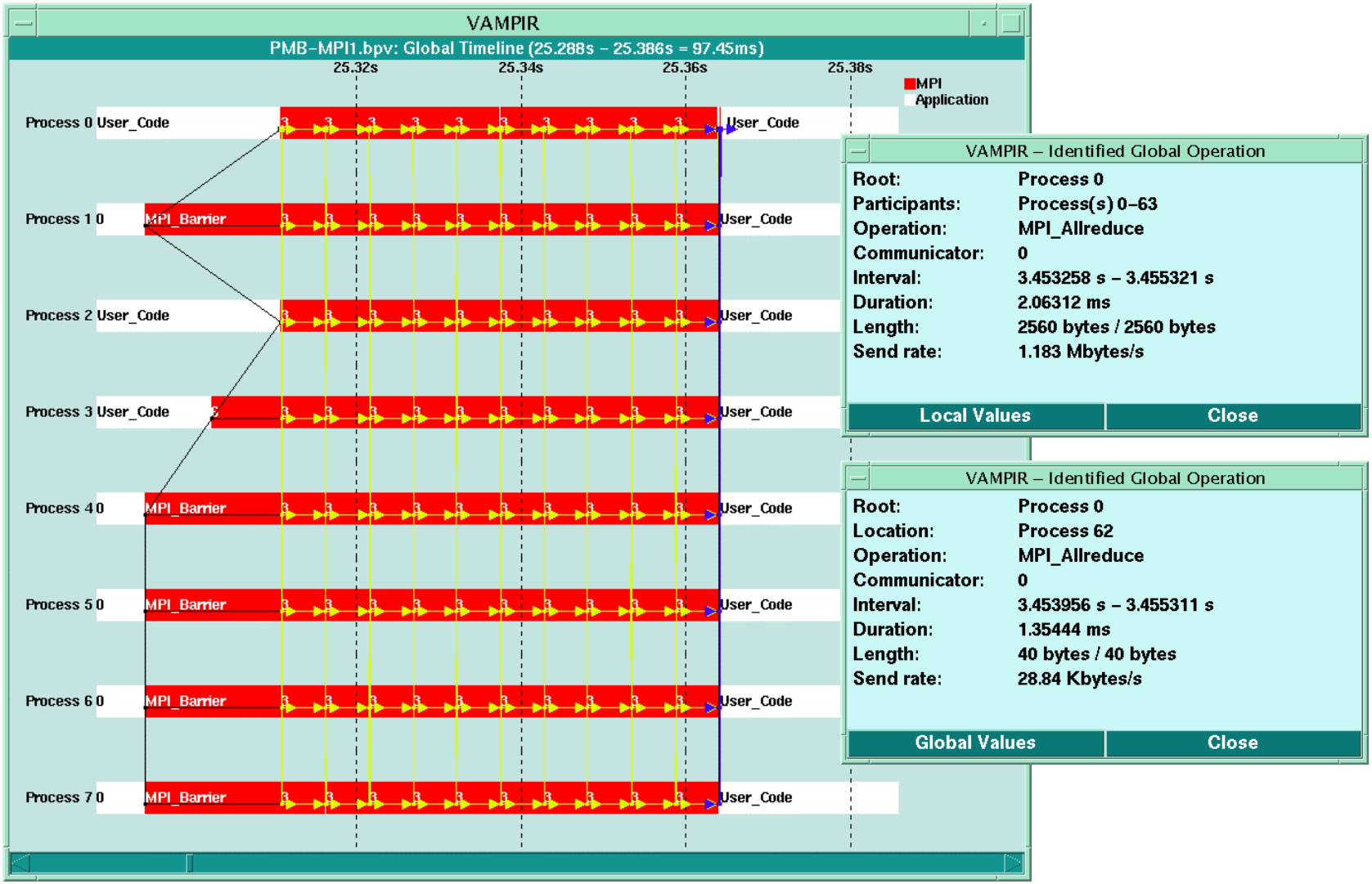
- For each process: locally mark operation



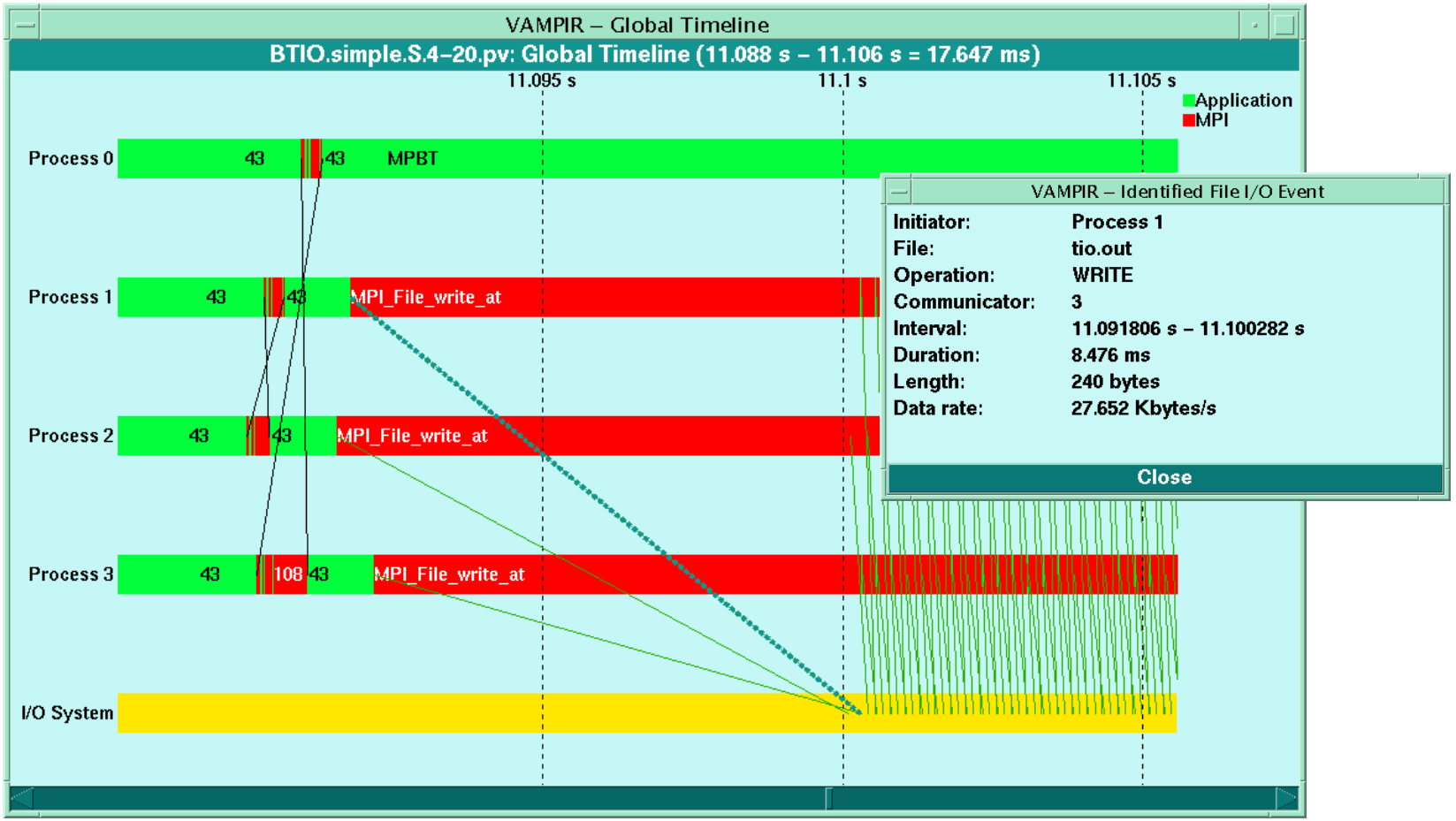
- Connect start/stop points by lines



Vampir: Collective Communication

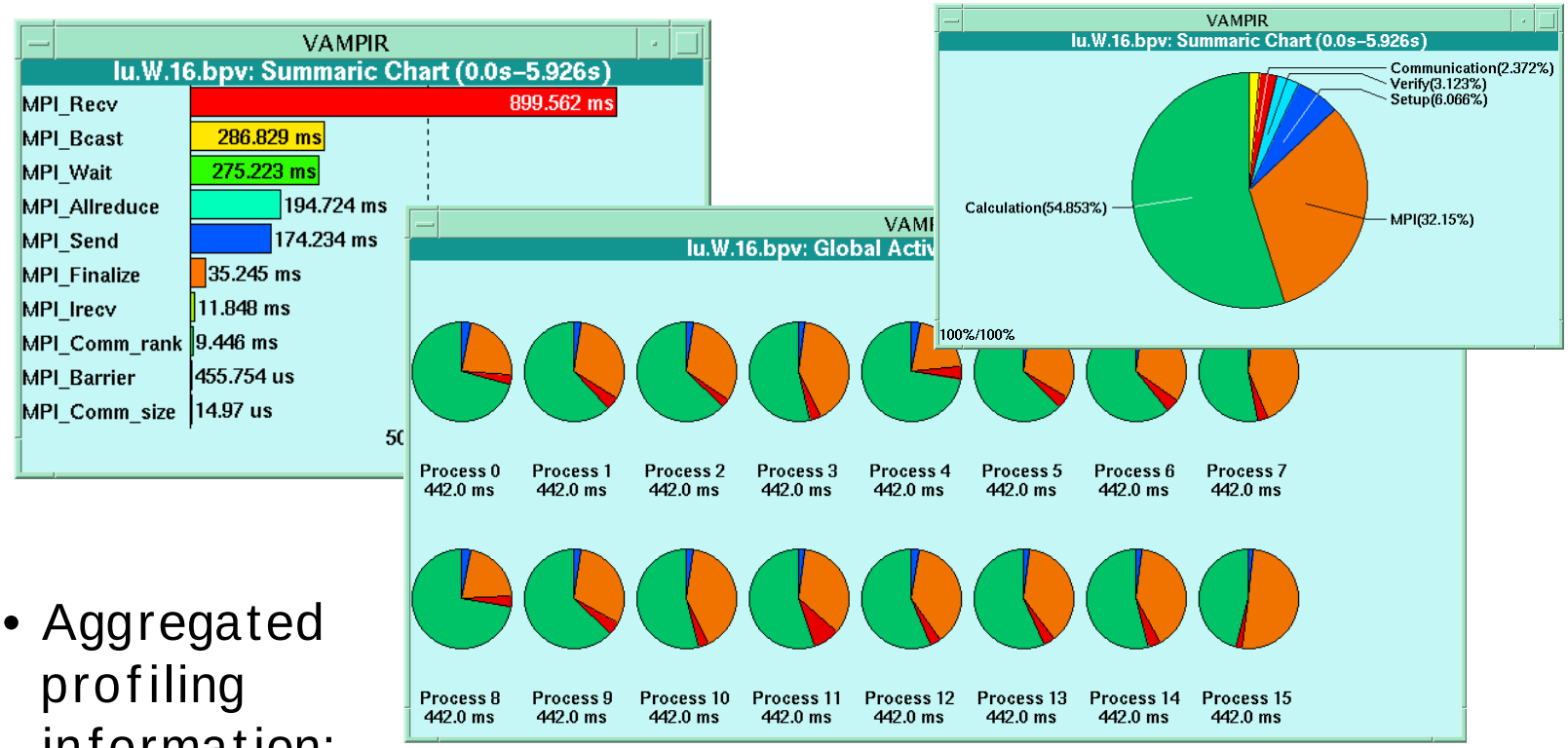


Vampir: MPI-I/O Support



- MPI I/O operations shown as message lines to separarte I/O system time line

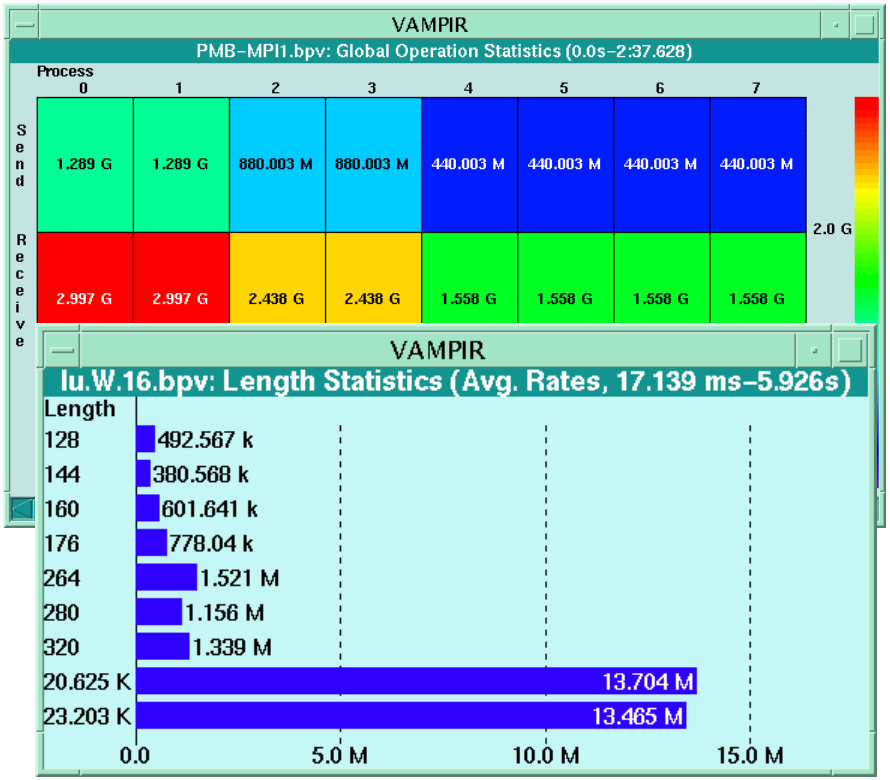
Vampir: Execution Statistics



- Aggregated profiling information: execution time, number of calls, inclusive/exclusive
- Available for all / any group (activity) or all routines (symbols)
- Available for any part of the trace (selectable through time line diagram)

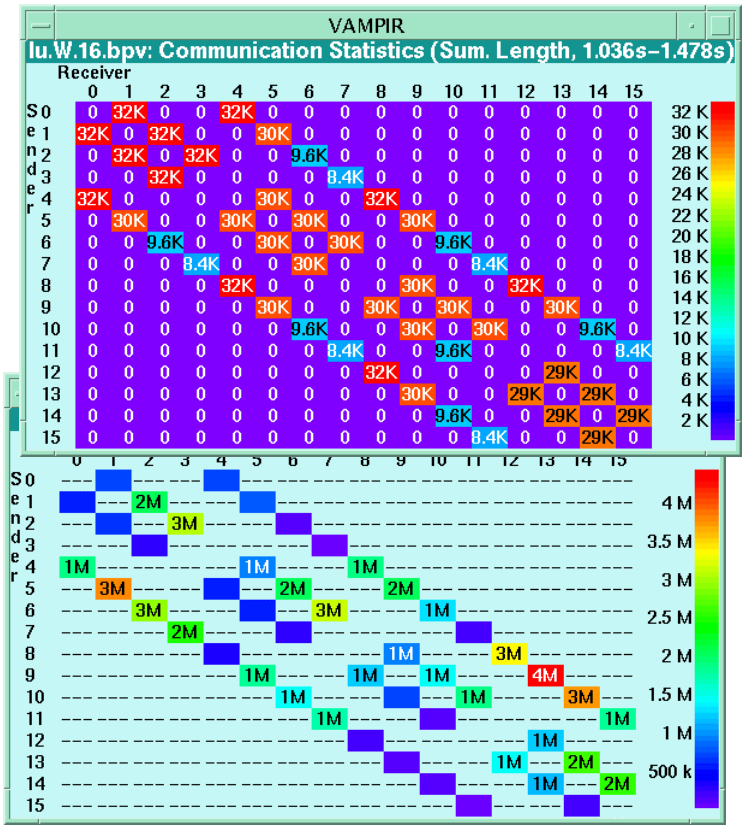
Vampir: Communication Statistics

- Bytes sent/received for collective operations

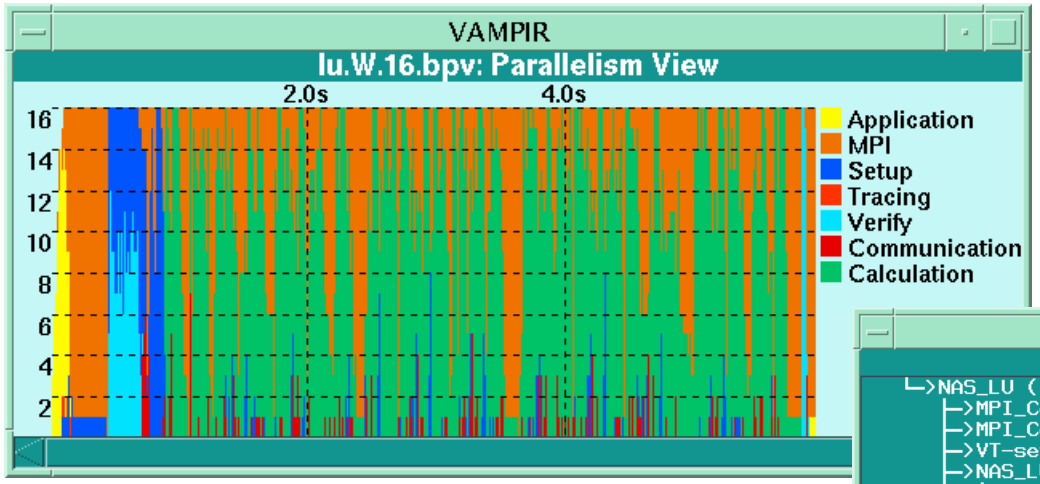


- Message length statistics
- Available for any part of the trace

- byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair



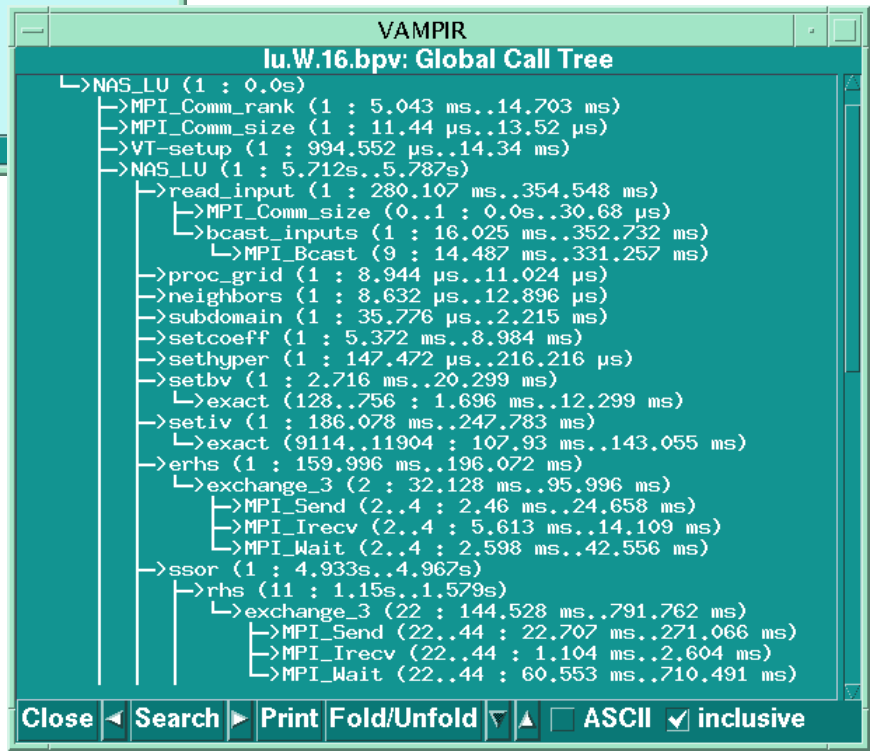
Vampir: Other Features



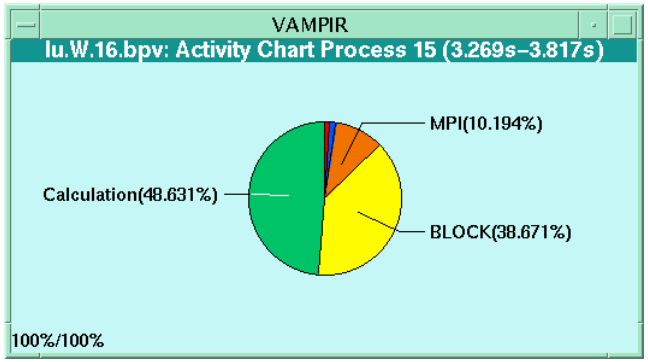
- parallelism display

- Powerful filtering and trace comparison features
- All diagrams highly customizable (through context menus)

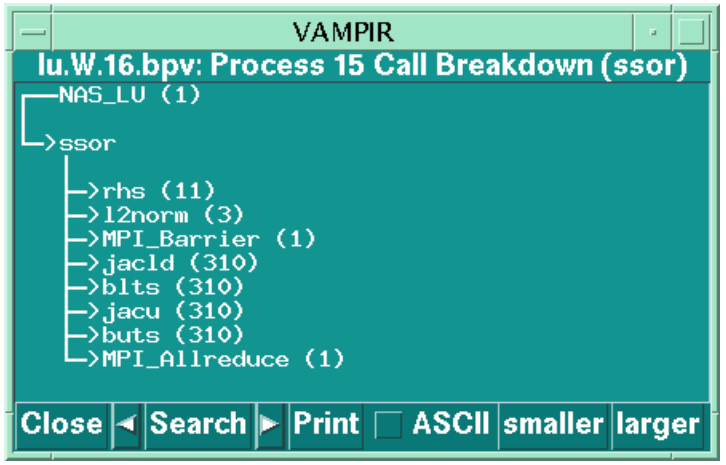
•Dynamic global call graph tree



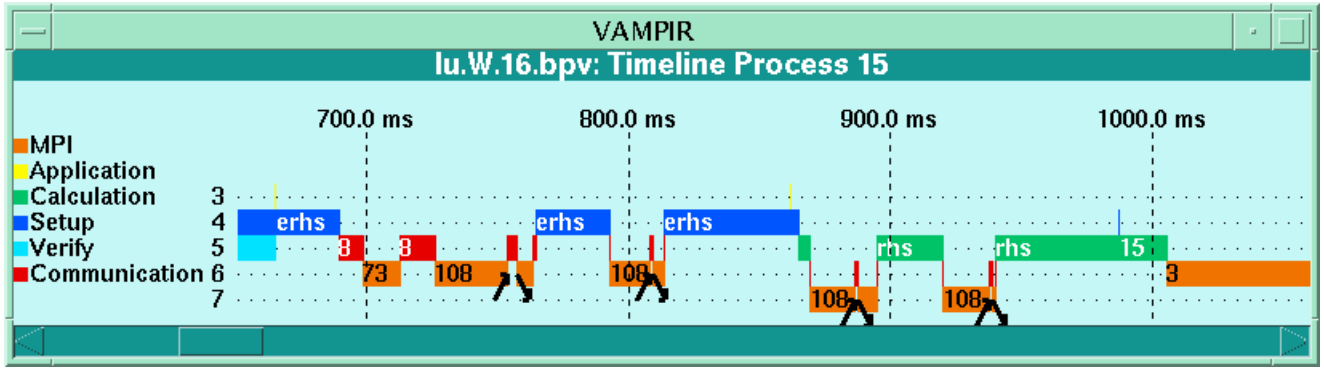
Vampir: Process Displays



•Activity chart



•Call tree



•Time line

•For all selected processes in the global displays

Vampir: Availability

- Supported platforms
  - Compaq Alpha, Tru64 Unix
  - Cray T3E
  - Hitachi SR2201, HI-UX/MPP
  - HP PA, HP-UX 10
  - HP Exemplar, SPP-UX 5.2
  - IBM RS/6000 and SP-2, AIX 4
  - Linux/Intel 2.0
  - NEC EWS, EWS-UX 4
  - NEC SX-4, S-UX 8
  - SGI workstations, Origin and PowerChallenge, IRIX 6
  - Sun Solaris/SPARC 2.7
  - Sun Solaris/Intel 2.6

Vampirtrace

- Commercial product of Pallas, Germany

- Library for Tracing of MPI and Application Events
  - records point-to-point communication
  - records user subroutines (on request)
- New version 2.0
  - records collective communication
  - records MPI–2 I/O operations
  - records source–code information (some platforms)
  - support for shmem (Cray T3E)
- uses the MPI profiling interface
- http://www.pallas.de/pages/vampirt.htm

Vampirtrace: Usage

- Record MPI–related information
  - Re–link a compiled MPI application (no re-compilation necessary!)

```
% f90 *.o -o myprog -L$(VTHOME)/lib -lVT -lpmpi -lmpi
% cc  *.o -o myprog -L$(VTHOME)/lib -lVT -lpmpi -lmpi
% CC  *.o -o myprog -L$(VTHOME)/lib -lVT -lpmpi -lmpi
```

  - [MPICH only] Re-link using `-vt` option to MPI compiler scripts

```
% mpif90 -vt *.o -o myprog
% mpicc  -vt *.o -o myprog
% mpiCC  -vt *.o -o myprog
```

  - Execute MPI binary as usual

- Record user subroutines
  - insert calls to the Vampirtrace API (portable, but inconvenient)
  - use automatic instrumentation (NEC SX, Fujitsu VPP, Hitachi SR)
  - use instrumentation tool (Cray PAT, dyninst, ...)

Vampirtrace Instrumentation API (C/C++)

- Calls for recording user subroutines

```
#include "VT.h"
/* -- Symbols can be defined with or without source information -- */
VT_symdefl(123, "foo", "USER", "foo.c:6");
VT_symdef (123, "foo", "USER");

void foo {
    VT_begin(123);              /* 1st executable line */
    ...
    VT_end(123);                /* at EVERY exit point! */
}
```

- VT calls can only be used between `MPI_Init` and `MPI_Finalize`!
- Event numbers used must be globally unique
- Selective tracing through
  - `VT_traceoff();` `# Turn tracing off`
  - `VT_traceon();` `# Turn tracing on`

VT++.h - C++ class wrapper for Vampirtrace

```
#ifndef __VT_PLUSPLUS_
#define __VT_PLUSPLUS_
#include "VT.h"
class VT_Trace {
  public:  VT_Trace(int code) {VT_begin(code_ = code);}
          ~VT_Trace()         {VT_end(code_);}
  private: int code_;
};
#endif /* __VT_PLUSPLUS_ */
```

- Same tricks can be used to wrap other tracing APIs for C++ too
- Usage:

```
VT_symdef(123, "foo", "USER"); // symbol definition as before
void foo(void) {               // user subroutine to monitor
  VT_Trace vt(123);            // declare VT_Trace object in 1st line
  ...                          // => automatic tracing by ctor/dtor
}
```

Vampirtrace Instrumentation API (Fortran)

- Calls for recording user subroutines

```
      include 'VT.inc'
      integer ierr
      call VTSYMDEF(123, "foo", "USER", ierr)    !or
      call VTSYMDEFL(123, "foo", "USER", "foo.f:8", ierr)
C
      SUBROUTINE foo(...)
      include 'VT.inc'
      integer ierr
      call VTBEGIN(123, ierr)
C     ...
      call VTEND(123, ierr);
      END
```

- Selective tracing through
  - VTTRACEOFF() # Turn tracing off
  - VTTRACEON() # Turn tracing on

Vampirtrace: Runtime Configuration

- Trace file collection and generation can be controlled by using a configuration file
  - trace file name, location, size
  - flush behavior
  - activation/deactivation of trace recording for specific processes, activities (groups of symbols), and symbols
- Activate a configuration file by setting environment variables

`VT_CONFIG`	name of configuration file (use absolute pathname if possible)
`VT_CONFIG_RANK`	MPI rank of process which should read and process configuration file (default: 0)

- Reduce trace file sizes
  - restrict event collection in a configuration file
  - use selective tracing functions (`VT_traceoff/VT_traceon()`)

Vampirtrace: Configuration File Example

```
# collect traces only for MPI ranks 1 to 5
TRACERANKS 1:5:1
# record at most 20000 records per rank
MAX-RECORDS 20000

# do not collect administrative MPI calls
SYMBOL MPI_comm*  off
SYMBOL MPI_cart*  off
SYMBOL MPI_group* off
SYMBOL MPI_type*  off

# do not collect USER events
ACTIVITY USER off
# except routine foo
SYMBOL foo on
```

- Be careful to record complete message transfers!
- For more complete description see Vampirtrace User's Guide

Vampirtrace: Availability

- Supported platforms
  - Compaq Alpha, Tru64 Unix
  - CRAY T3E Unicos/mk
  - Fujitsu VPP300, VPP700, and VPP5000 UXP/V
  - Hitachi SR8000, HI-UX/MPP
  - HP PA, HP-UX 10 + 11, SPP5
  - IBM RS/6000 and SP-2, AIX 4.x
  - Linux/Intel 2.x
  - NEC SX-4, S-UX 8
  - SGI workstations, Origin, and PowerChallenge, IRIX 6
  - Sun Solaris/SPARC 2.7
  - Sun Solaris/Intel 2.6

DIMEMAS

- Developed by CEPBA-UPC Spain

CEPBA
EUROPEAN CENTER FOR PARALLELISM OF BARCELONA

- Performance prediction tool for message-passing programs
  - supports several message-passing libraries including PVM, MPI and PARMACS
- Uses
  - Prediction of application runtime on a specific target machine
  - Study of load imbalances and potential parallelism
  - Sensitivity analysis for architectural parameters
  - Identification of user modules that are worthwhile to optimize
- Limitations
  - cache effects
  - process scaling
  - non–deterministic programs
- http://www.cepba.upc.es/tools/dimemas/

Dimemas Model

- Trace file contains
  - CPU time spent in each code block
  - parameters of communication events (point–to–point, collective)
- Dimemas simulates application execution
  - scales time spent in each block according to target CPU speed
  - performs communication according to target machine model
- Dimemas can model
  - multitasking / multithreading
  - arbitrary process–to–processor mappings
  - heterogeneous machines with various interconnections
    - includes database of machine parameters
  - communication: point–to–point according to latency/bandwidth model, network contention, collective communication (constant, linear, logarithmic)

Dimemas Outputs

- Simulator
  - Execution time + Speed–up
  - Per task:
    - Execution, CPU, Blocking time
    - Messages sent + received
    - Data volume communicated
- What–If Analysis
  - Analysis per code block or subroutine
  - Analysis for target machine parameters
- Critical Path Analysis
  - Critical path (longest communication path)
  - Analysis per code block: percentage of computing + communication time
- Synthetic Perturbation Analysis
- Vampir and Paraver Trace files

Dimemas: Application Analysis

- Analyze load–balance and communication dependencies:
  - set bandwidth=∞, latency=0
- Should messages be aggregated?
  - set bandwidth= ∞
- Does the bandwidth constrain performance?
  - set latency=0
- Does message contention constrain performance?
  - set number of connections=1, 2, ...

Dimemas: Availability

- Supported platforms
  - SGI workstations, Origin, and PowerChallenge, IRIX 6

GuideView

- Commercial product of KAI
- OpenMP Performance Analysis Tool
- Part of KAP/Pro Toolset for OpenMP
- Looks for OpenMP performance problems
  - load imbalance
  - synchronisation (waiting for locks)
  - false sharing
- Works from execution trace(s)
- Usage:
  - compile with Guide compiler and link with instrumented library

```
% guidec++ -Wgstats myprog.cpp -o myprog
% guidef90 -Wgstats myprog.f90 -o myprog
```

  - run with real input data sets
  - view traces with `guideview`

- `http://www.kai.com/parallel/kappro/`

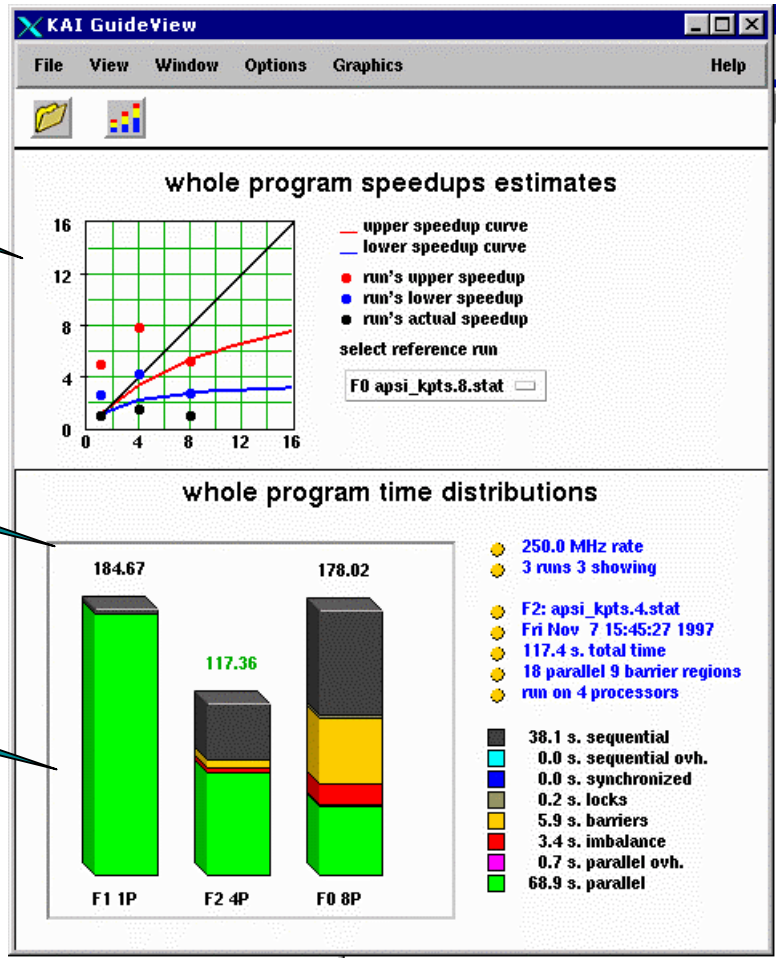
GuideView: Whole Application View

Compare actual vs. ideal performance

Identify bottlenecks (barriers, locks, seq. time)

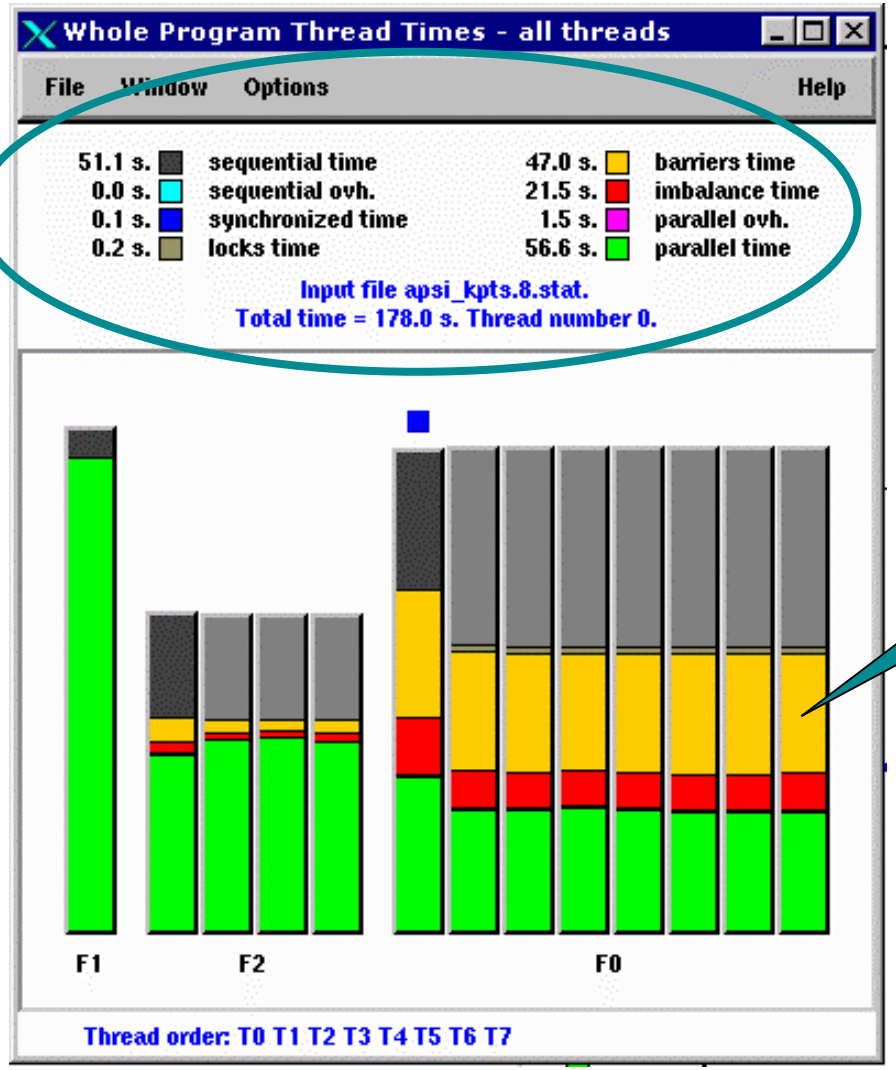
Compare multiple runs

- Different number of processors
- different datasets
- different platforms



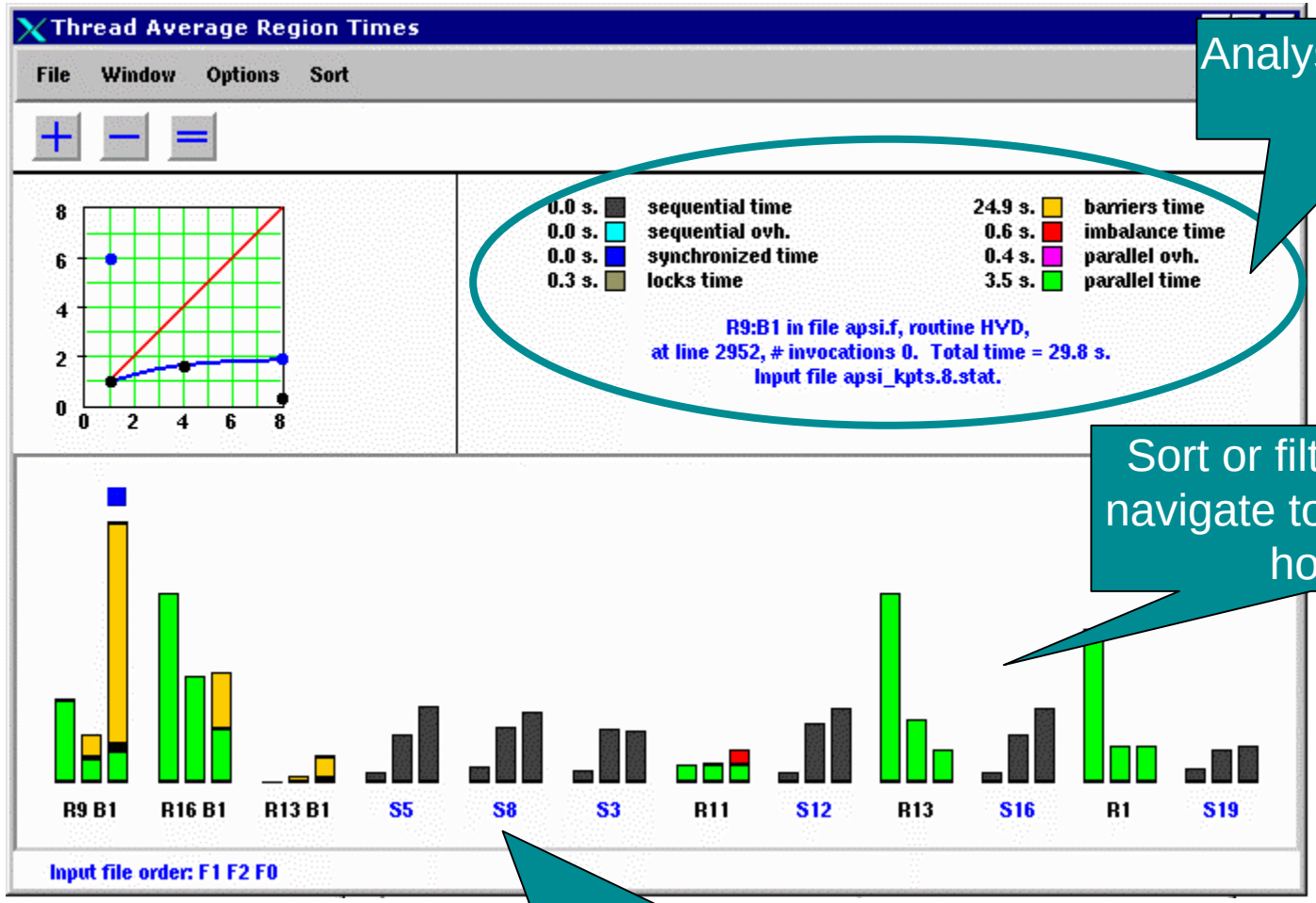
GuideView: Per–Thread View

Analyse each thread's performance



Show scalability problems

GuideView: Per–Section View



Analyse each parallel region

Sort or filter regions to navigate to performance hotspots

Identify serial regions that hurt scalability

GuideView: Advanced Usage

- Named parallel regions
  - Use utility function `kmp_set_parallel_name("some name")` to specify name for all following parallel regions.
  - Passing an empty string disables naming
  - Allows readily location of the region(s) in GuideView

- Analysis of hybrid MPI / OpenMP applications
  - Generate different Guide execution traces for each node
    - run application with node-local file system as current directory
    - set trace file name using environment variable `KMP_STATSFILE`
      - point to file in node-local file system

        ```
        % export KMP_STATSFILE=/node-local/guide.gvs
        ```

      - use special meta-character sequences
        (%H: hostname, %I: pid, %P: number of threads used)

        ```
        % export KMP_STATSFILE=guide-%H.gvs
        ```

  - Use "compare-multiple-run" feature to display them together

KAP/Pro (GuideView): Availability

- Supported platforms
  - Compaq Alpha Systems
    - Compaq Tru64 Unix 4.0 (C/C++, f77, f90)
  - SGI MIPS Systems
    - Irix 6.5 (C/C++, f77, f90)
  - Sun SPARC Systems
    - Solaris 2.5, 2.6, and 2.7 (C/C++, f77, f90)
  - Intel (IA32) Systems
    - Windows NT 4.0 and Windows2000 (C, f77, f90)
    - Linux 2.0 (C/C++)
    - Solaris 2.5 (f77)
  - Hewlett Packard PA-RISC 2.0 Systems
    - HP-UX 11.0 (C/C++, f77, f90)
  - IBM RS/6000 Systems
    - IBM AIX 4.1 - 4.3 (C/C++, f77, f90)

3rd Party: URLs

- TRAPPER's parallel programming environment includes performance analysis and execution visualization tools
  http://www.genias.de/projects/winpar/
- PGI's PGPROF HPF Profiler
  Post-mortem, Graphical or command-level profiling of HPF programs on MPP, SMP, and workstation cluster systems
  http://www.pgroup.com/ppro_docs/pgprof/pgprof.htm
- ETNUS Inc. TimeScan Multiprocess Event Analyzer
  http://www.etnus.com/products/timescan/
- CEPBA (European Center for Parallelism of Barcelona, Spain) PARAVER (Parallel Program Visualization and Analysis Tool) supports MPI, OpemMP, HW Counter Analysis
  http://www.cepba.upc.es/tools/paraver/
- Rational Quantify
  http://www.rational.com/products/quantify_nt/index.jsp

Research: Performance Tools

- Tracing of MPI Programs
  - Upshot, Nupshot, Jumpshot-2, Jumpshot-3 (MPICH)
  - MPICL / Paragraph
  - AIMS, NTV

- Profiling / Tracing of parallel, multithreaded programs
  - Dynaprof
  - TAU's Portable Profiling and Tracing Package

- Dynamic object code instrumentation
  - dyninst
- Automatic Performance Analysis
  - Paradyn

} Part III

MPICH Tracing

- MPICH distribution provides portable MPI tracing library (called logging by MPICH) through Multi-Processing Environment (MPE)

- MPI programs generate traces when compiled with `-mpilog` option to `mpicc`, `mpiCC`, `mpif77`, or `mpif90` command [or link with `-llmpi -lmpe -lpmpi(ch) -lmpi(ch)`]
- With option `-mpitrace` compiled programs write execution trace to standard output (useful for debugging)
- implemented the standard MPI profiling interface
  - ⇒ works with other MPI implementations too (e.g., SGI, IBM, Cray)

- use MPICH Version 1.2.X if possible
- `http://www.mcs.anl.gov/mpi/mpich/docs/mpeguide/paper.htm`

MPICH Tracing

- Supports three trace formats
  - ALOG: older, can be viewed with upshot and nupshot
  - CLOG: default, can be viewed with Jumpshot-2
  - SLOG: newer, can be viewed with Jumpshot-3

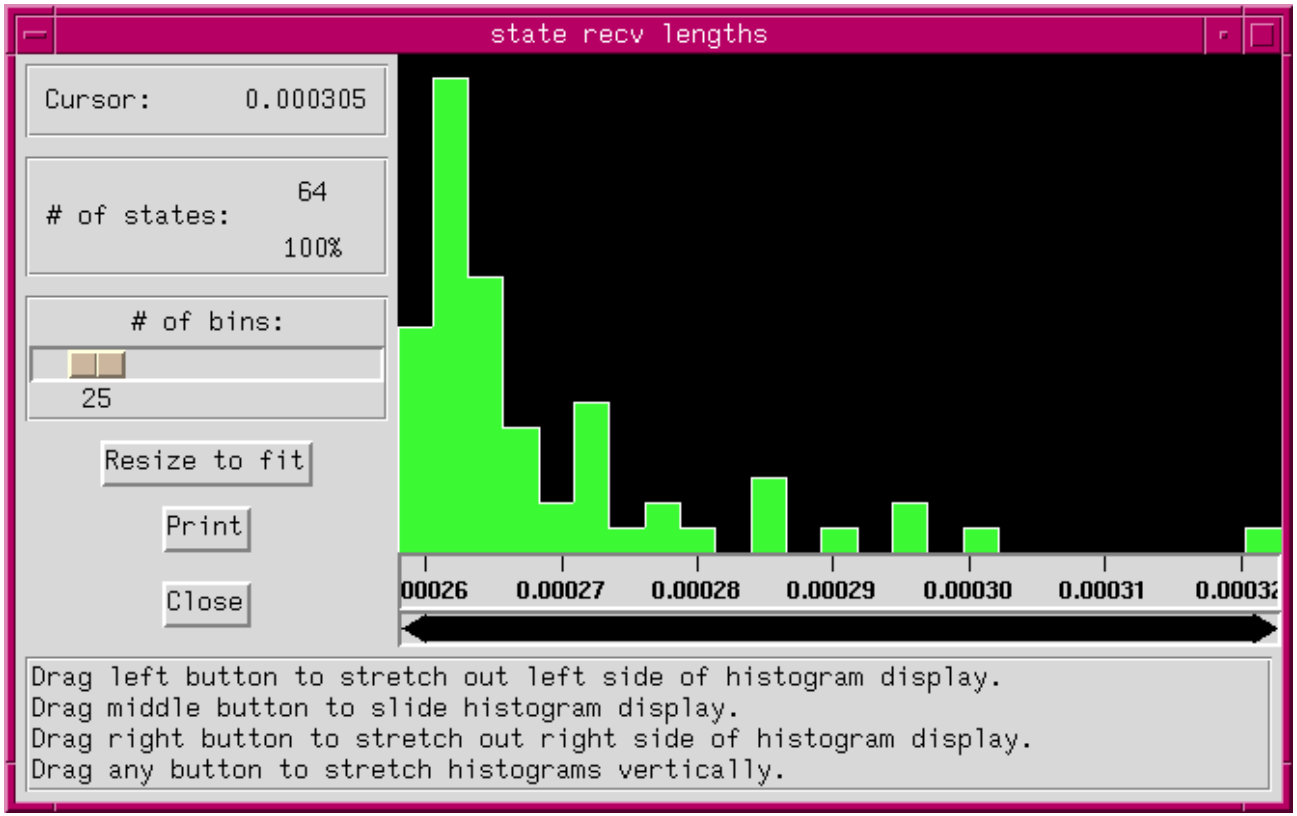
  controlled by environment variable MPE_LOG_FORMAT
- Recommendation:
  - generate traces in CLOG (default) format
  - convert to SLOG with utility tool `clog2slog` afterwards if traces get larger (5 to 10Mbyte)
- Trace analysis
  - `logviewer` *`mpich-trace-file`*
    (automatically selects correct *shot viewing program)
  - `clog_print` *`clog-file`*
  - `slog_print` *`slog-file`*

Nupshot: Timeline Display

- Static timeline display with zooming / panning
- Displays ALOG and PICL V1 formats
- Implemented with Tcl/Tk (3.6/7.3 needed)
- More portable, but slower version available as upshot

Nupshot: Statistics Display

- Statistic display showing execution time distribution of selected state
- Accessed through “state” button in timeline display

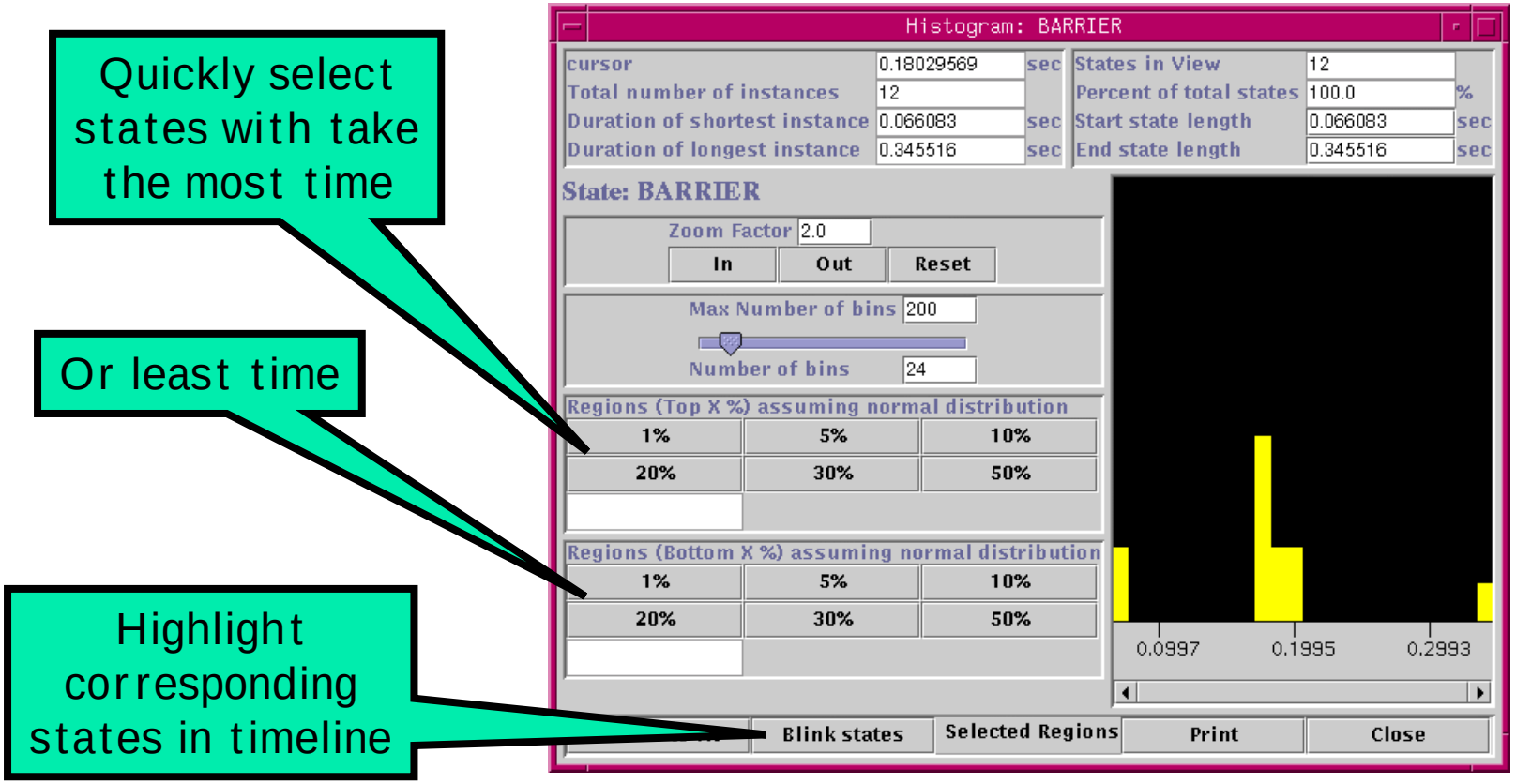


Jumpshot-2: Timeline Display

- Static timeline display with zooming / panning
- Displays CLOG and PICL V1 formats
- “Mountain-Ranges” Display also available
- Implemented in Java

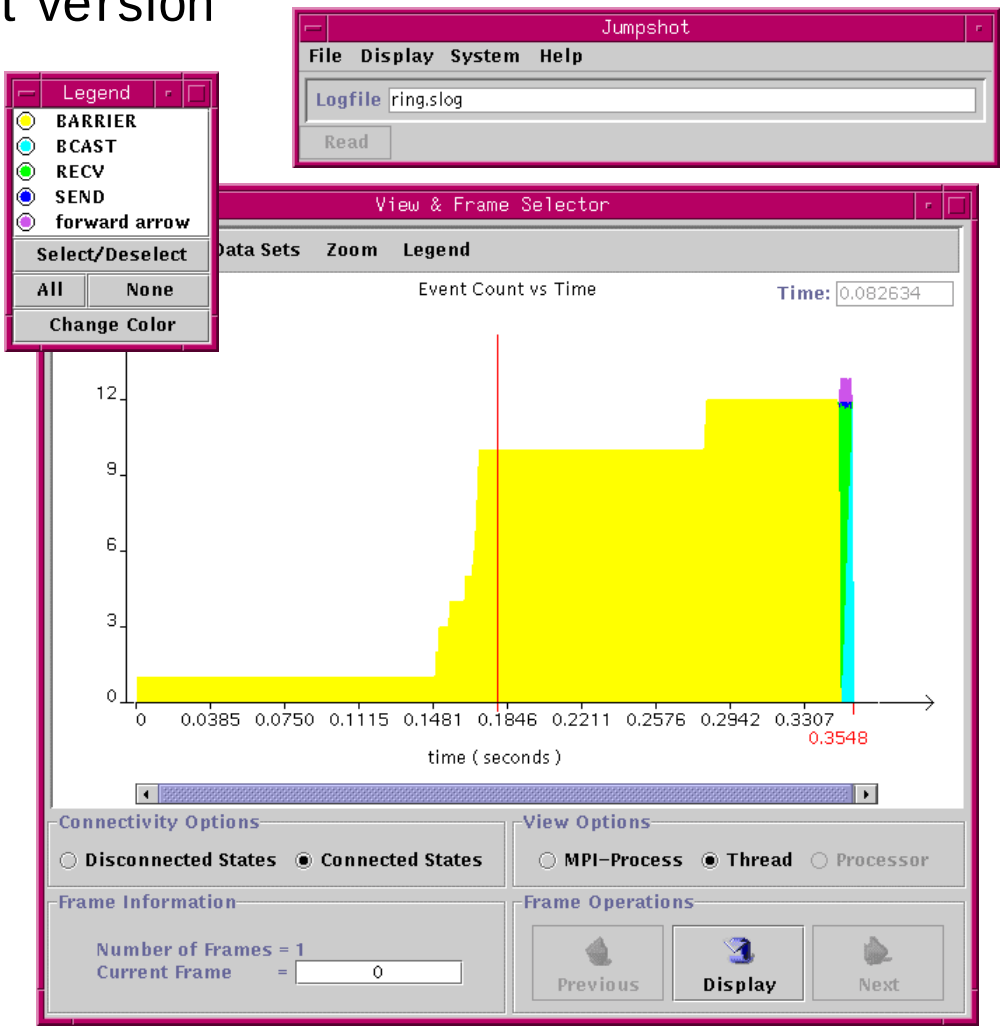
Jumpshot-2: Statistics Display

- Showing execution time distribution of selected state
- Accessed through “state” button in timeline display

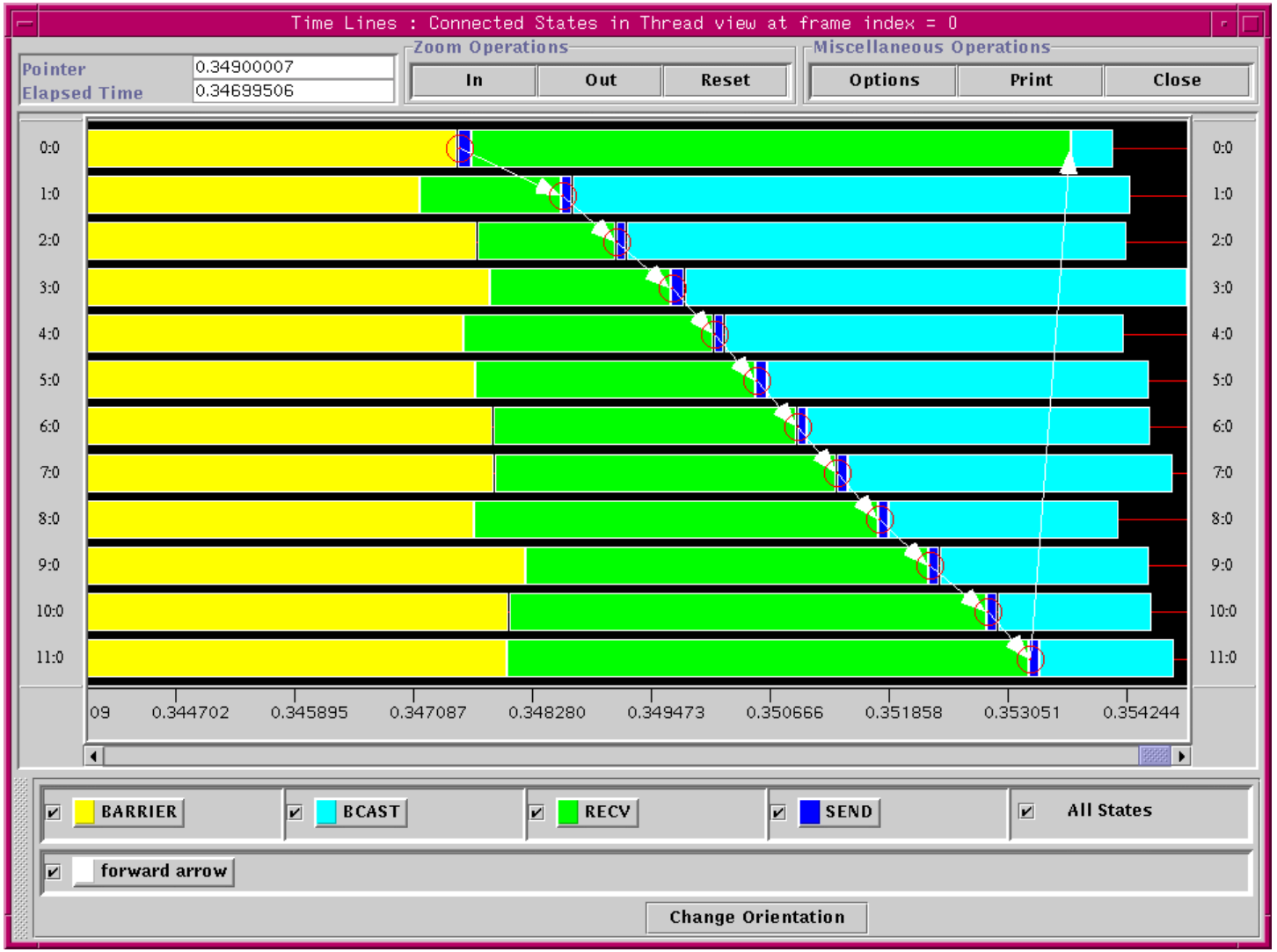


Jumpshot-3

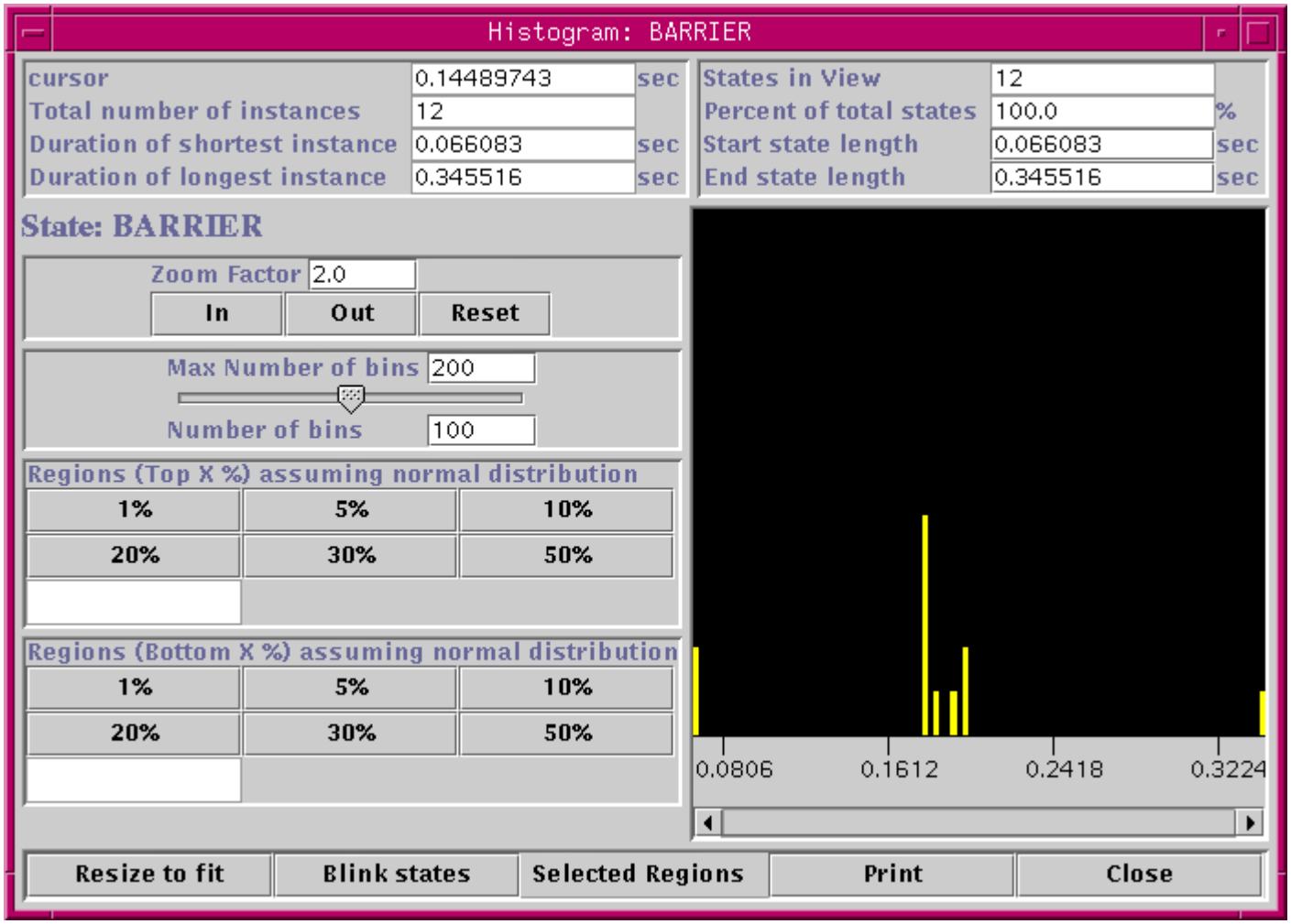
- New, enhanced Jumpshot version
- Displays SLOG format (Scalable LOG)
  - trace is segmented into frames
  - View&Frame Selector window shows rough overview
  - Timeline or Mountain-Ranges can be displayed for selected frames
- Also implemented in Java



Jumpshot-3: Timeline Display



Jumpshot-3: Statistics Display



MPE: Customizing Log Files (C/C++)

- MPE calls for recording user subroutines

```
#include "mpe.h"
int foo_start = MPE_Log_get_event_number();
int foo_end   = MPE_Log_get_event_number();
MPE_Describe_state(foo_start, foo_end, "foo", "blue");
...
void foo {
    MPE_Log_event(foo_start, 0, (char *)0);
    ...
    MPE_Log_event(foo_end, 0, (char *)0);
}
```

- MPE_Log_event can log one integer and string argument
- Selective tracing through
  - MPE_Stop_log(); or MPI_Pcontrol(0); # Turn tracing off
  - MPE_Start_log(); or MPI_Pcontrol(1); # Turn tracing on

MPE: Customizing Log Files (Fortran)

- MPE calls for recording user subroutines

```
      SUBROUTINE foo(...)
      include 'mpif.h'
      integer e1, e2, ierr
      e1 = MPE_LOG_GET_EVENT_NUMBER()
      e2 = MPE_LOG_GET_EVENT_NUMBER()
      ierr = MPE_DESCRIBE_STATE(e1, e2, "foo", "blue")
C
      ierr = MPE_LOG_EVENT(e1, 0, "")
C     ...
      Ierr = MPE_LOG_EVENT(e2, 0, "")
      END
```

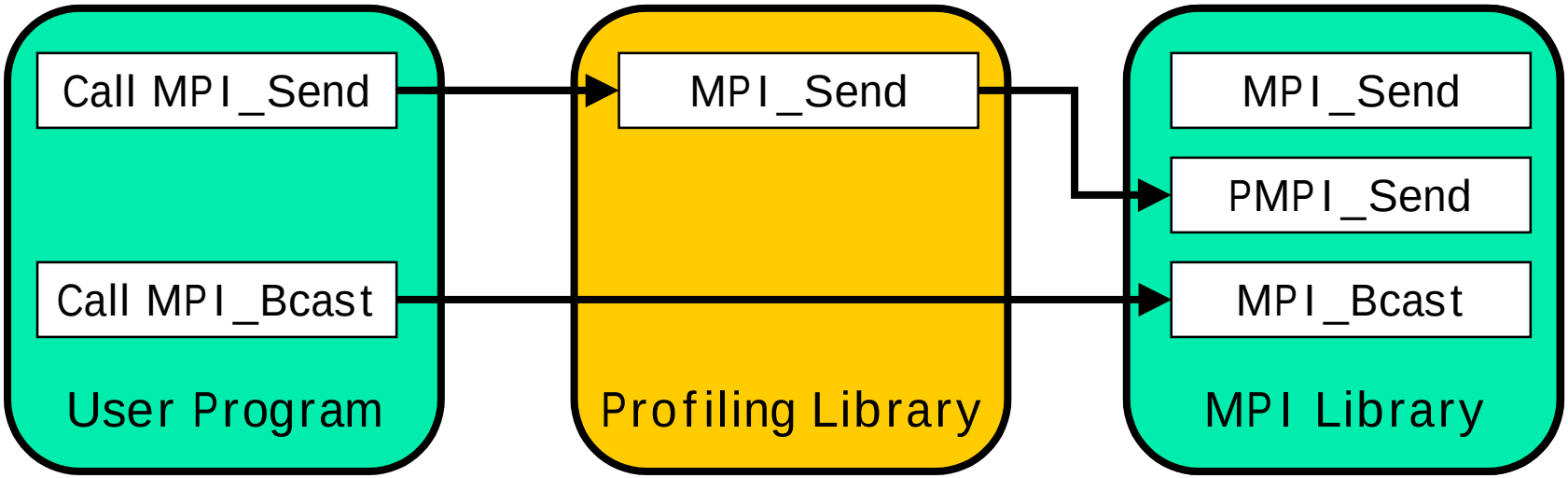
- Selective tracing through
  - `MPE_STOP_LOG();  or MPI_PCONTROL(0); # Turn tracing off`
  - `MPE_START_LOG(); or MPI_PCONTROL(1); # Turn tracing on`

MPICL / Paragraph

- subroutine library for collecting information
  - on communication and user-defined events
  - for Fortran / C MPI programs
- uses the standard MPI profiling interface
- two collection modes
  - profile data, summarizing the number of occurrences, the associated volume statistics, and the time spent in communication and user-defined events for each processor
  - collect detailed traces of each MPI function in PICL format suitable for viewing with Paragraph (nupshot, jumpshot)
- Tested on HP/Convex Exemplar, IBM SP, Intel Paragon, SGI/Cray Origin, SGI/Cray T3E, and on COW / MPICH
- But can easily be ported to any standards-compliant MPI
- http://www.epm.ornl.gov/picl/mpicl.html
- http://www.csar.uiuc.edu/software/paragraph

PMPI: The Standard MPI Profiling Interface

- PMPI allows selective replacement of MPI routines at link time
  ⇒ no re-compilation necessary
- Used by most MPI performance tools
  - DEEP/MPI
  - Vampirtrace
  - MPICH MPE
  - . . .



PMPI Example (C/C++)

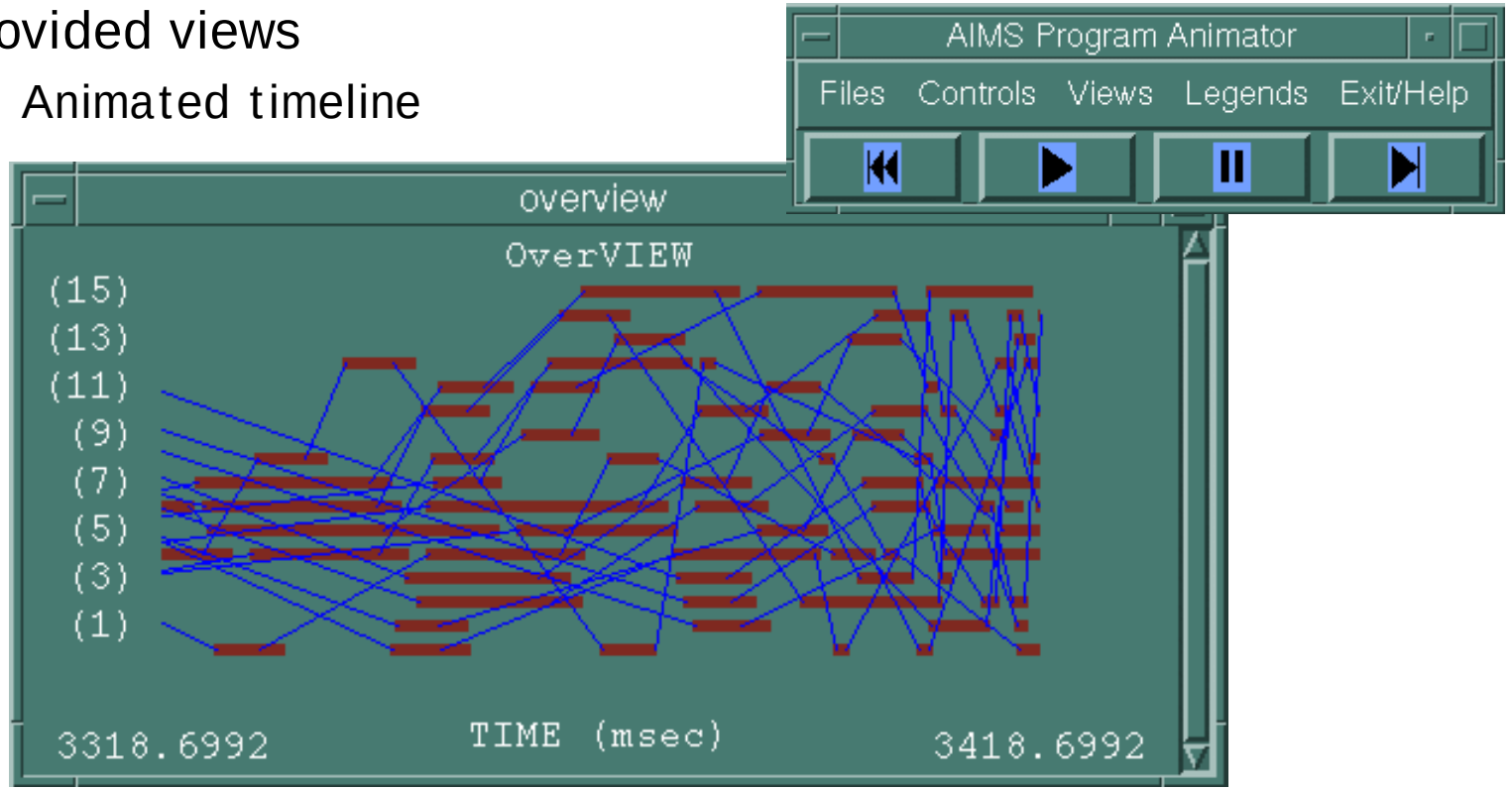
```
#include <stdio.h>
#include "mpi.h"

static int numsend = 0;

int MPI_Send(void *buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm) {
  numsend++;
  return PMPI_Send(buf, count, type, dest, tag, comm);
}

int MPI_Finalize() {
  int me;
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  printf("%d sent %d messages.\n", me, numsend);
  return PMPI_Finalize();
}
```

AIMS

- Automated Instrumentation and Monitoring System
- tool suite for performance measurement and analysis
  - xinstrument: source-code instrumentor for Fortran77 and C message-passing programs (MPI or PVM)
  - monitor: trace-collection library (for IBM SP-2 and workstation cluster (Convex/HP, Sun, SGIs, and IBM)
  - pc: utility for removing monitoring overhead and its effects on the communication patterns
  - trace file visualization and analysis toolkit: AIMS provides four trace post-processing kernels:
    - visualization/animation (VK)
    - text-based profiling (tally)
    - hierarchical performance tuning (xisk)
    - performance prediction (MK)
  - trace converters for PICL (Paragraph) and SDDF(Pablo)

AIMS: VK

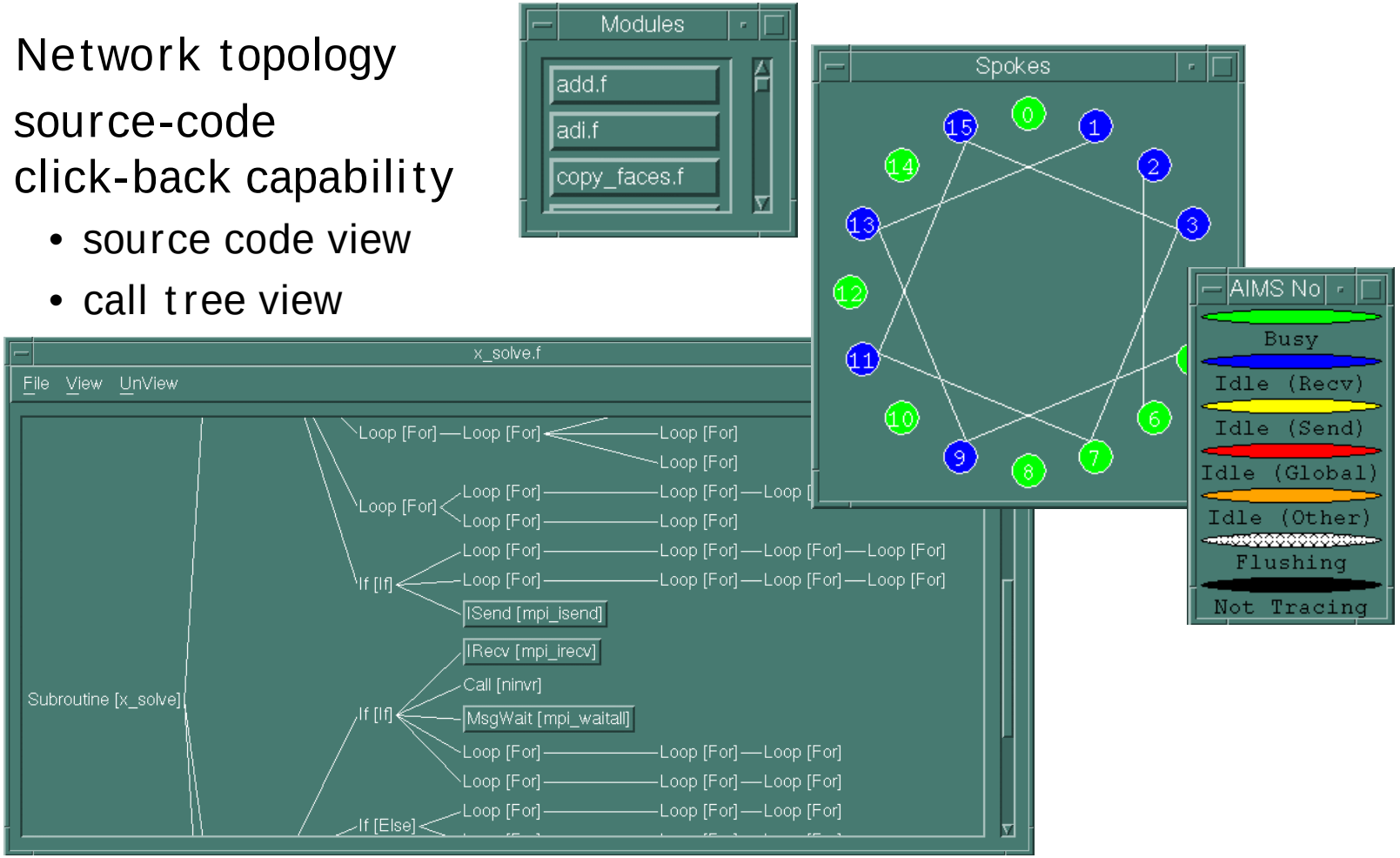
- Provided views
  - Animated timeline



  - The amount, type, duration of various I/O activities
  - Cost for partitioning data structures across the machine
  - The effective rate of execution for each basic block

AIMS: VK

- Network topology
- source-code click-back capability
  - source code view
  - call tree view

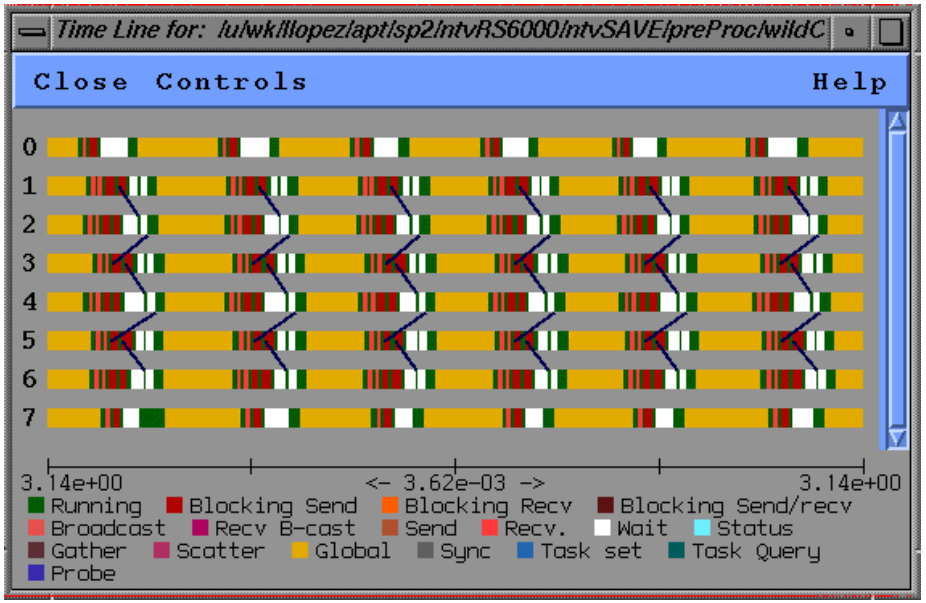


- http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/

NTV

NTV
The NAS Trace Visualizer

- NAS Trace Visualizer
- Supports AIMS trace format and IBM AIX trace format
- Static timeline with zooming / panning



- http://www.nas.nasa.gov/Groups/Tools/Projects/NTV/

Dynaprof

- A portable tool to instrument a running executable with performance probes
  - no re-compilation necessary
  - uses DynInst 2.2 on Linux (maybe others)
  - uses DPCL on AIX
- User interface: command line & script interface
- Uses GNU readline and GNU history for command line editing and command completion
- Open Source
- Work in progress
- http://www.cs.utk.edu/~mucci/dynaprof/

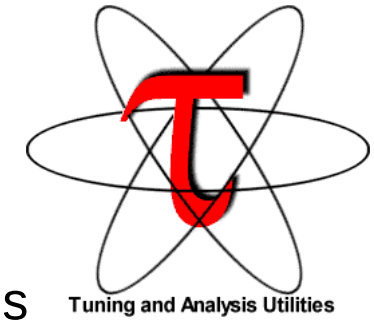
DynaProf Commands

- Load [executable]
- Use [probe [probe_args]
- List [module,function,functions] [regexp]
- Select [<module>]
- Run [args]
- Info
- Help

Dynaprof

- Supports
  - serial
  - threaded (pthreads, OpenMP)
  - MPI
  - mixed applications
- Multiple probe modules
  - wall clock: measures real microseconds using cycle counter
  - hardware counter: counts hardware events using PAPI
  - vprof: generates data for vprof visual profiler
  - lperfex: dynamic version of lperfex command
- Provides
  - Inclusive metrics
  - Exclusive metrics
  - 1 level call tree and metrics

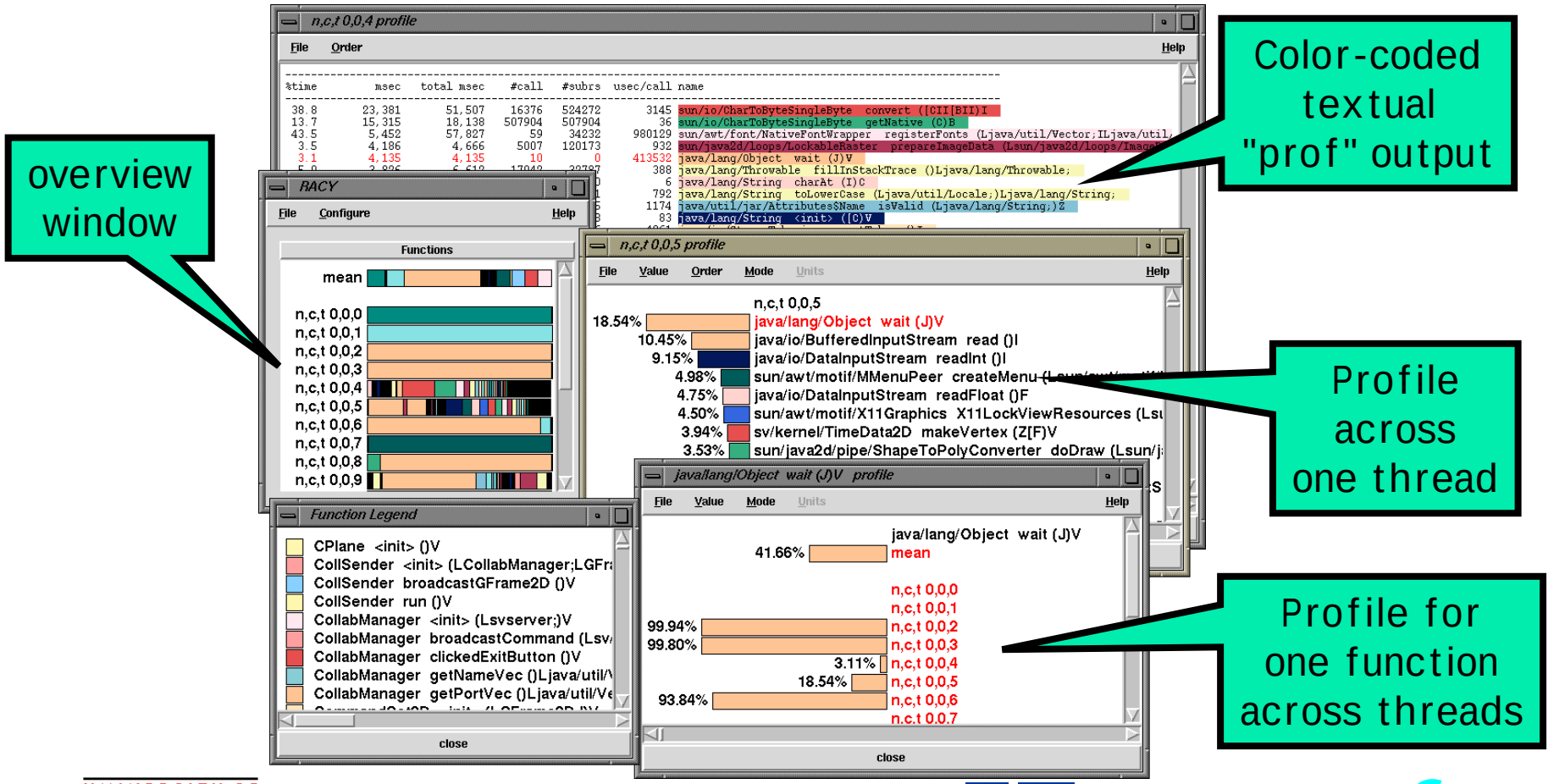
TAU's Portable Profiling and Tracing Package



- Portable toolkit for performance instrumentation, measurement, and analysis of parallel, multithreaded programs
- captures data for functions, methods, basic blocks, statement execution at all execution levels
- instrumentation API that allows the user to select
  - between profiling and tracing
  - application level
  - performance metric (e.g., timers, HW counters (PCL or PAPI))
  - measurement groups for organizing and controlling instrumentation
- works with a powerful code analysis system, the Program Database Toolkit (PDT), to implement automatic source instrumentation and static program browsers linked to dynamic performance information

TAU's Profile Visualization Tool: RACY

- provides graphical displays of all the performance analysis results, in aggregate and per node/context/thread form



TAU's Portable Profiling and Tracing Package

- Trace analysis utilizes the Vampir trace visualization tool

- Supports
  - platforms: SGI Power Challenge and Origin, IBM SP, Intel Teraflop, Cray T3E, HP 9000, Sun, Windows 95/98/NT, Compaq Alpha and Intel Linux cluster
  - languages: C, C++, Fortran 77/90, HPF, HPC++, Java
  - thread packages: pthreads, Tulip threads, SMARTS threads, Java threads, Win32 threads, OpenMP
  - communications libraries: MPI, Nexus, Tulip, ACLMPL
  - compilers: KAI (KCC, Guide), PGI, GNU, Fujitsu, Sun, Microsoft, SGI, IBM and Cray

- http://www.acl.lanl.gov/tau/
- http://www.cs.uoregon.edu/research/paracomp/tau/
- http://www.cs.uoregon.edu/research/paracomp/pdtoolkit/

TAU Instrumentation

- Flexible, multiple, co-operating instrumentation mechanisms
  - Source code instrumentation
    - manual (using TAU API) for C, C++, and Fortran
    - automatic source-to-source translation using PDT (`tau_instrumentor`) for C++
  - Pre-instrumented libraries
    - mathematical libraries (e.g., POOMA, Blitz++)
    - MPI Profiling Interface based wrapper library (`libTauMpi.a`)
  - Dynamic instrumentation
    - using DynInstAPI (`tau_run`)
  - Virtual Machine instrumentation
    - Java Virtual Machine Profiler Interface (JVMPI)

Manual Instrumentation using TAU API

- Plain C++ Code
  - routines & methods need just one `TAU_PROFILE()`
  - arbitrary code regions with `TAU_PROFILE_START() / _STOP()`

```
#include "Profile/Profiler.h"

int main(int argc, char **argv) {
  TAU_PROFILE("main", "int(int,char**)", TAU_DEFAULT);
  TAU_PROFILE_TIMER(ft, "For-loop", "", TAU_USER);

  TAU_PROFILE_START(ft);
  for (int j = 0; j < N; j++) {
    /* -- some code -- */
  }
  TAU_PROFILE_STOP(ft);
}
```

Manual Instrumentation using TAU API

- C++ Template Code
  - dynamically build type information with `TAU_TYPE_STRING()` and `CT()`

```
template<class T, unsigned Dim, class M, class C>
void Field<T,Dim,M,C>::init(Mesh_t&  m,
                            FieldLayout<Dim>& l,
                            const Bconds<T,Dim,M,C>& bc,
                            const GuardCellSizes<Dim>& gc) {
  TAU_TYPE_STRING(taustr, "void(Mesh_t," + CT(l)
                  + "," + CT(bc) + "," + CT(gc) + ")");
  TAU_PROFILE("Field::init ()", taustr, TAU_USER);

  BareField<T,Dim>::initialize(l,gc);
  store_mesh(&m, false);
}
```

Manual Instrumentation using TAU API

- Multi-threaded applications

```
void *thread_func (void *data) {
  TAU_REGISTER_THREAD();
  TAU_PROFILE("thread_func", "void*(void*)",
              TAU_DEFAULT);
  /* -- some code -- */
}

int main() {
  TAU_PROFILE("main", "int()", TAU_DEFAULT);
  pthread_t tid;

  pthread_create(&tid, NULL, thread_func, NULL);
  /* -- some code -- */
}
```

TAU Java Virtual Machine instrumentation

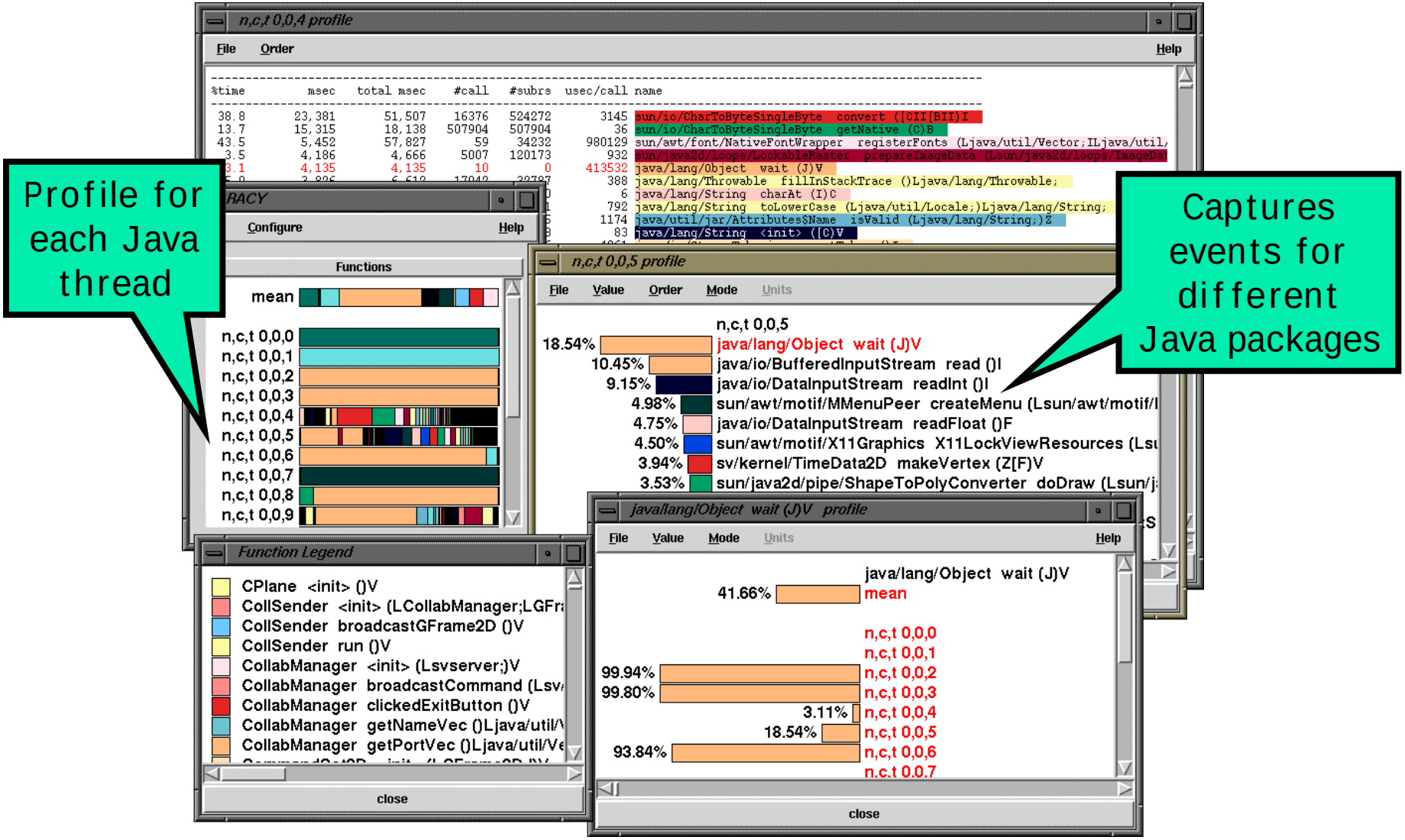
- No changes to Java source code, bytecode or JVM (JDK 1.2+)
- For Profiling

```
% configure -jdk=/usr/local/packages/jdk
% make clean; make install
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:/usr/tau-
  2.x/solaris2/lib
% java -XrunTAU java_app
% racy
```

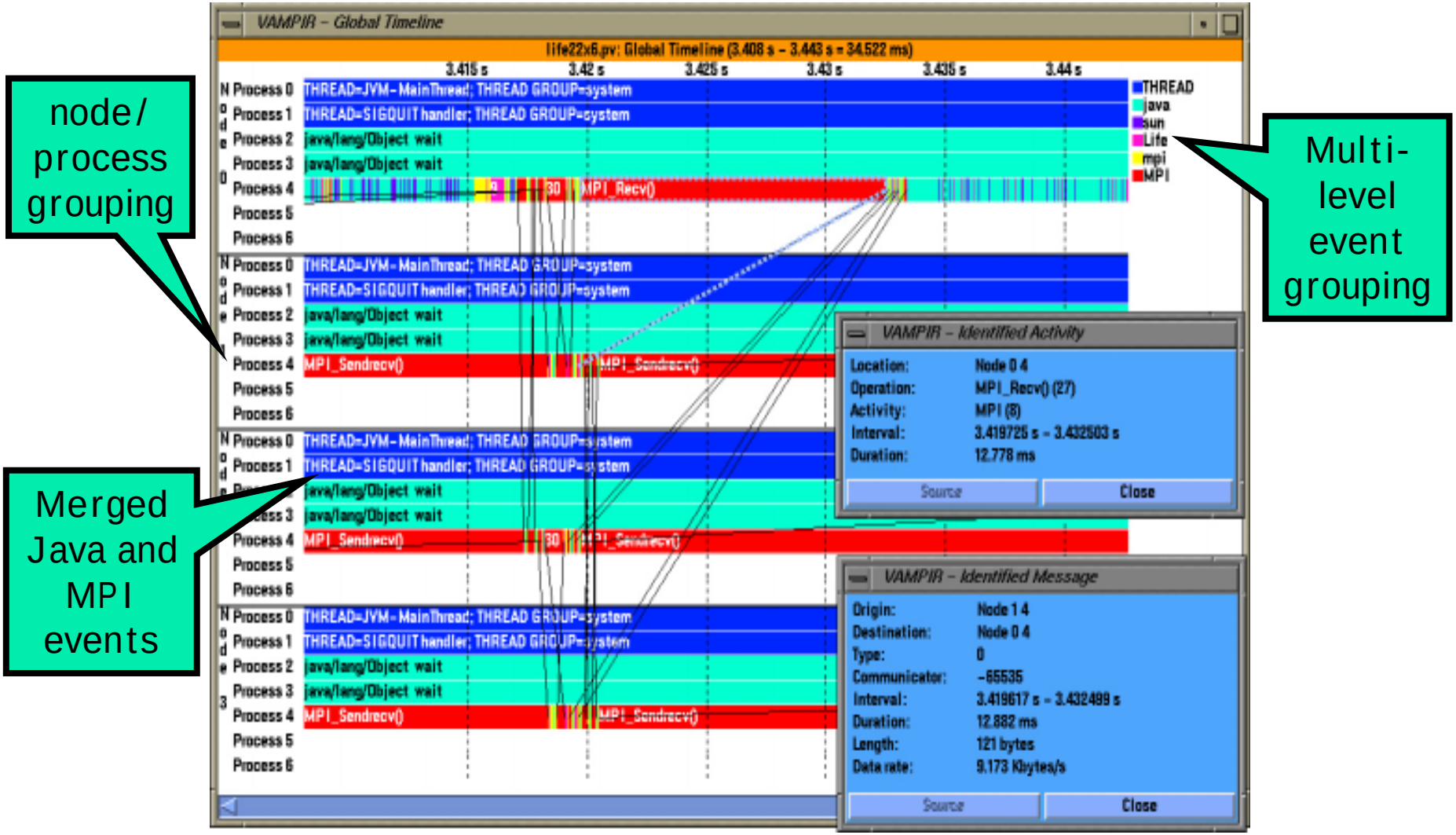
- For Tracing

```
% configure -jdk=/usr/local/packages/jdk -TRACE
% make clean; make install
% java -XrunTAU java_app
% tau_merge *.trc app.trc
% tau_convert -vampir app.trc tau.edf app.pv
% vampir app.pv
```

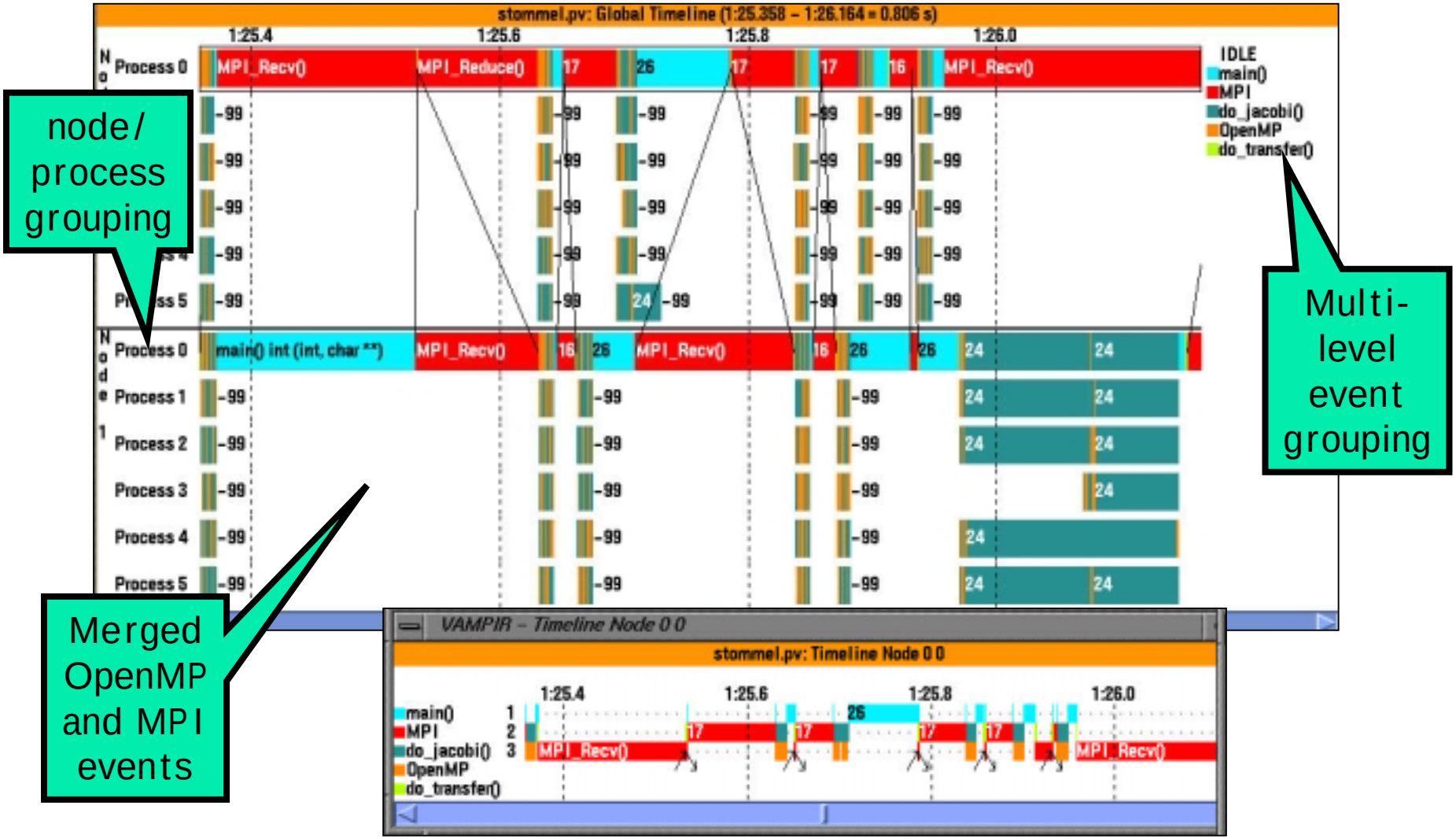
TAU Profiling of Java Application (SciVis)



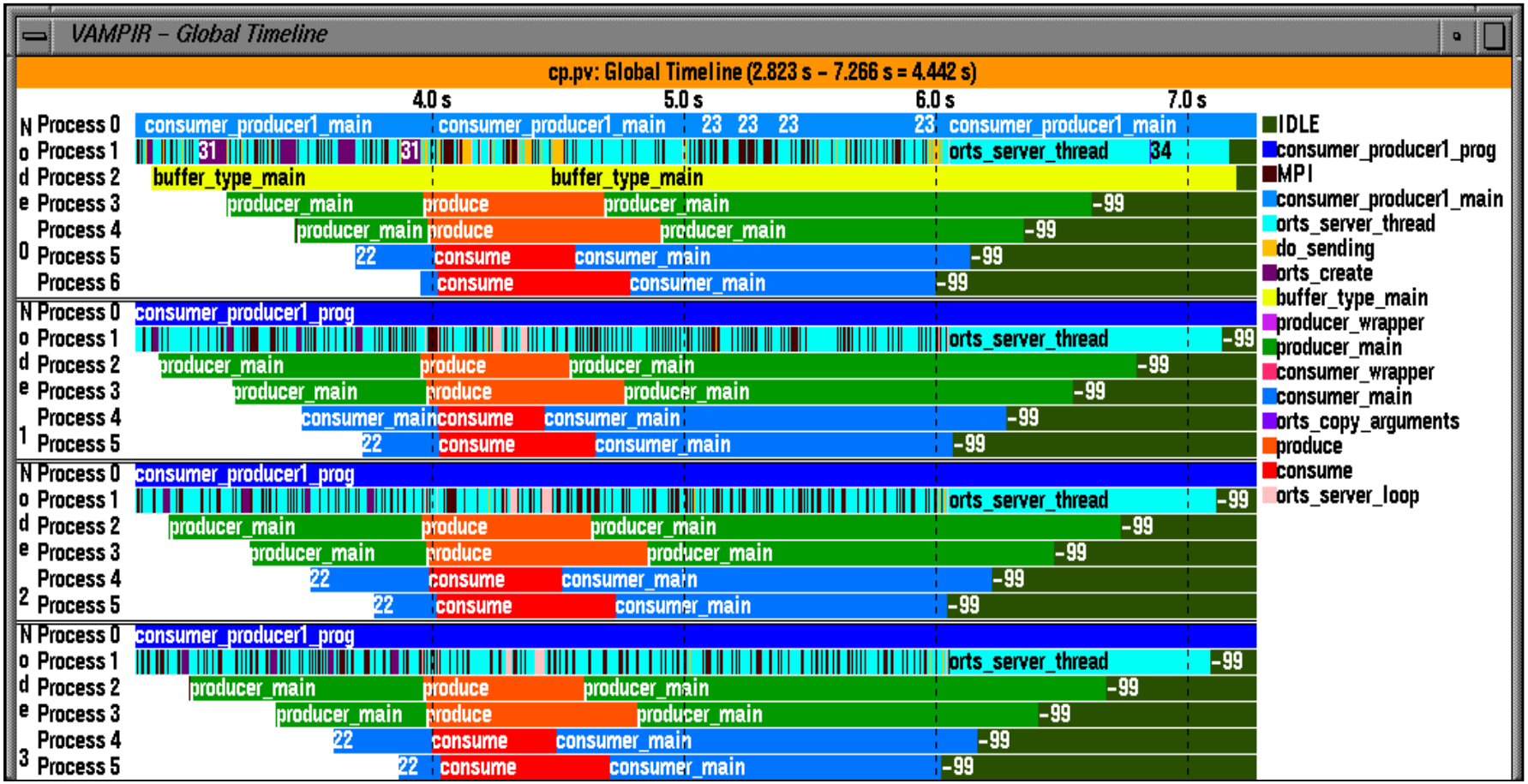
Hyprid Programming Analysis with TAU (Java+MPI)



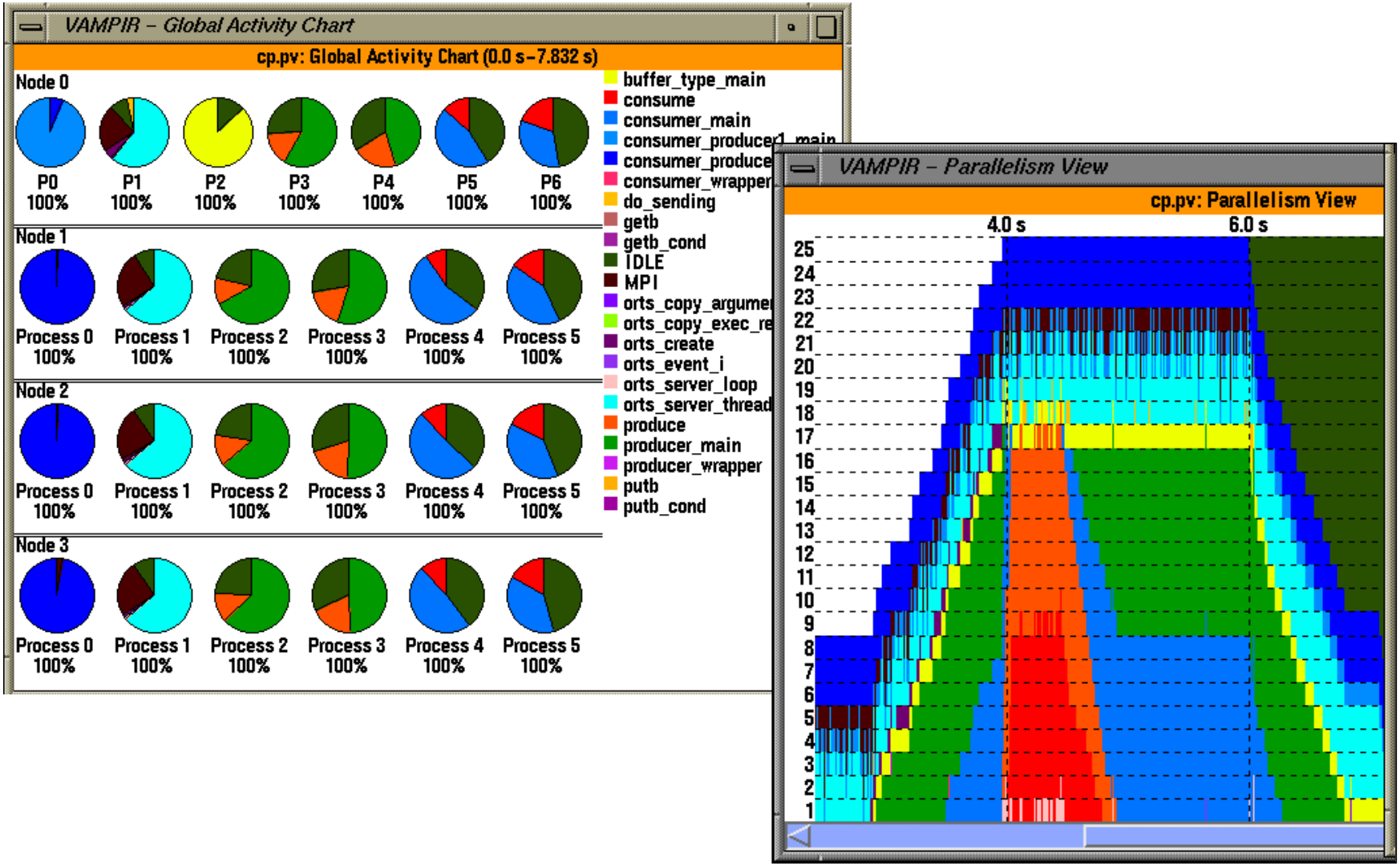
Hyprid Progr. Analysis with TAU (OpenMP+MPI)



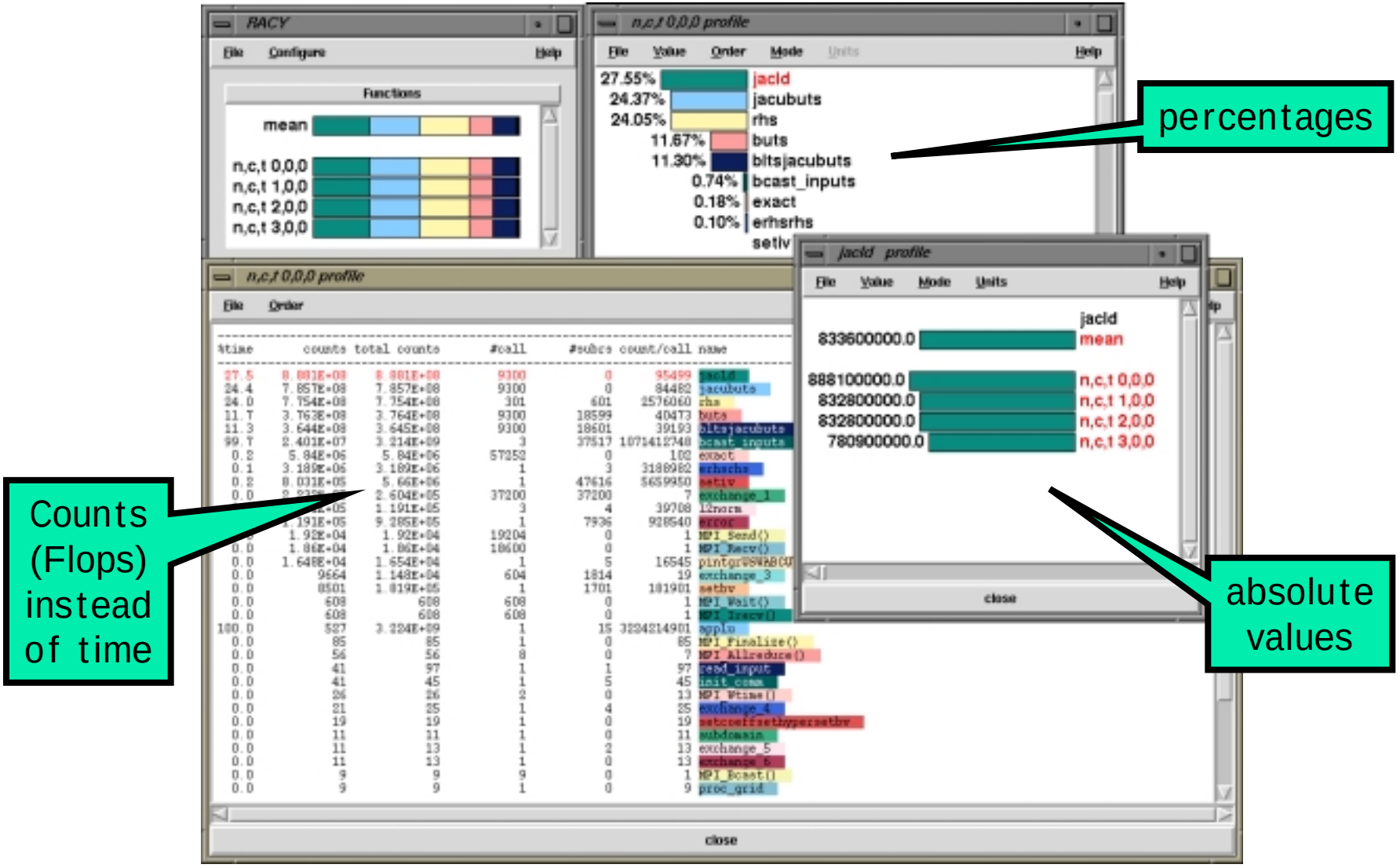
Hyprid Programming Analysis with TAU (Opus+HPF)



Hyprid Programming Analysis with TAU (Opus+HPF)

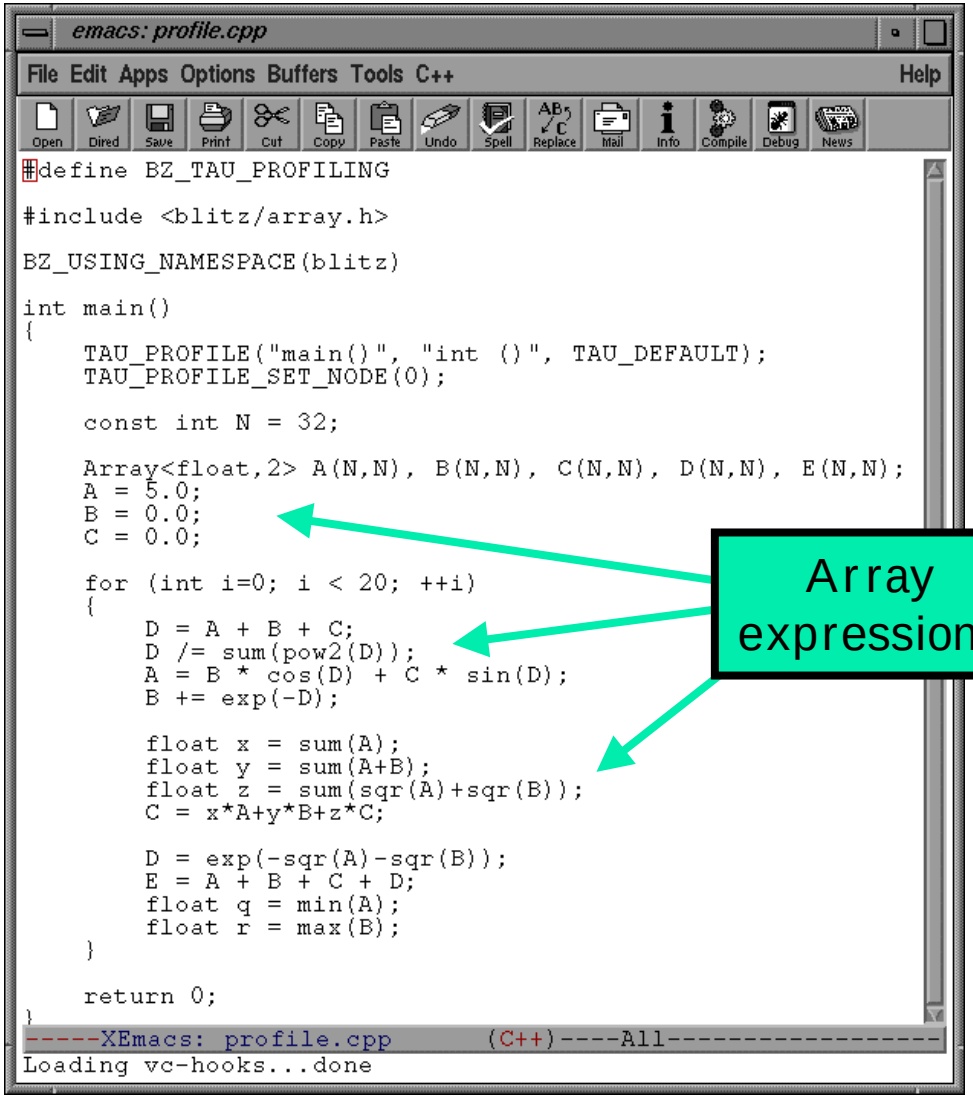


TAU and Hardware Counter (PAPI)



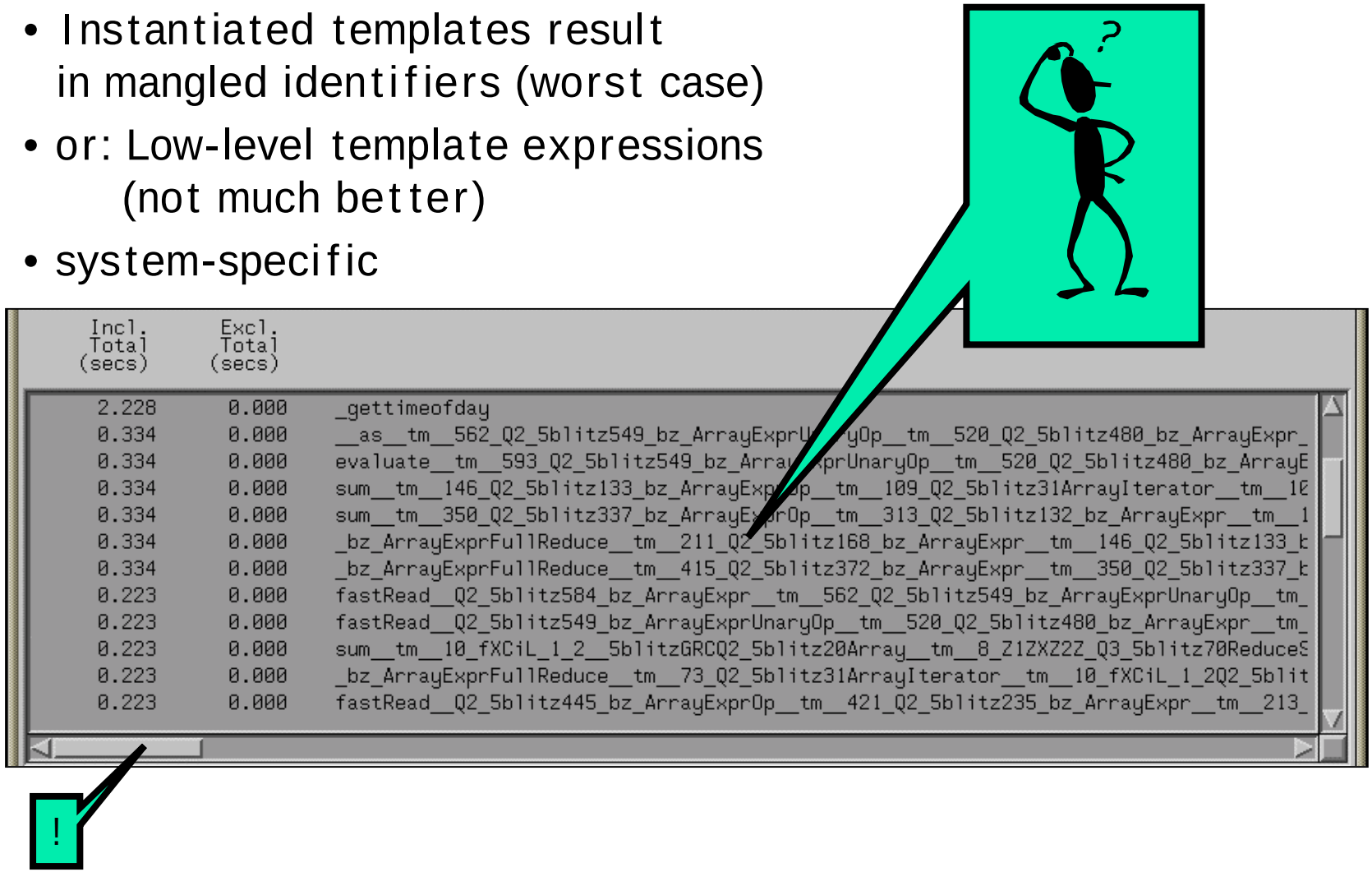
Analyzing Complex C++ Template Code

- High-level objects (Blitz++ arrays)
- Optimizations
  - Array processing
  - expression templates
- Problem: relate performance data to high-level abstractions
- Solution: TAU Mapping API

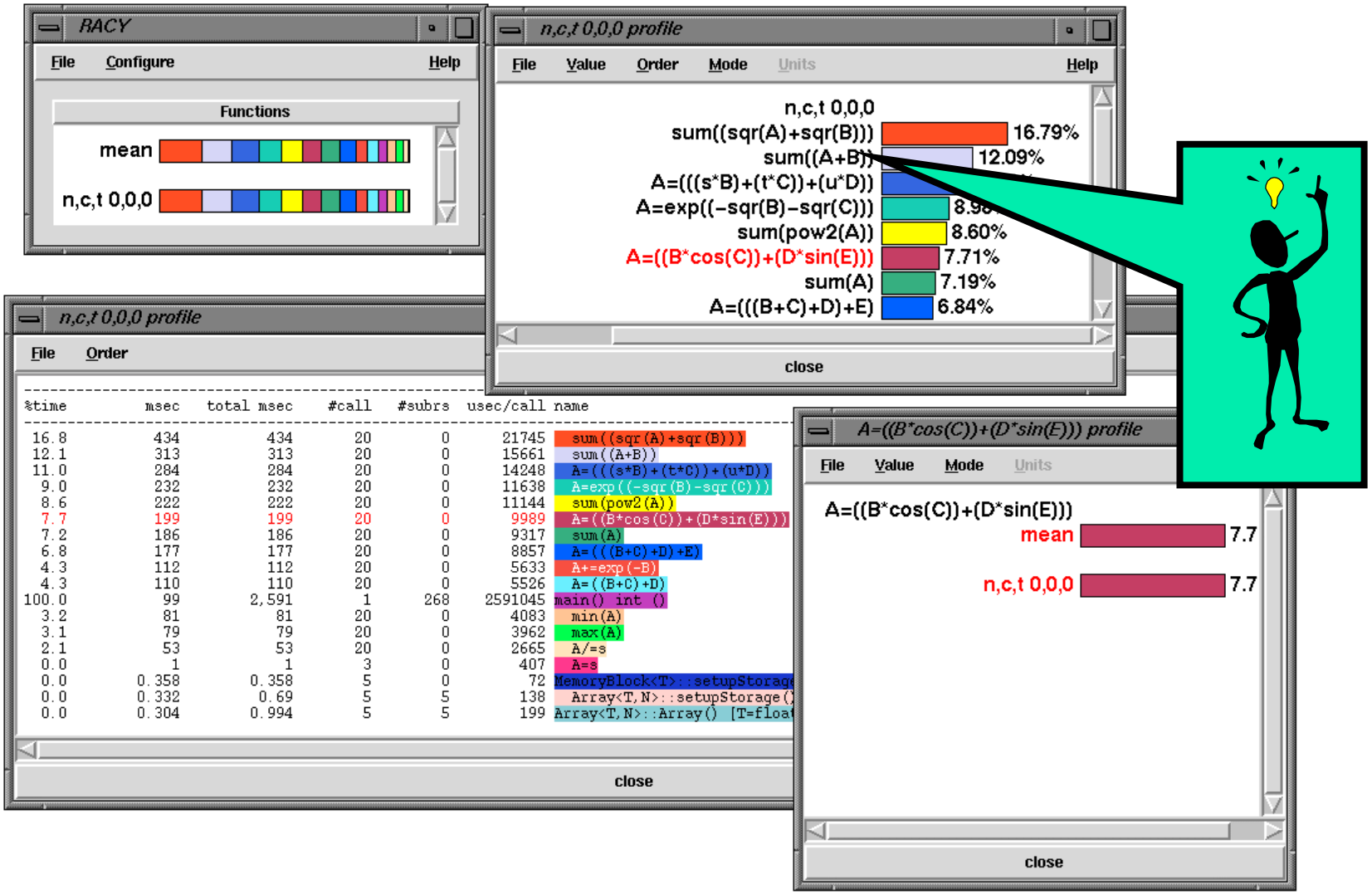


Analyzing Complex C++ Template Code

- Instantiated templates result in mangled identifiers (worst case)
- or: Low-level template expressions (not much better)
- system-specific



Applying the TAU Mapping API



Research: URLs: Profiling

- DCPI, the Digital Continuous Profiling Infrastructure
  http://www.unix.digital.com/dcpi/
- ThreadMon
  http://www.cs.brown.edu/research/thmon/
- WARTS, Wisconsin Architectural Research ToolSet, includes a number of profiling and tracing tools based on EEL:
  - QPT, Quick program Profiling and Tracing system
  - Cprof is a cache performance profiler that processes program traces generated by QPT (above)
  - PP is a Path Profiler

  http://www.cs.wisc.edu/~larus/warts.html
- Vprof (Sandia Livermore), visual profiler (uses PAPI)
  http://aros.ca.sandia.gov/~cljanss/perf/vprof/
- lperfex (Ohio SC Center), Linux perfex (uses PAPI)
  http://www.osc.edu/~troy/lperfex/

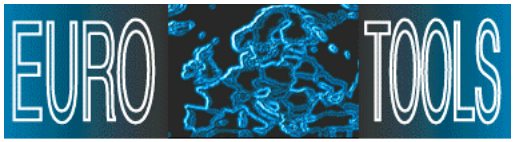
Research: URLs: Tracing

- Review of Performance Analysis Tools for MPI Programs (AIMS, Nupshot, SvPablo, Paradyn, VAMPIR, VT)
  http://www.cs.utk.edu/~browne/perftools-review/
- PABLO performance analysis and visualization toolkit
  http://vibes.cs.uiuc.edu/
  Includes SDDF trace file format
  http://vibes.cs.uiuc.edu/Project/SDDF/SDDFOverview.htm
- SvPablo, graphical source code browser for performance tuning and visualization
  http://vibes.cs.uiuc.edu/Software/SvPablo/svPablo.htm
- Scala, HPF+ post-execution performance tool
  http://www.par.univie.ac.at/~tf/aurora/project4/scala-project/
- MEDEA, MEasurements Description, Evaluation and Analysis
  http://mafalda.unipv.it/Laboratory/research/Medea/index.html

Research: URLs: Catalogs / Misc

- NHSE tool catalog
  `http://www.nhse.org/rib/repositories/ptlib/catalog/`
- Parallel Tools Index from Cornell Theory Center
  `http://www.tc.cornell.edu/UserDoc/Software/PTools/`
- NAS Parallel Tools Group Hotlist
  `http://www.nas.nasa.gov/Groups/Tools/Outside/`
- Nan's Parallel Computing Page (Huge Index)
  `http://www.cs.rit.edu/~ncs/parallel.html`
- Ptools:The Parallel Tools Consortium
  `http://www.ptools.org`

- EuroTools Working Group
  `http://www.eurotools.org`



Fall-back: Home-grown Performance Tools

- If performance analysis and tuning tools are
  - not available
  - too complex and complicated

  it is still possible to do some simple measurements

- Time Measurements
  - clock( )
  - times( )
  - ...
- Hardware Performance Counter Measurements
  - PCL
  - PAPI

- Note: only wall clock time is comparable between systems!

Timer: clock

- ANSI C / C++ function
- returns amount of CPU time (user+system) in microseconds used since first call to `clock()`

```
#include <time.h>

clock_t startime, endtime;
double elapsed;  /* seconds */

startime = clock();
   /* -- long running code -- */
endtime = clock();

elapsed = (endtime - startime)
          / (double) CLOCKS_PER_SEC;
```

Timer: clock

- If measured region is short, run it in a loop
  [applies to all measurements]

```
#include <time.h>

clock_t startime, endtime;
double elapsed;  /* seconds */
int i;

startime = clock();
for (i=0; i<1000; ++i)
   /* -- short running code -- */
endtime = clock();
elapsed = (endtime - startime)
          / (double) CLOCKS_PER_SEC
          / i;
```

- But beware of compiler optimizations!!!

Timer: times

- Std. UNIX function: returns wall clock, system and user time

```
#include <sys/times.h>
#include <limits.h>

struct tms s, e;
clock_t startime, endtime;
double wtime, utime, stime;  /* seconds */

startime = times(&s);
   /* -- long running code -- */
endtime = times(&e);

utime = (e.tms_utime - s.tms_utime) / (double) CLK_TCK;
stime = (e.tms_stime - s.tms_stime) / (double) CLK_TCK;
wtime = (endtime - startime) / (double) CLK_TCK;
```

Timer: gettimeofday

- OSF X/Open UNIX function
- returns wall clock time in microsecond resolution
- base value is 00:00 UCT, January 1, 1970
- portability quirk: optional / missing 2nd argument

```
#include <sys/time.h>

struct timeval tv;
double walltime;  /* seconds */

gettimeofday(&tv, NULL);
walltime = tv.tv_sec + tv.tv_usec * 1.0e-6;
```

Timer: getrusage

- OSF X/Open UNIX function
- fills in structure with lots of information:
  user time, system time, memory usage, context switches, ...

```
#include <sys/resource.h>

struct rusage ru;
double usrtime;  /* seconds */
int memused;

getrusage(RUSAGE_SELF, &ru);
usrtime = ru.ru_utime.tv_sec +
          ru.ru_utime.tv_usec * 1.0e-6;
memused = ru.ru_maxrss;
```

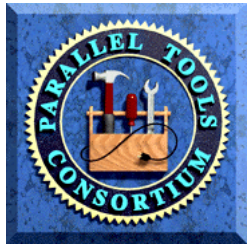
Timer: others

- MPI programmers can use portable MPI wall-clock timer

```
#include <mpi.h>
double walltime;   /* seconds */

walltime = MPI_Wtime();
```

- Fortran90 users can use intrinsic subroutines
  - cpu_time()
  - system_clock()

- Ptools Portable Timing Routines (PTR) Project
  - library of high resolution timers supported across current MPP and workstation platforms
  - see http://www.ptools.org/projects/ptr/



HW Counter: PCL

- Performance Counter Library (PCL)
  - portable API for accessing
    - hardware performance counters
    - elapsed time (2.0)
  - supports counting in
    - user mode
    - system mode
    - user+system mode
  - provides C, C++, Fortran, and Java interfaces
  - nested counters
  - thread-safe for Linux, IBM (possibly more) (2.0)

  - see http://www.fz-juelich.de/zam/PCL/

PCL: API Example

```
#include <stdio.h>
#include "pcl.h"

PCL_DESCR_TYPE  d;
PCL_CNT_TYPE    ires[2];
PCL_FP_CNT_TYPE fpres[2];
int ctr[2]          = { PCL_FP_INSTR, PCL_CYCLES };
unsigned int flags = PCL_MODE_USER;

if (PCLinit(&d) != PCL_SUCCESS) error();
if (PCLstart(d, ctr, 2, flags) != PCL_SUCCESS) error();
    /* -- some other code -- */
if (PCLstop(d, ires, fpres, 2) != PCL_SUCCESS) error();
printf("%15.0f flops in %15.0f cycles\n",
                (double) ires[0], (double) ires[1]);
if (PCLexit(d) != PCL_SUCCESS) error();
```

PCL: Availability

- Supported platforms
  - SGI/Cray T3E with DEC Alpha 21164 CPU
  - Compaq/DEC with DEC Alpha 21164 and 21264 CPU
  - SUN with UltraSPARC I and II CPU
  - SGI Origin with MIPS R10000 and R12000 CPU
  - IBM with PowerPC 604, 604e, Power3, and Power3-II CPU
  - Intel/Linux with Pentium, Pentium Pro, Pentium II, and Pentium III CPU
  - Hitachi SR8000
- Tools based on PCL
  - TAU (University of Oregon)
  - autopcl (INRIA)
  - TOPAS (Forschungszentrum Jülich)

HW Counter: PAPI

- Performance Data Standard and API (PAPI)
- Ptools Project
  - portable API for accessing
    - hardware performance counters
    - wall-clock time (high resolution, low overhead)
    - process virtual time
  - Supports any countable event on all platforms
  - High level API for application engineers
  - Low level (native events) API for tools designers
  - SVR4 profil() compatible hardware profiling interface
  - thread-safe (with bound threads)
  - provides C, C++, Fortran interfaces
  - nested counters

  - see http://icl.cs.utk.edu/projects/papi/

PAPI: High Level API Example

```
#include <stdio.h>
#include “papi.h”

unsigned long long res[2];
int ctr[2] = { PAPI_FP_INS, PAPI_TOT_CYC };

if (PAPI_query_event(ctr[0]) != PAPI_OK) error();
if (PAPI_query_event(ctr[1]) != PAPI_OK) error();

PAPI_start_counters(ctr, 2);
    /* -- long running code -- */
PAPI_stop_counters(res, 2);
printf("%lld flops in %lld cycles\n", res[0], res[1]);
```

PAPI: Low Level API Example

```
#include <stdio.h>
#include “papi.h”

int EventSet;
int native = (0x47<<8) | 1; /* -- Pentium event 0x47 on counter 1 -- */
long long cntrs[2];

PAPI_library_init(PAPI_VER_CURRENT);
if (PAPI_create_eventset(&EventSet) != PAPI_OK) error();
if (PAPI_query_event(PAPI_L1_TCM) != PAPI_OK) error();
if (PAPI_add_event(&EventSet,PAPI_L1_TCM) != PAPI_OK) error();
if (PAPI_add_event(&EventSet,&native) != PAPI_OK) error();
if (PAPI_start(EventSet) != PAPI_OK) error();
    /* -- some other code -- */
if (PAPI_stop(EventSet, cntrs) != PAPI_OK) error();
PAPI_shutdown();
```

PAPI: Wallclock and Process Virtual Time

```
#include “papi.h”

/* -- Only required for threaded programs:  -- */
/* -- function to return unique thread id    -- */
unsigned long tid(void) { return my_thread_id; }

if (PAPI_library_init(PAPI_VER_CURRENT) !=
        PAPI_VER_CURRENT) error();

/* -- Only required for threaded programs -- */
if (PAPI_thread_init(tid, 0) != PAPI_OK) error();

long long current_time = PAPI_get_real_usec();
long long process_virtual_time = PAPI_get_virt_usec();
```

PAPI: Availability

- Supported platforms
  - Cray T3E with DEC Alpha 21164 CPU
  - SUN with UltraSPARC I and II CPU
  - SGI Origin with MIPS R10000 and R12000 CPU
  - IBM with PowerPC 604, 604e, Power3 CPU
  - Intel/Linux with Pentium, Pentium Pro, Pentium II, Pentium III, and AMD K7 CPU

- Tools based on PAPI
  - DEEP/PAPI (Veridan -- Pacific-Sierra Research)
  - TAU (University of Oregon)
  - lperfex (Ohio Supercomputing Center)
  - vprof (Sandia National Laboratories, Livermore)
  - dynaprof (University of Tennessee, IBM ACTC)
  - SvPablo (IBM ACTC)

The Future: Automatic Performance Analysis?

- Paradyn (University of Wisconsin-Madison)
  `http://www.cs.wisc.edu/paradyn/`
- Autopilot (University of Illinois at Urbana-Champaign)
  `http://vibes.cs.uiuc.edu/Project/Autopilot/AutopilotOverview.htm`
- Kappa-PI (Universitat Autònoma de Barcelona)
  `http://www.caos.uab.es/kpi.html`
- KOJAK (Forschungszentrum Jülich)
  `http://www.fz-juelich.de/zam/kojak/`
- Aurora (Universität Wien)
- Peridot (Technische Universität München)
- OVALTINE, ADAPT (University of Manchester)

- Esprit Working Group APART
  Automatic Performance Analysis: Resources and Tools
  `http://www.fz-juelich.de/apart/`

Acknowledgements

- The author wants to thank the following persons for providing very valuable material and advice

  - Reiner Vogelsang, Cray Research, now SGI
  - Jim Galarowicz, Cray Research, now SGI
  - Alexandros Poulos, SGI
  - Luiz A. DeRose, IBM Research, ACTC
  - David McNamara, Veridan -- Pacific-Sierra Research
  - Hans-Christian Hoppe, Pallas GmbH
  - Werner Krotz-Vogel, Pallas GmbH
  - Bob Kuhn, Kuck & Associates, Inc.
  - Uwe Fladrich, Technische Universität Dresden
  - Sameer Shende, University of Oregon
  - Philip Mucci, University of Tennessee & IBM Research, ACTC
  - Rudolf Berrendorf, Forschungszentrum Jülich

Performance Analysis and Tuning of Parallel Programs: Resources and Tools

PART1

Introduction and Overview

Michael Gerndt (Technische Universität München)

PART2

Resources and Tools

Bernd Mohr (Forschungszentrum Jülich)

PART3

Automatic Performance Analysis with Paradyn

Barton Miller (University of Wisconsin-Madison)

Outline

- Brief motivation
- Sample analysis/tuning session with Paradyn
  - simple application start/attach (with no preparation required)
  - usual performance summaries/visualizations
  - automated exigency/bottleneck search
- Paradyn technologies and architecture
  - ***Dynamic Instrumentation***
  - the ***Performance Consultant***
- On-going research and development
  - current status and availability

Some History and Motivation

Experience with IPS-2 tools project:

- Trace-based tool running on workstations, SMP (Sequent Symmetry), Cray Y-MP.
- Commercial Success: in Sun SPARCWorks, Informix OnView, NSF Supercomputer Centers.
- Many real scientific and database/transaction users.

A 1992 Design Meeting at Intel:

- Goal was to identify hardware support for profiling and debugging for the Touchstone Sigma (Paragon) machine.
- Estimated the cost of tracing, using IPS-2 experience, and extrapolated to the new machine:
  2 MB/sec/node $\rightarrow$ 2 GB/sec of trace data for 1000-node machine.

An entirely new approach was required!

The Challenges

Scalability:

- Large, complex programs
- 100s or 1000s of nodes
- Long runs (hours or days)

Automation:

- Simplify programmer's task in the tuning process
- Manage increasing complexity of application/system

Heterogeneity:

- COWs, SMPs, MPPs
- UNIX, WindowsNT, Linux
- SPARC, x86, MIPS, PowerPC, Alpha
- Source languages: C, C++, Fortran
- Parallel libraries: PVM, MPI

Extensibility:

- Incorporate new sources of performance data and metrics
- Include new visualizations

Searching for Bottlenecks

1. Start with coarse-grain view of whole program performance.
2. When a problem seems to exist, collect more information to refine this problem.
3. Repeat step #2 until a precise enough cause is determined.
4. Collect information to try to refine (*localize*) the problem to particular hosts, processes, modules, functions, files, etc.
5. Repeat step #4 until a precise enough location is determined.

This type of iteration can take many runs of a program (and slow re-builds) to reach a useful conclusion.

Paradyn allows the programmer to do this on-the-fly in a single execution

Paradyn performance analysis/tuning scenario

Parallel/distributed application running on multiple nodes...

How's it doing?

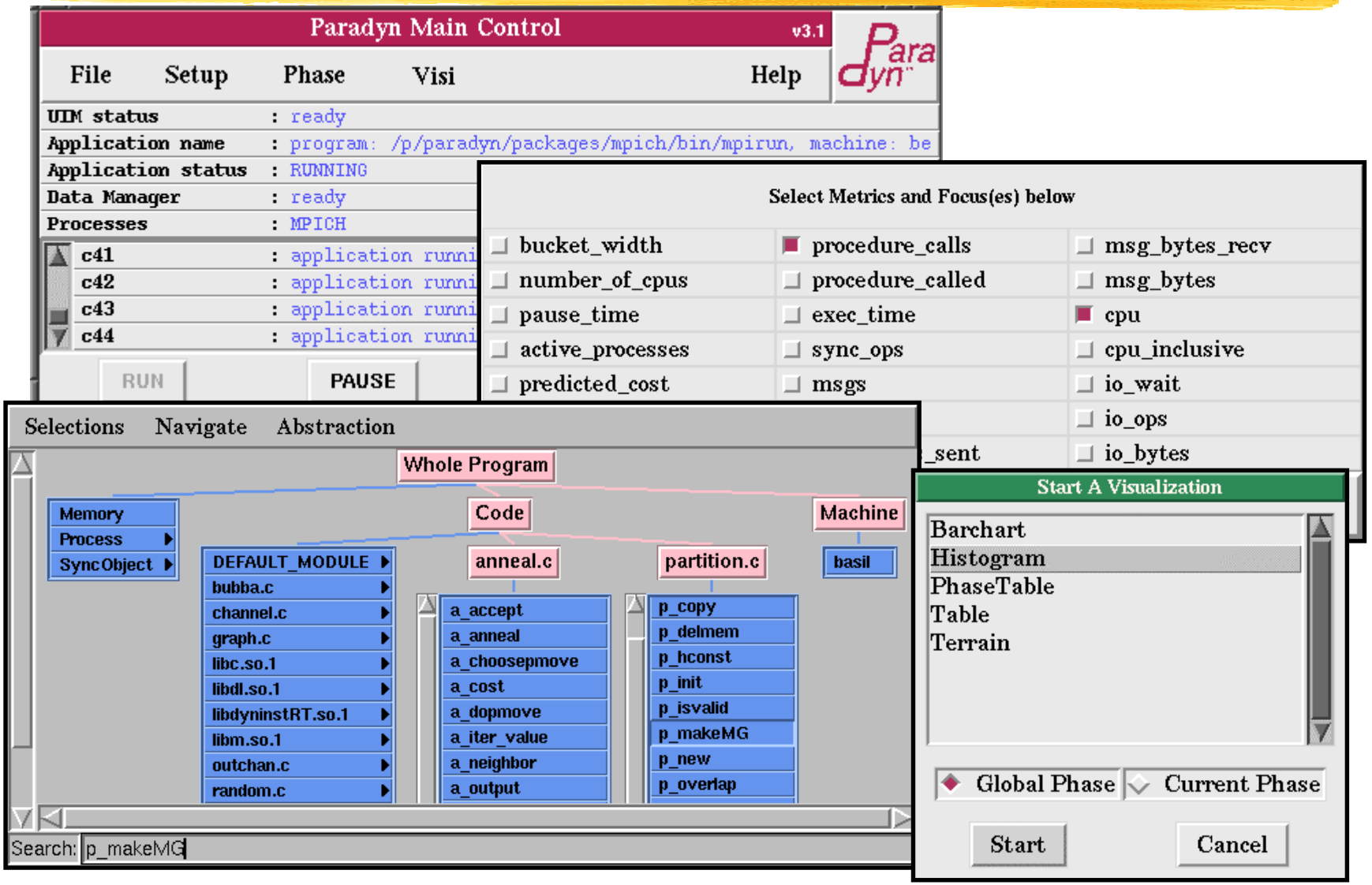
Let's investigate:

- Start Paradyn and attach to running application
  - Executable (and dynamic libraries) parsed to determine structural components and resources used

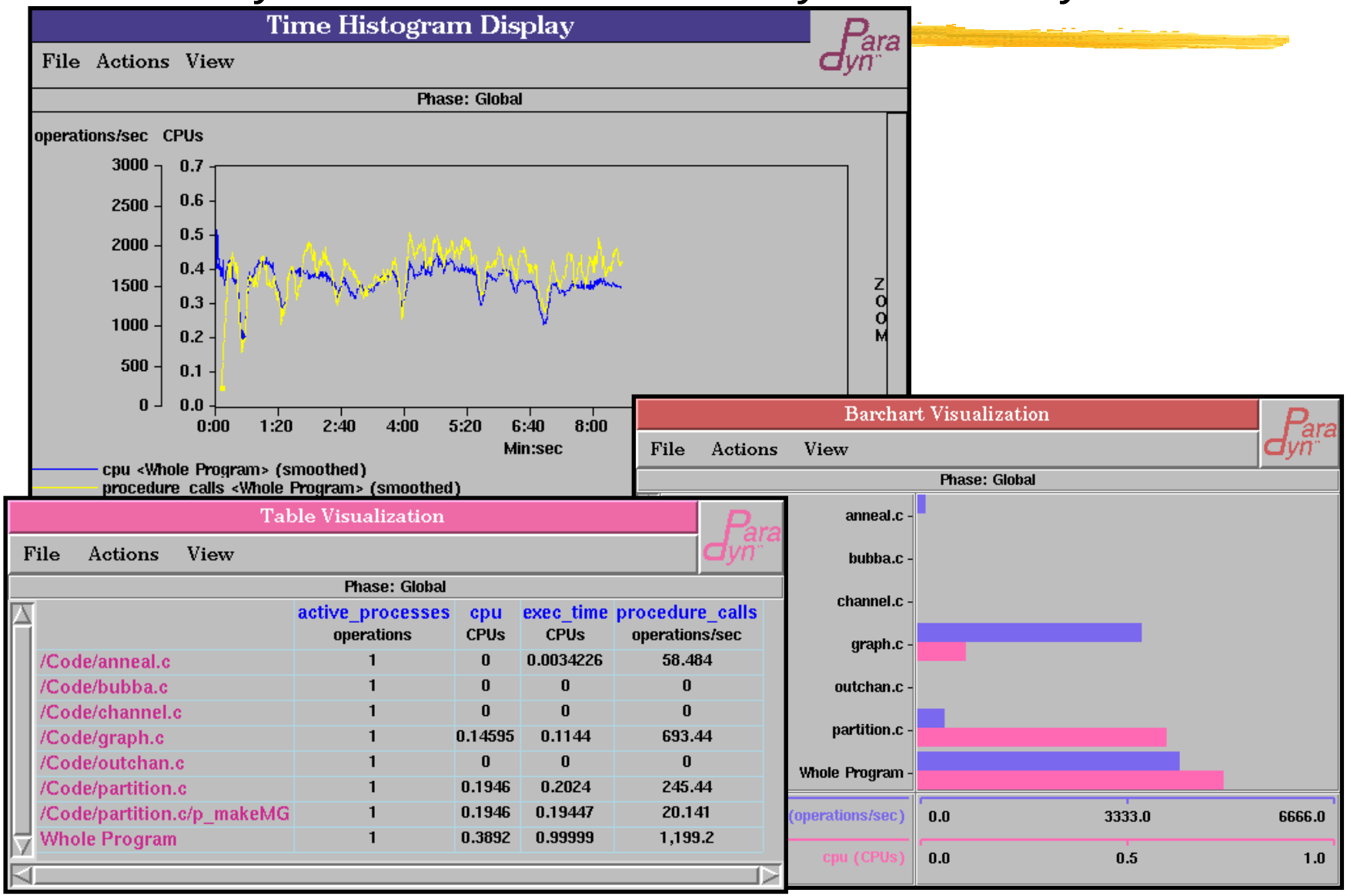
Have some idea where the problem may lie?

- Select components, performance metrics and presentations
  - Required instrumentation generated and patched into application
  - Accumulated metric data periodically sampled
- Examine presented information for insight into potential cause
- Implement a possible solution (rebuild and repeat...)

Paradyn <metric,focus> & visualization selection



Paradyn assorted time-history & summary visis



Paradyn performance analysis/tuning scenario (cont'd)

No idea where to start (or not finding the problem quickly)?

- Initiate Paradyn's *Performance Consultant* automated search
  - A pre-defined (tunable) set of hypotheses are considered in turn (to determine what type of problems may exist)
  - Instrumentation enabled as required and execution characteristics tracked for each current hypothesis
  - Step-wise hierarchical refinement to resolve bottleneck(s) (to focus where the problems are located)
  - Manage instrumentation costs/perturbations (trading accuracy with speed to judgement)
- Follow-up with additional execution statistics or time histories targeted at understanding the likely bottleneck(s)
  - Consider execution phases with distinct characteristics separately (to isolate when the problems manifest)
- Implement a possible solution and repeat...

Decision Support: "Performance Consultant"

Answer three questions:

- ***Why*** is the program running slowly?
- ***Where*** in the program is this occurring?
- ***When*** does this problem occur?

We create a regular structure for the causes of bottlenecks.

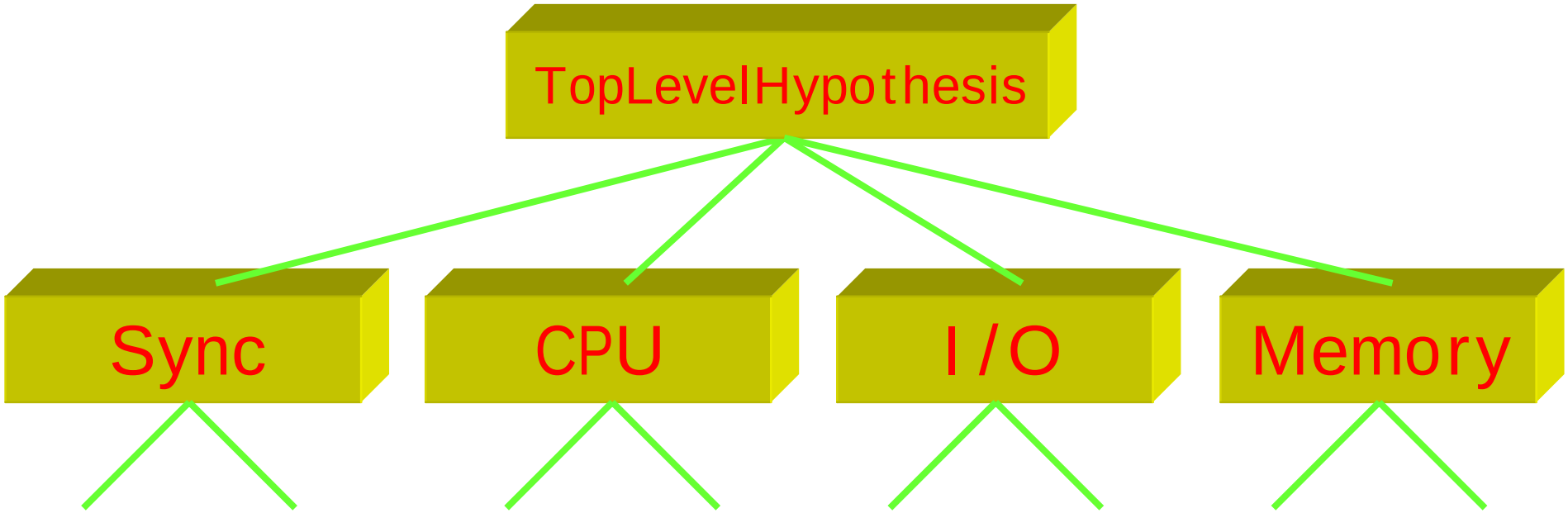
This makes it possible to automate the search for bottlenecks.

The “Why” Axis: A Hierarchy of Bottlenecks

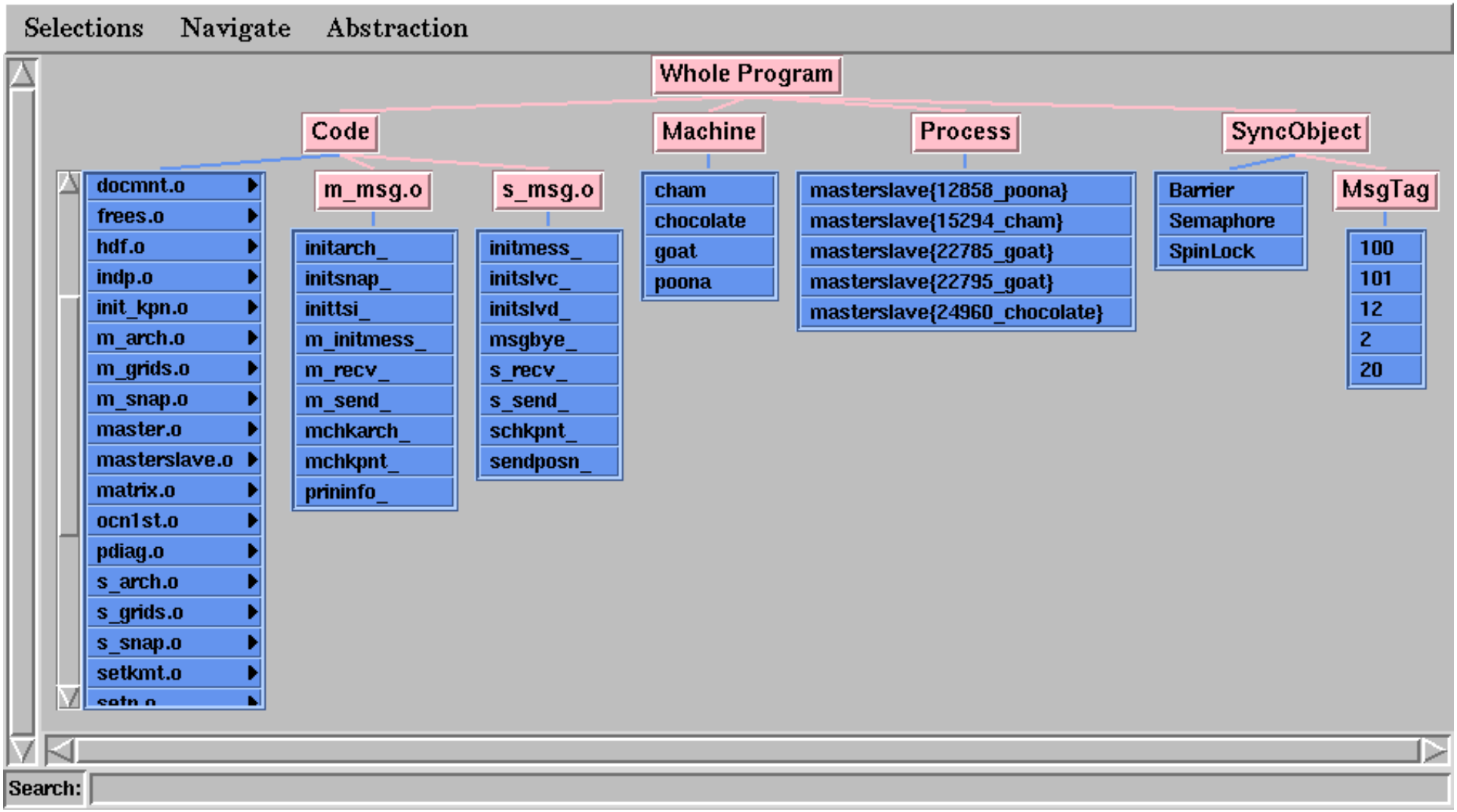
- Potential bottlenecks are represented as hypotheses.
- Evaluating hypotheses triggers dynamic instrumentation.
- Bottlenecks are based on user-set thresholds:

  Total sync blocking time < 25% of exec time

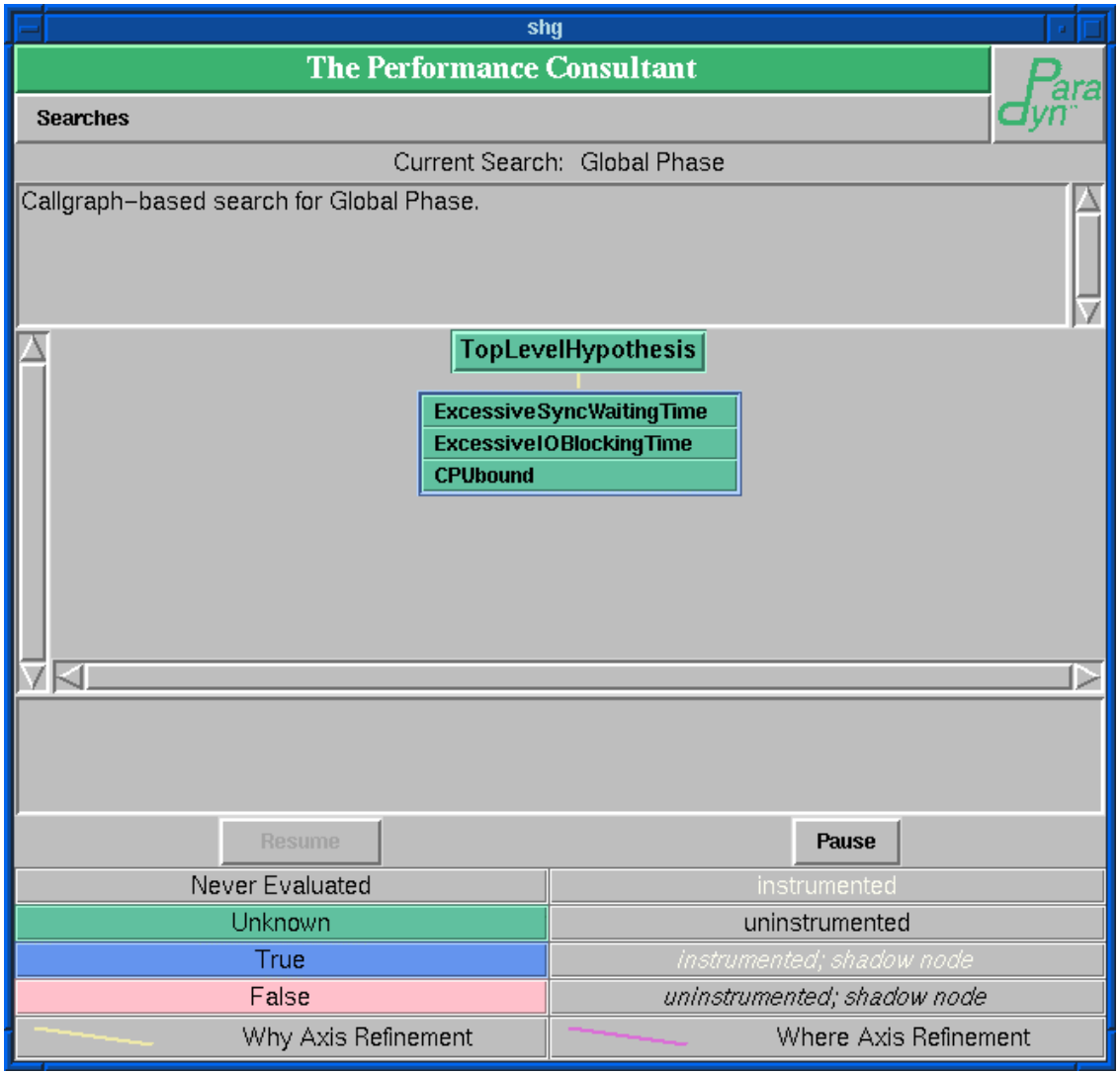
The “Why” Axis: A Hierarchy of Bottlenecks



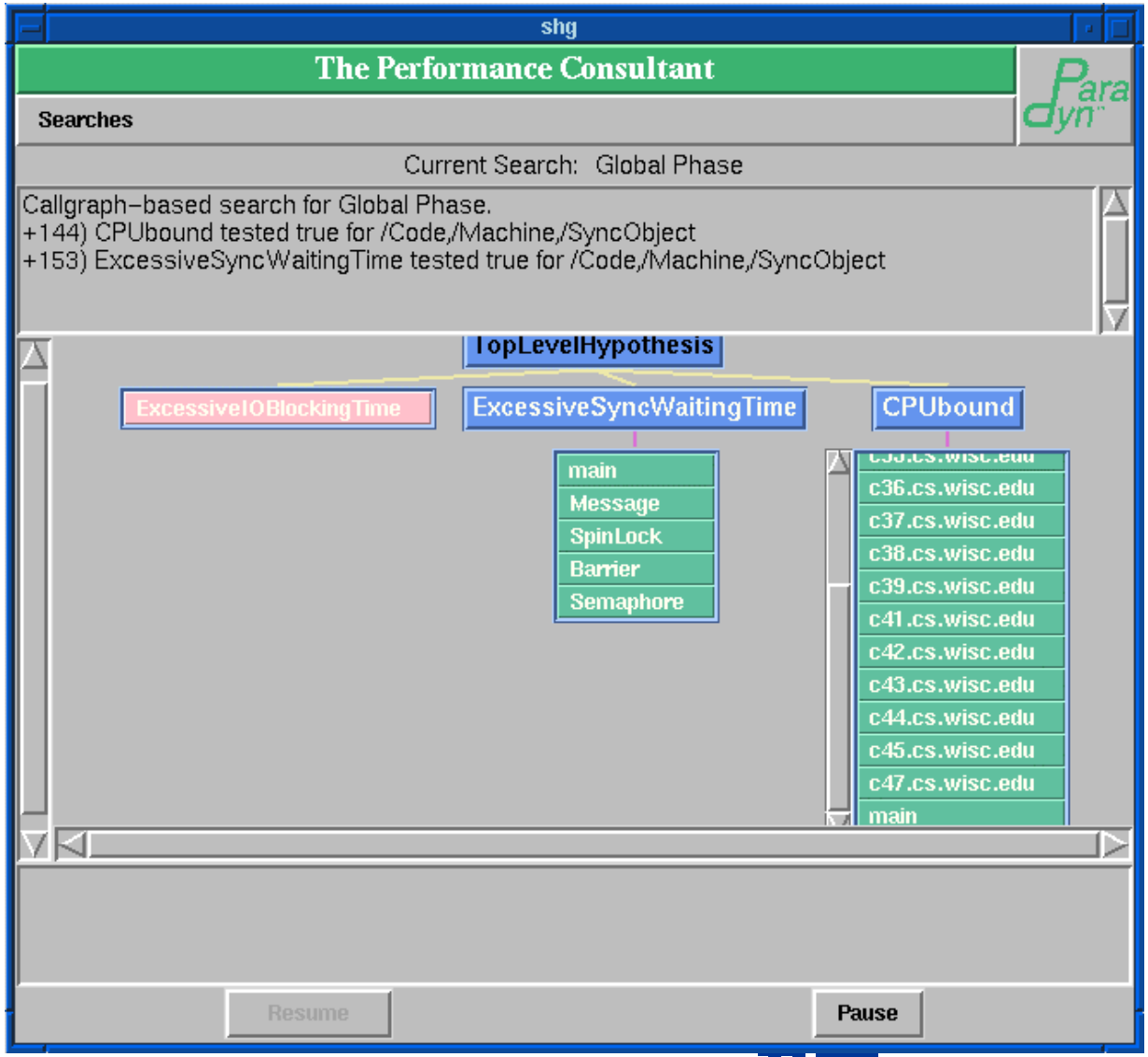
Sample resource “Where Axis”



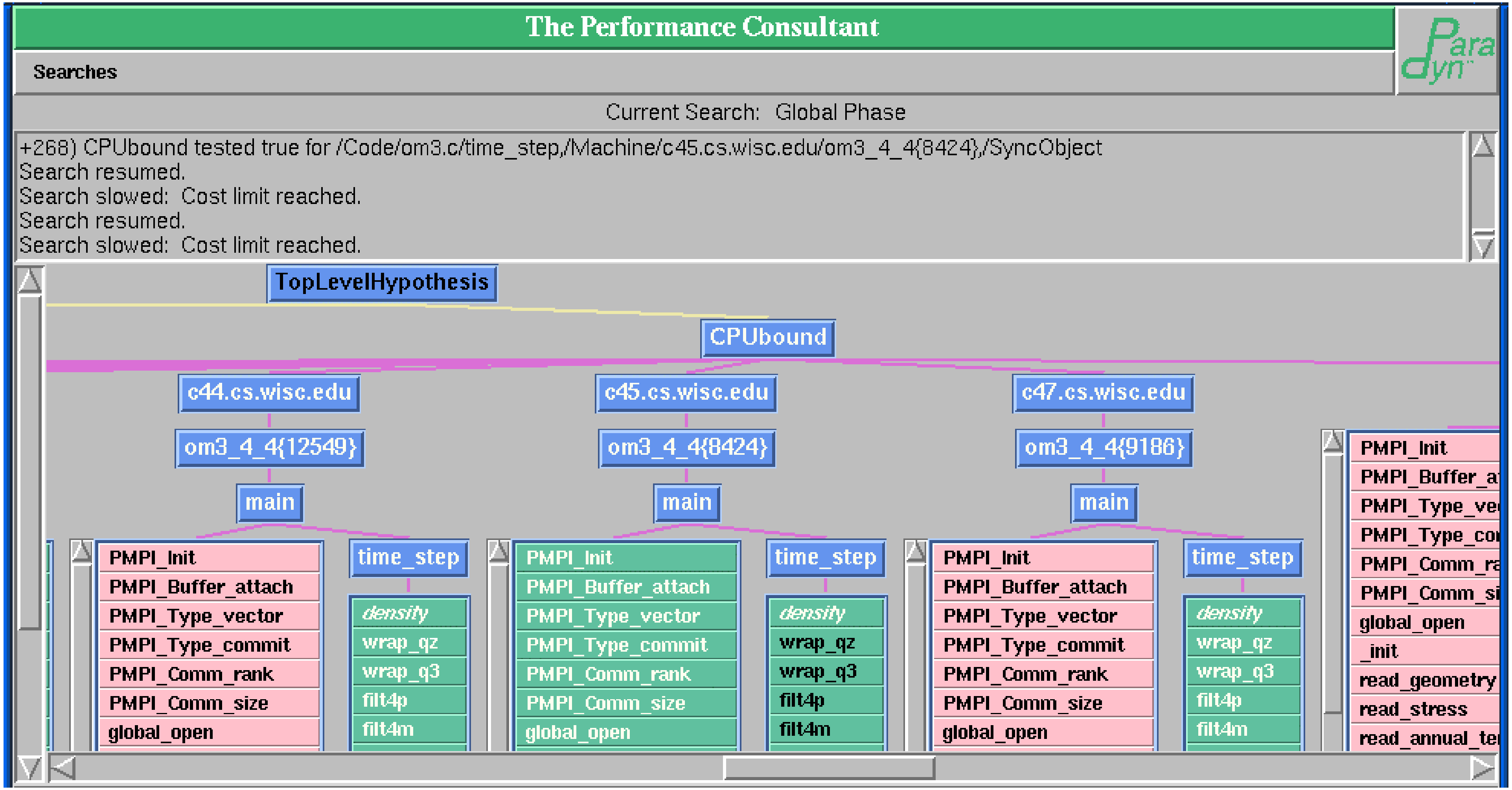
Performance Consultant search begins...



Performance Consultant 1st refinement ...

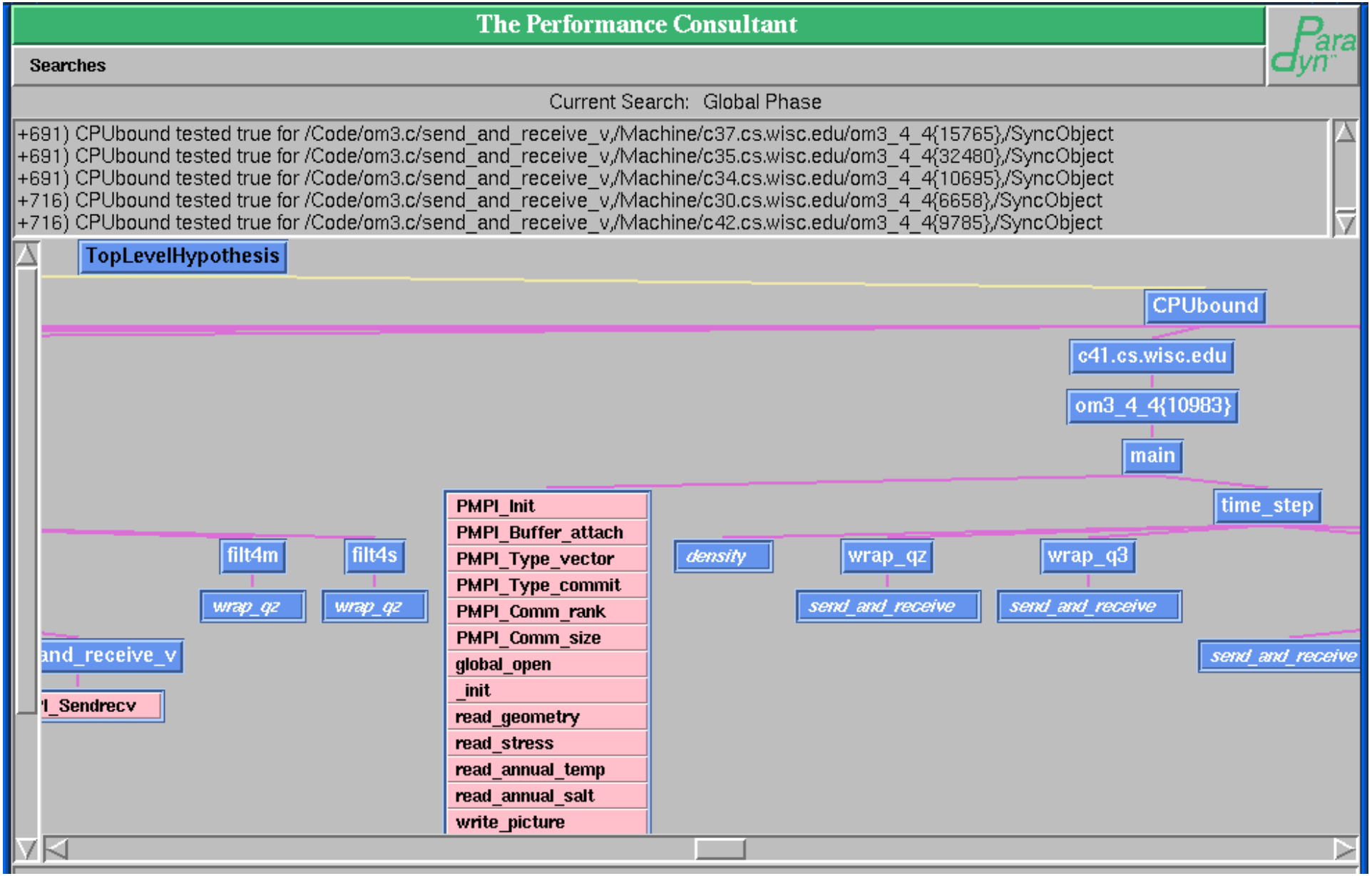


Performance Consultant 2nd refinements ...

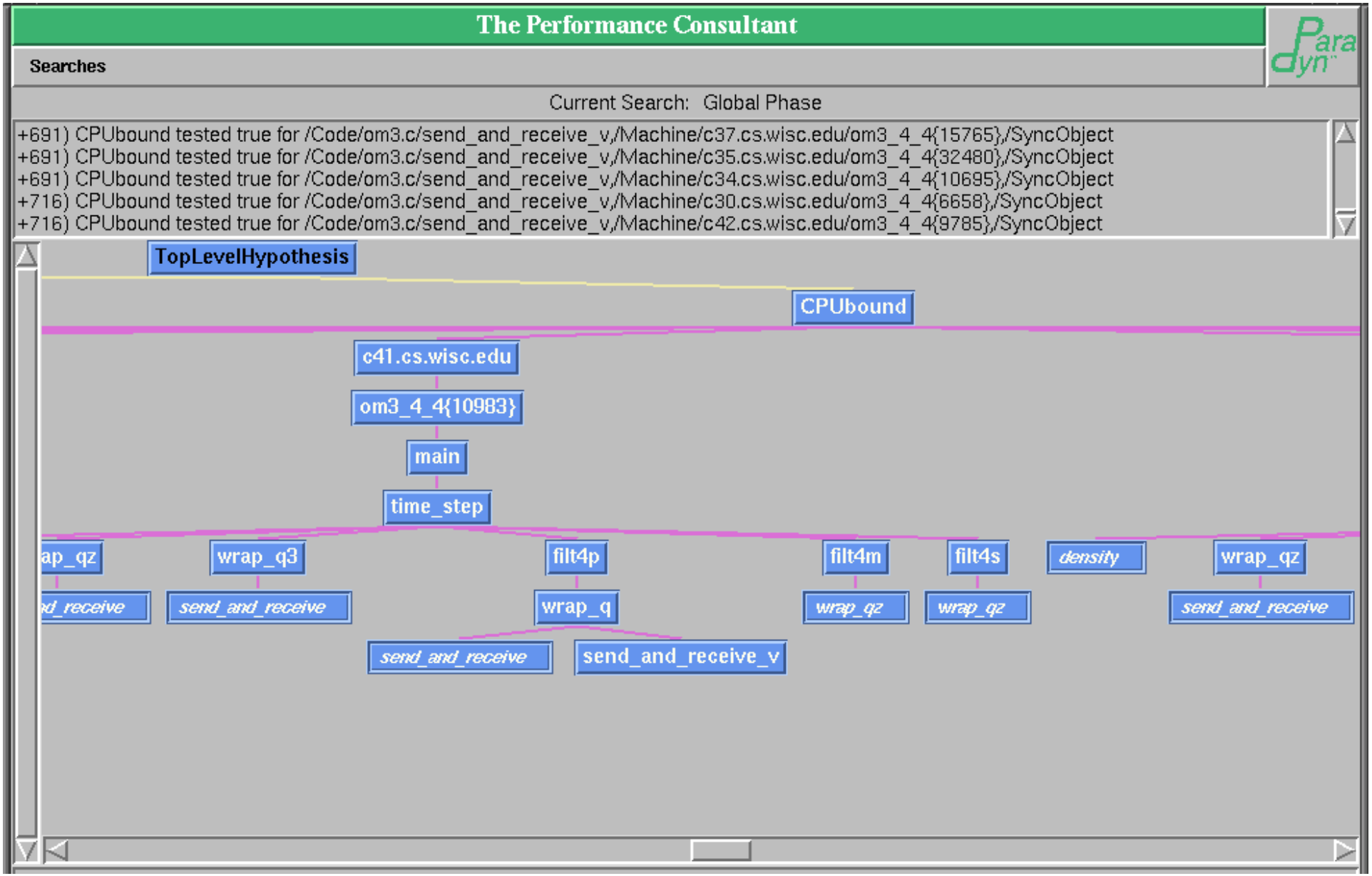


Forschungszentrum Jülich

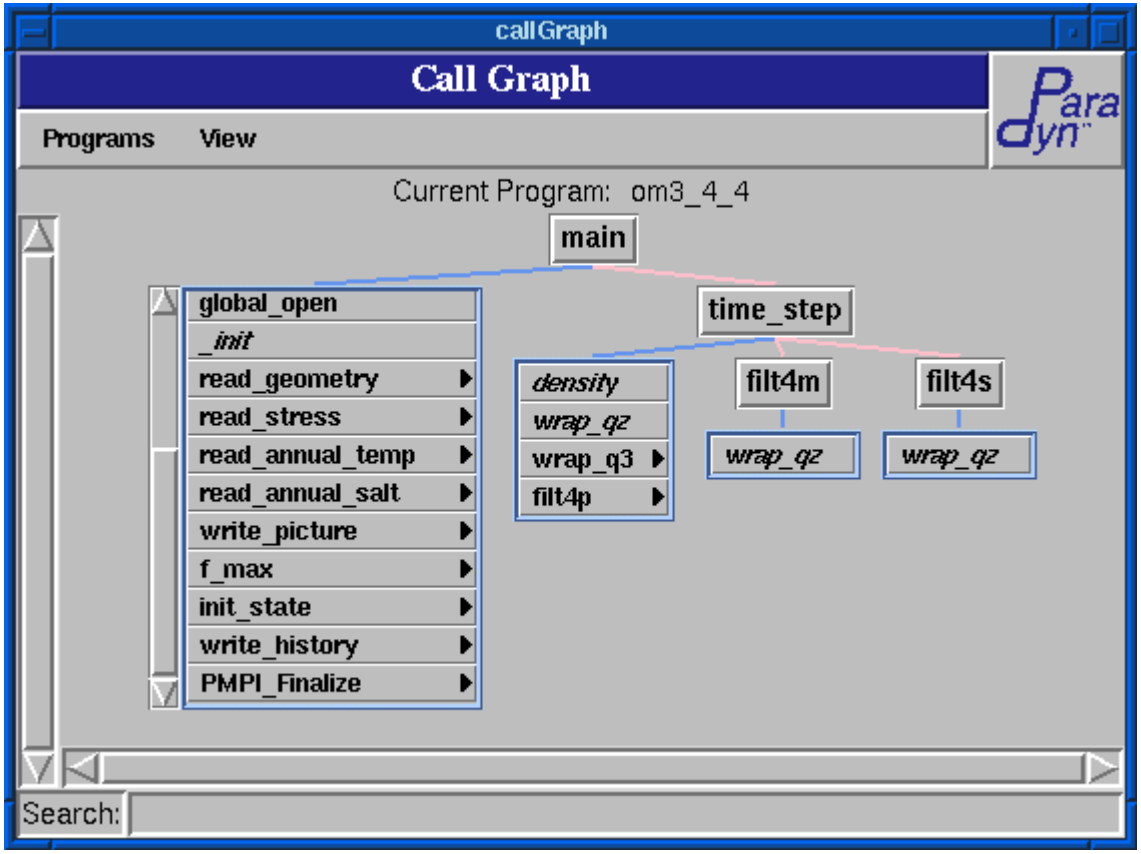
Performance Consultant search complete



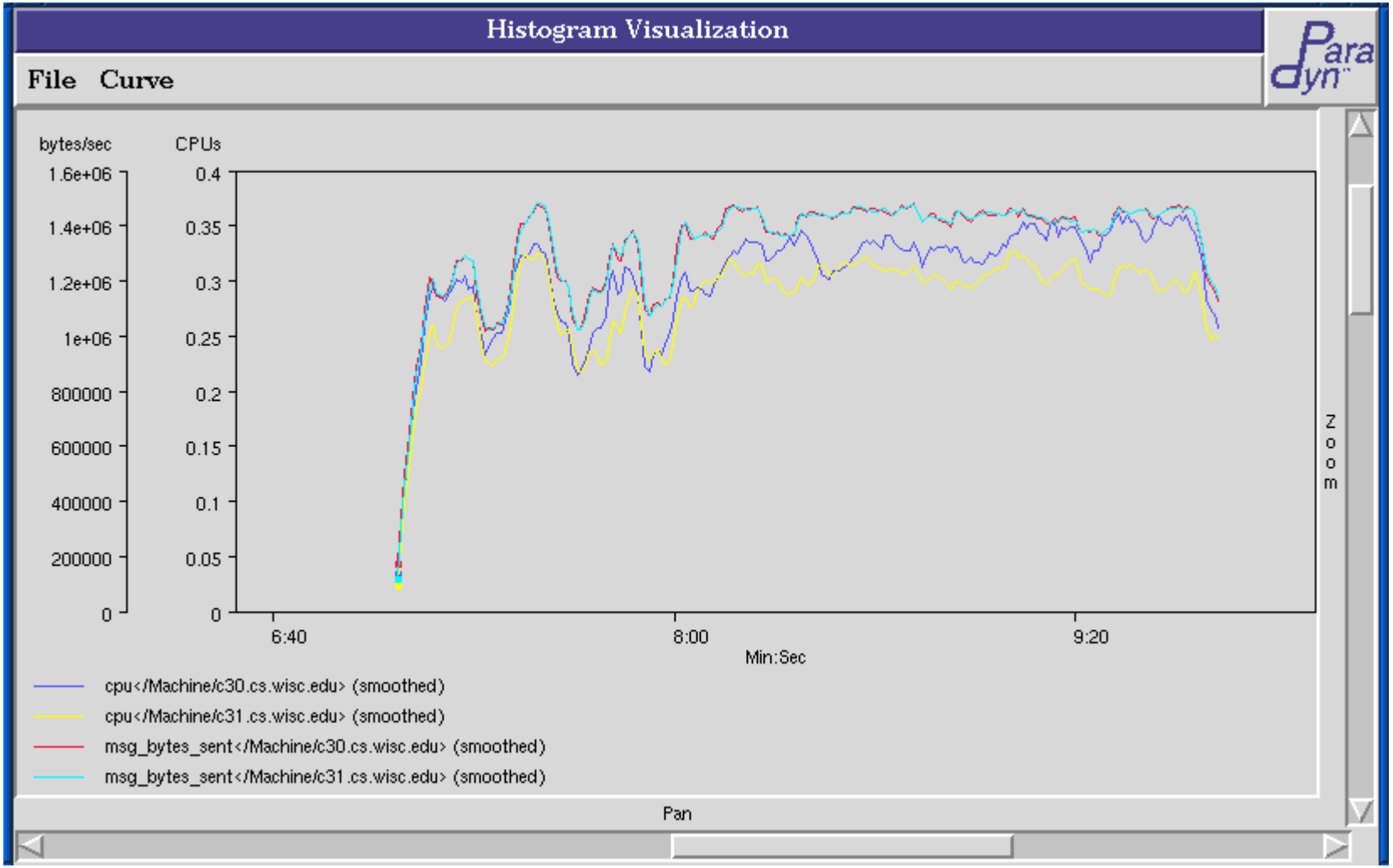
Performance Consultant complete (compact view)



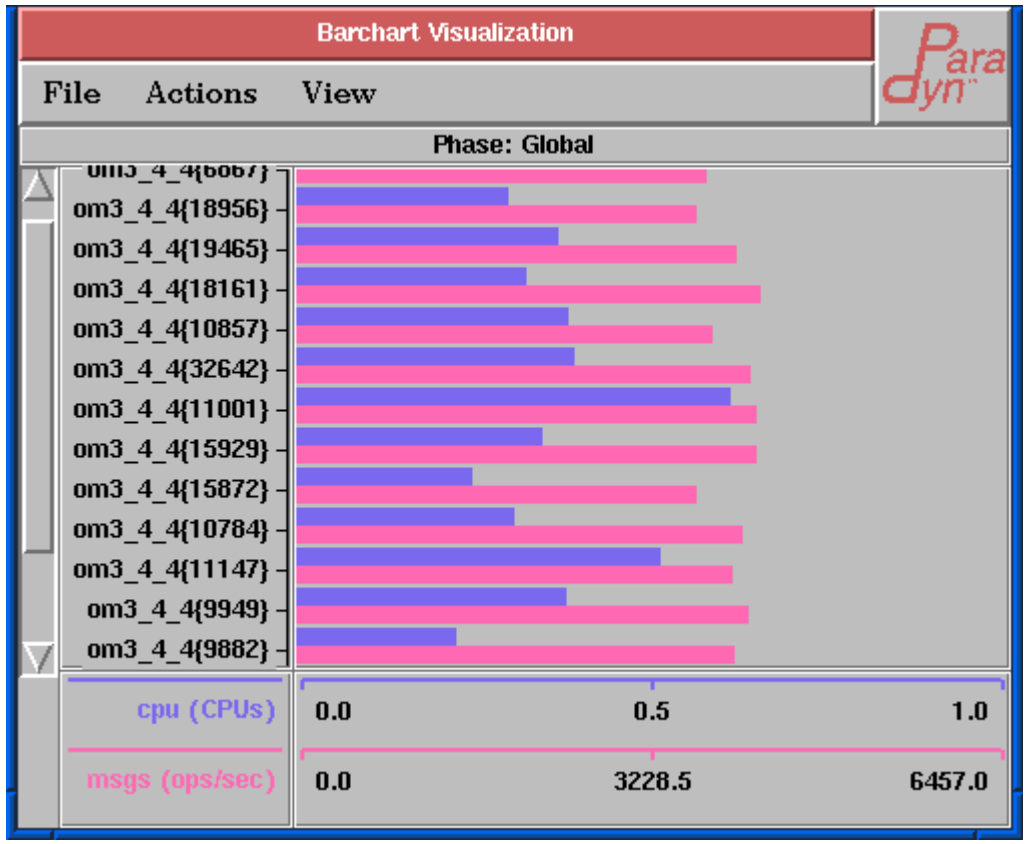
Call Graph Display



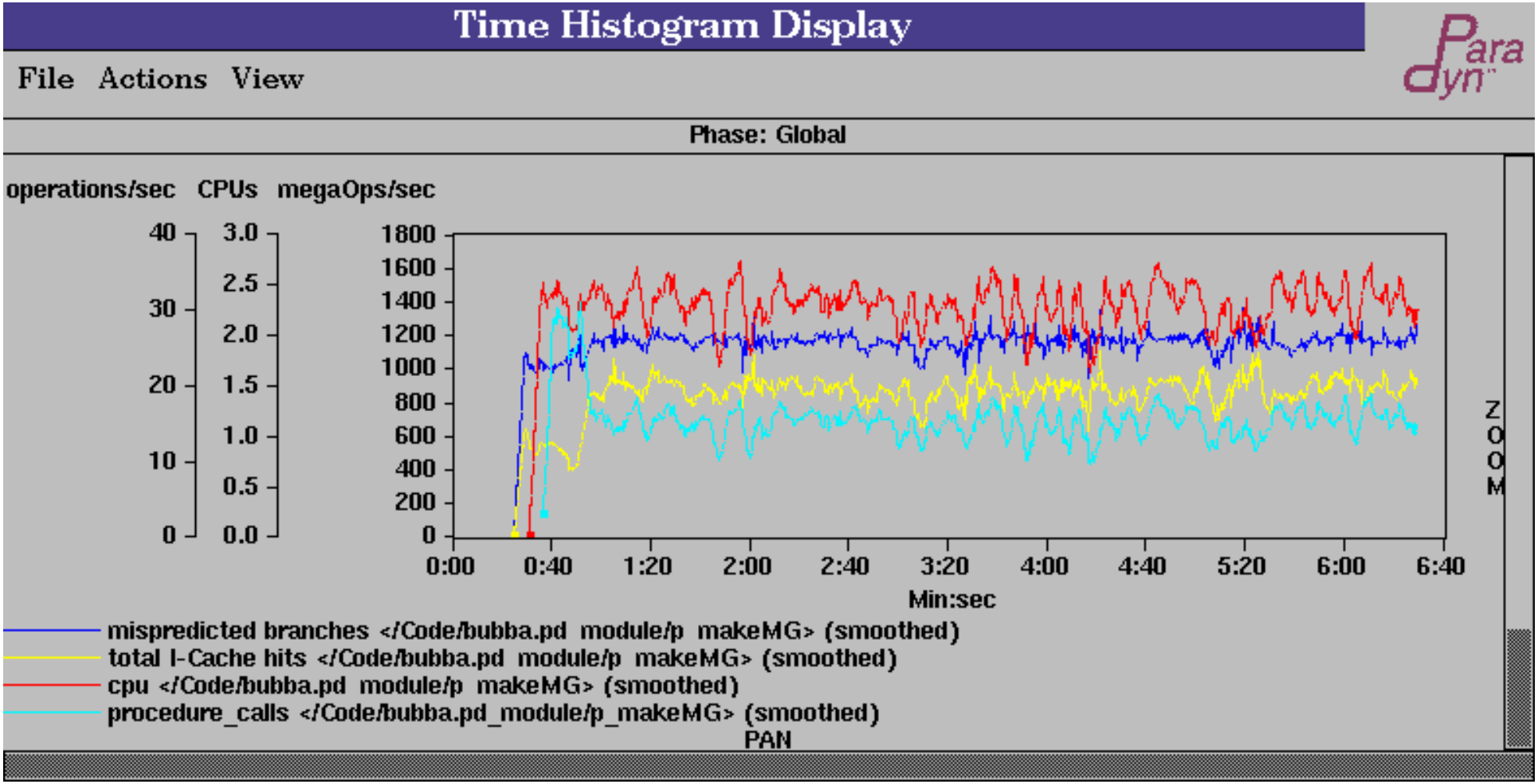
Time Histogram (Time Series) Display



Bar Chart Display



UltraSPARC Hardware Performance Counters



Controlling Instrumentation Cost

"What is the overhead of instrumentation?"

translates to:

"How many hypotheses can be evaluated at once?"

Predicted Cost:
- Known primitive cost
- Estimate frequency
- User-defined threshold

Observed Cost:
- Calculates actual cost
- Meta-instrumentation
- Reports to Performance Consultant

Paradyn Base Technology

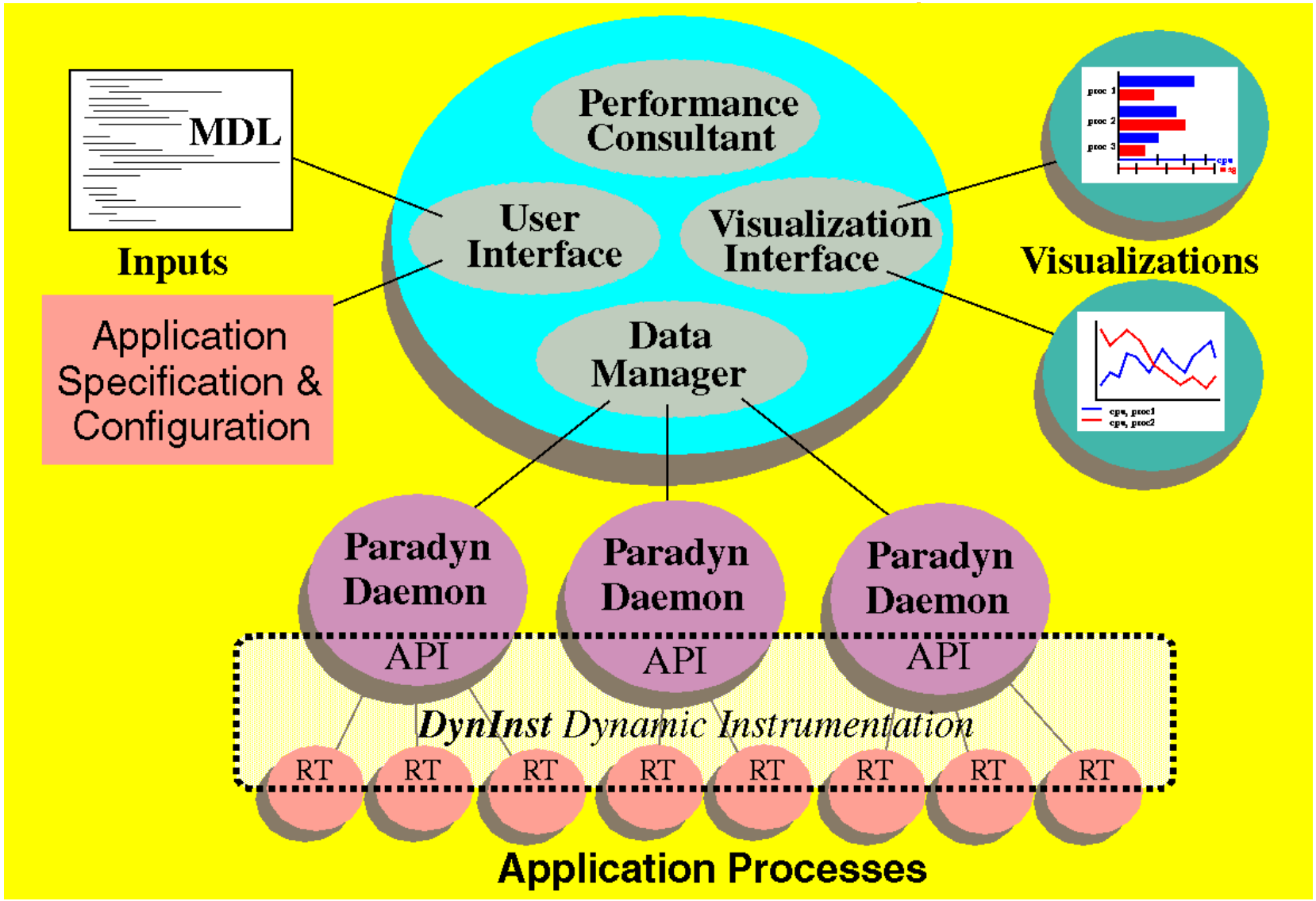
Dynamic Instrumentation

- On-the-fly generation, insertion, removal and modification of instrumentation in the application program while it is running.

This provides:

- Expressive event/metric instrumentation specification
- Efficient scalable data collection/storage management
- Extensible execution/performance data presentation/visualization

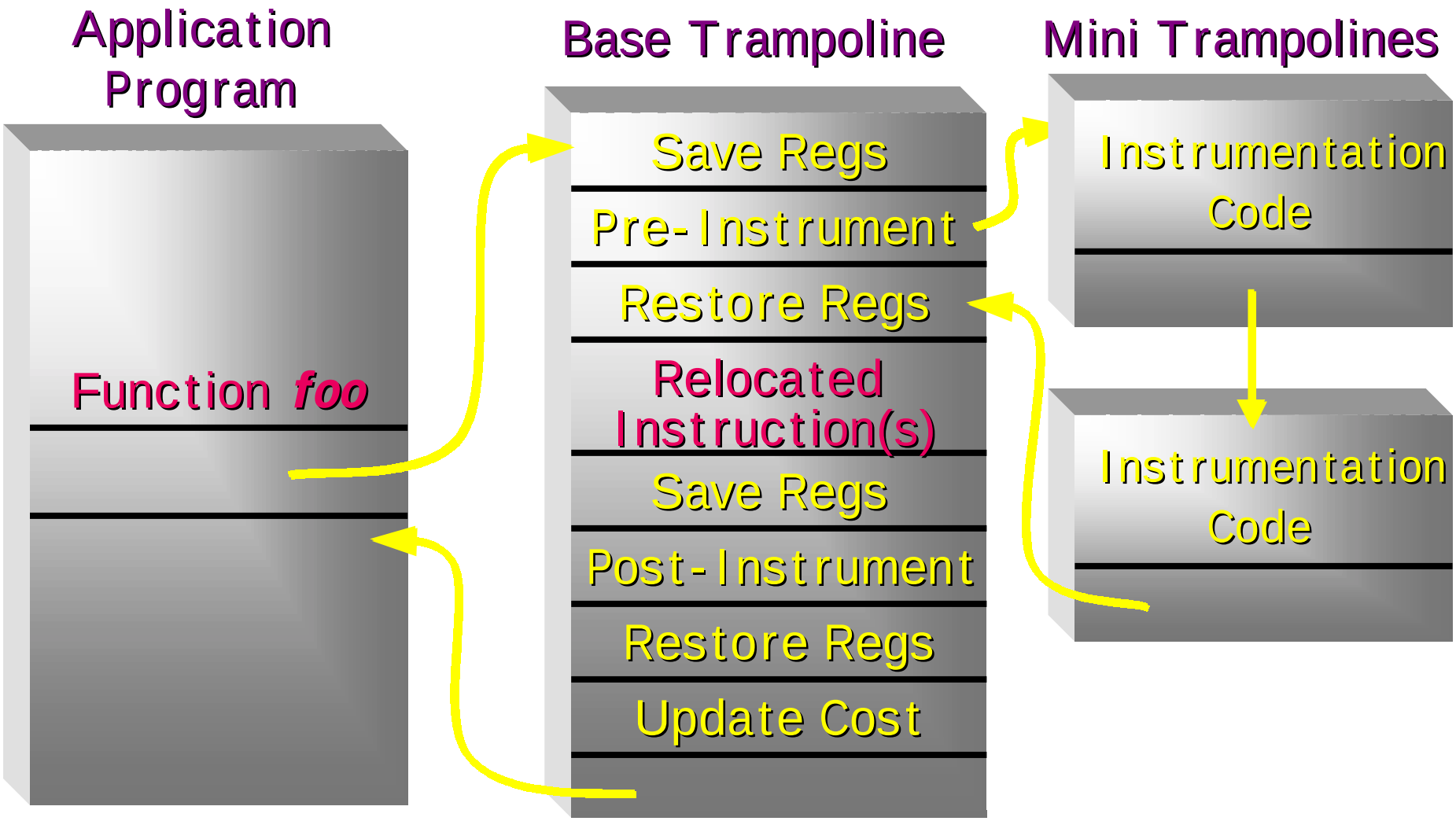
Paradyn Architecture



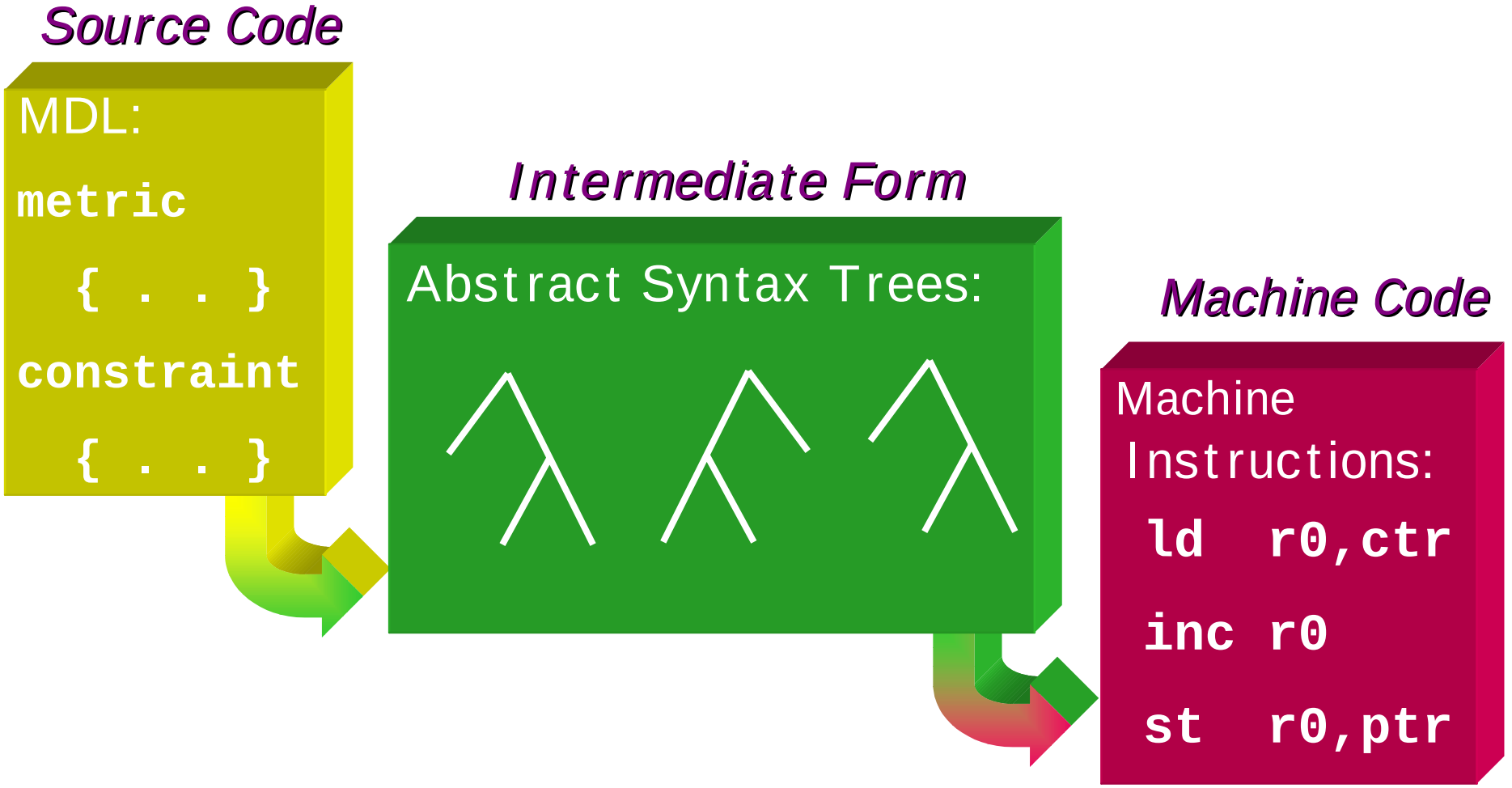
Dynamic Instrumentation

- For each subject program executable object/process:
  - Parse for 'interesting' symbols: functions, variables, etc.
  - Determine potential instrumentation points: *func_entry*, *func_exit*s, *callsite*s, ...
- At each chosen instrumentation point:
  - Relocate 'footprint' instruction(s) to instrumentation framework ("base trampoline") where execution state is reconstructed
  - De-optimize/re-write/relocate function, as necessary
  - Replace footprint instruction(s) with branch to base trampoline
- For each 'snippet' of executable instrumentation:
  - Synthesize it from abstract metric specification based on primitives and predicates
  - Embed instrumentation snippet in its own mini-trampoline
  - Daisy-chain snippet mini-trampolines from the base trampoline

Instrumentation Patching

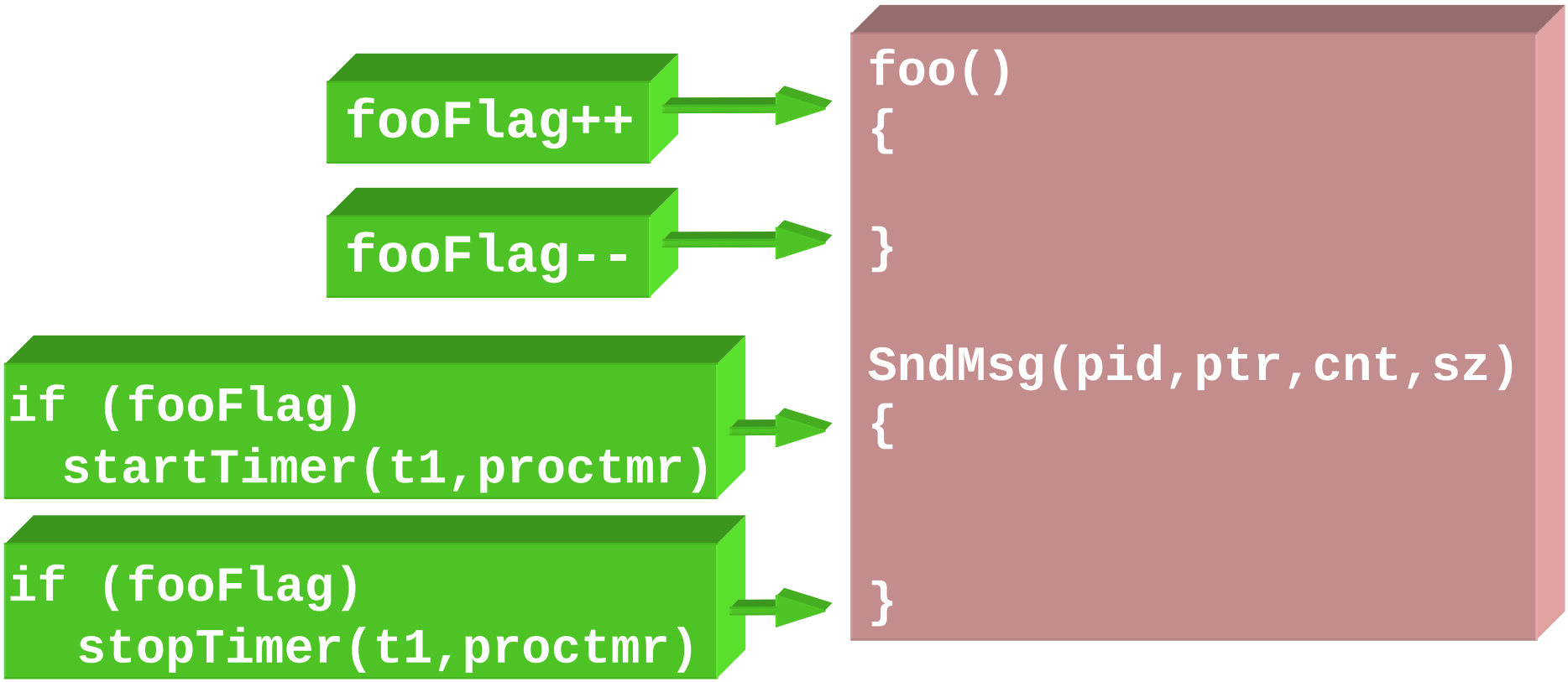


Compiling for Dynamic Instrumentation



Basic Instrumentation Operations

- Points: places to insert instrumentation
- Snippets: code that gets inserted



Dynamic Instrumentation Benefits

- Does not require recompiling or relinking
  - Saves time: compile and link times are significant in real systems.
  - Can profile without the source code (e.g., proprietary libraries).
  - Can profile without linking (relinking is not always possible).
- Instrument optimized code and threaded applications
  - De-optimize code as required on-the-fly
- Can monitor still-running programs (such as database servers)
  - Production systems, embedded systems.
  - Systems with complex start-up procedures.
- Only instrument what's needed, when it's needed
  - No hidden cost of latent instrumentation.
  - No *a priori* restrictions on the granularity of instrumentation.

Enables "one pass" performance sessions.

Dynamic Instrumentation Benefits (cont'd)

- Anything in the application's address space can become a performance measure
  - Application metrics: transactions/second, commits/second.
  - OS metrics: page faults, context switches, disk I/O operations.
  - Hardware metrics: cycle & instruction counters, miss rates, network statistics.
  - User-level protocol metrics: consistency messages, cache activity.
- Metrics defined with the ***Metric Description Language*** (MDL)
  - Neither performance tool nor application need be modified.
  - Define metrics once for each environment.
- Can dynamically monitor and control instrumentation overhead
  - Allows programmer to *monitor* intrusiveness.
  - Allows programmer to *control* intrusiveness.
  - Allows Performance Consultant to work efficiently.

Dynamic Instrumentation Challenges

- Finding instrumentation points (function entry, exits, call sites)
  - Procedure exits are often the toughest.
- Finding space for jump to trampoline
  - Long jumps (2-5 words or 5 bytes).
  - Short code sequences.
  - Small functions.
  - One byte instructions.
- Compiler optimizations (instrumenting optimized code)
  - No explicit stack frame (leaf functions).
  - Tight instrumentation points.
  - Data in code space (e.g., jump tables).
  - De-optimize code on the fly (e.g., tail calls).
- Threaded code

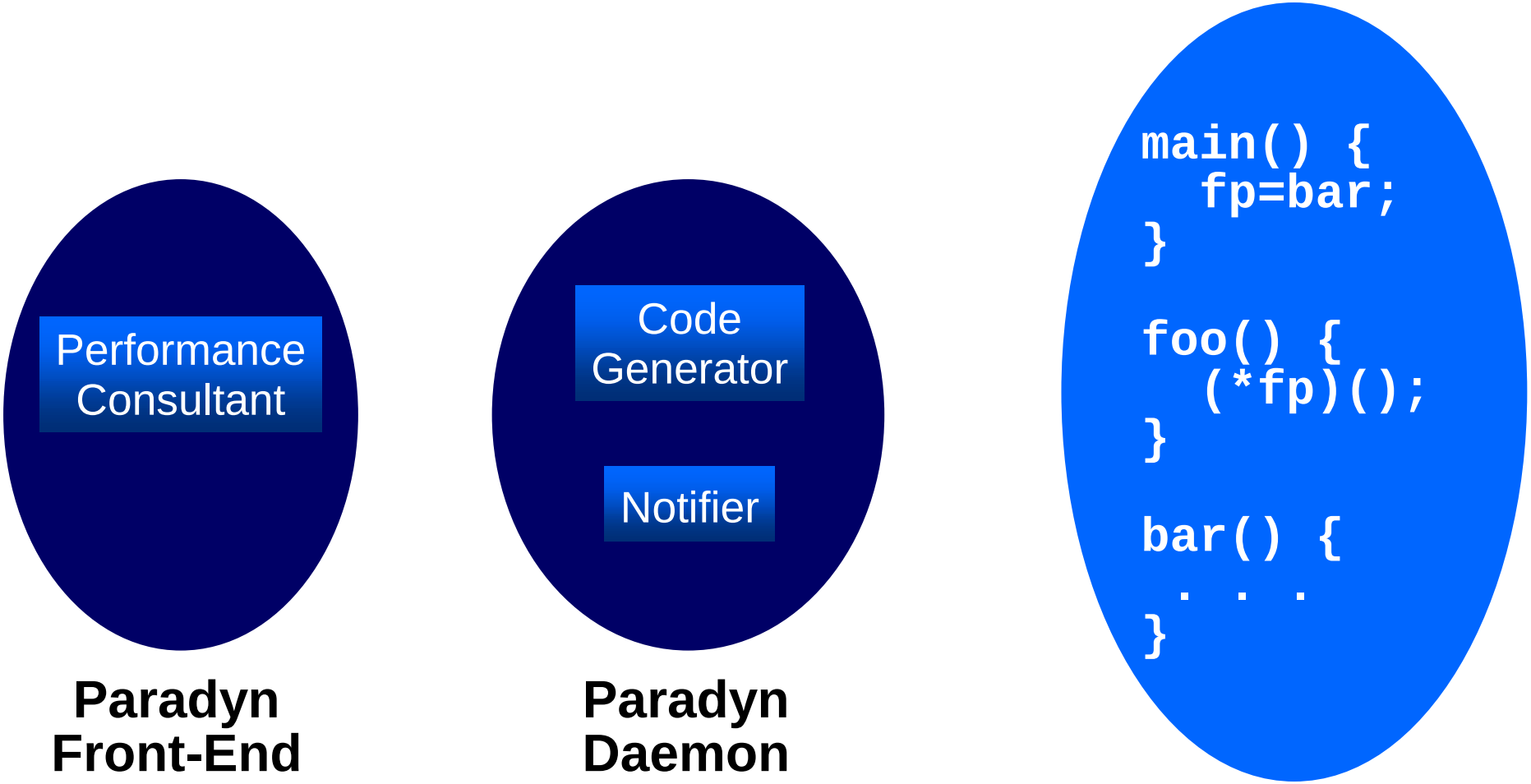
Scalable Data Collection & Storage

- Metrics are time-varying functions that quantify execution behavior (e.g., CPU utilization, messages/second, blocking time, file reads/second, ...)
- Instrumentation code in each application process calculates its own metrics (with counters and timers).

- "Every so often" each associated Paradyn daemon samples the performance data from a shared region.
- Accuracy of performance metrics is not affected by sampling rate ... only how often updated values are provided.

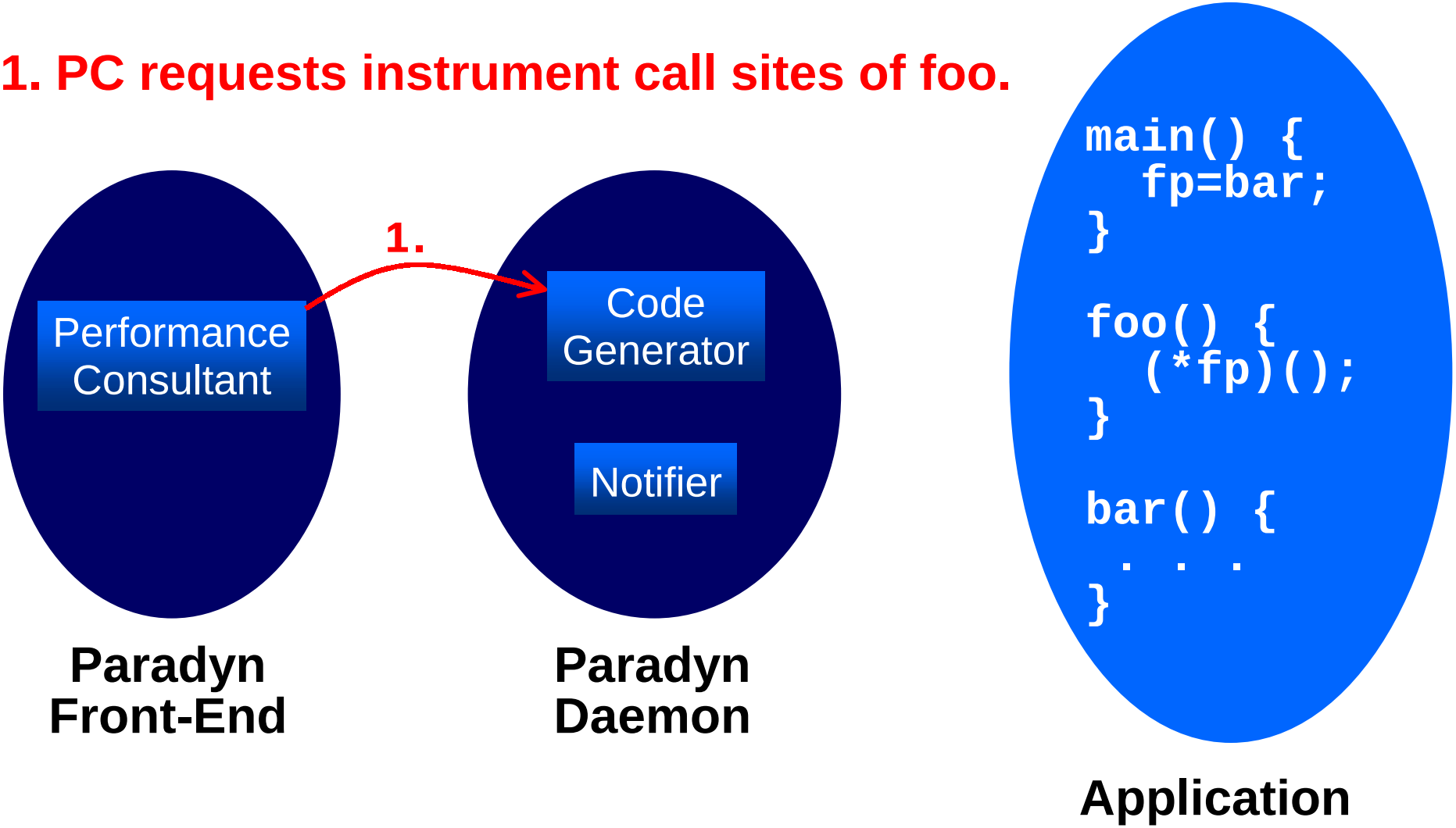
- Fixed-size discrete structure used to store time-varying data.
  - Each bucket holds the metric value for a time interval
  - When histogram is full, interval size doubled and data folded

Dynamic Call Site Instrumentation



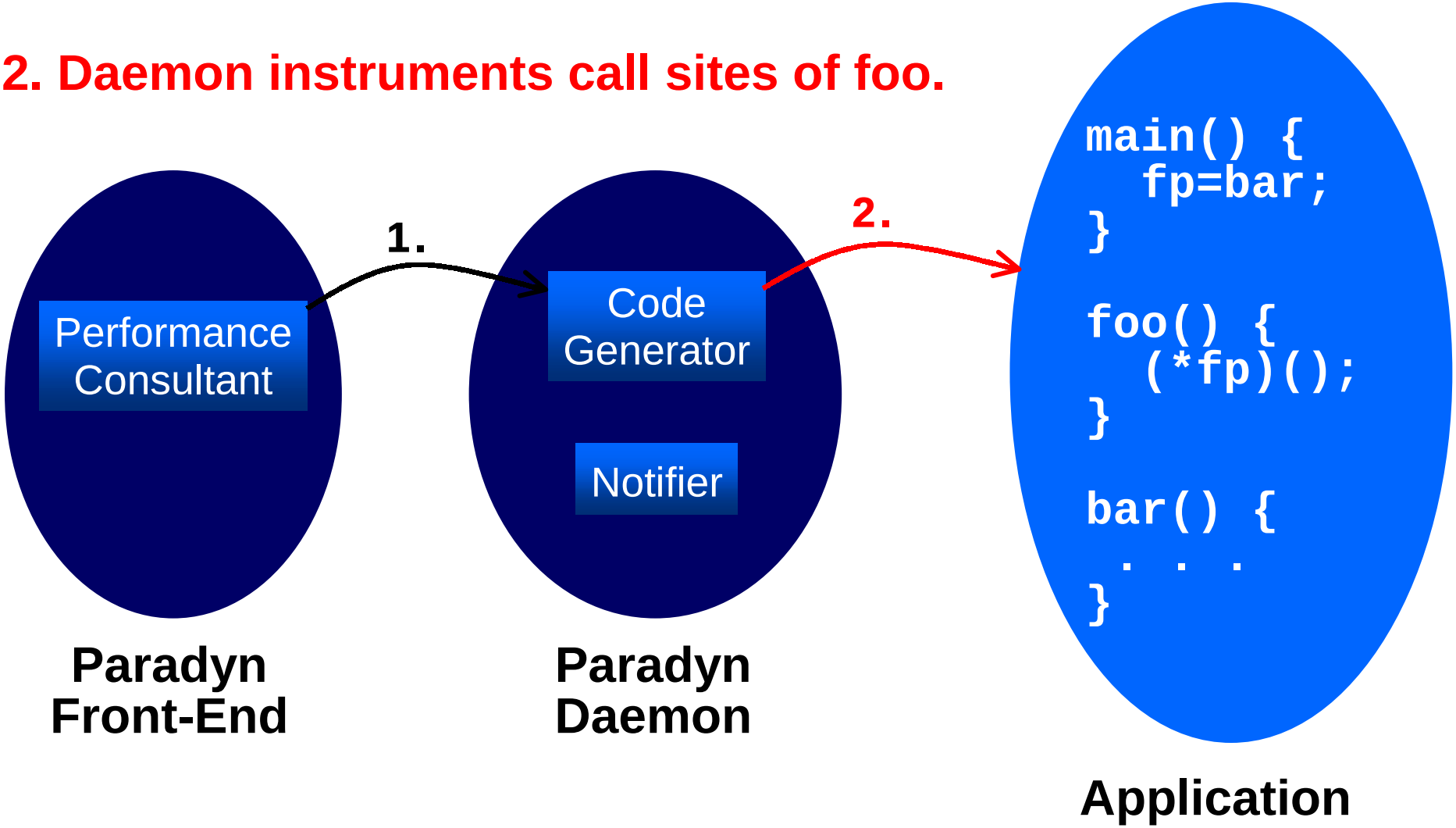
Dynamic Call Site Instrumentation

1. PC requests instrument call sites of foo.



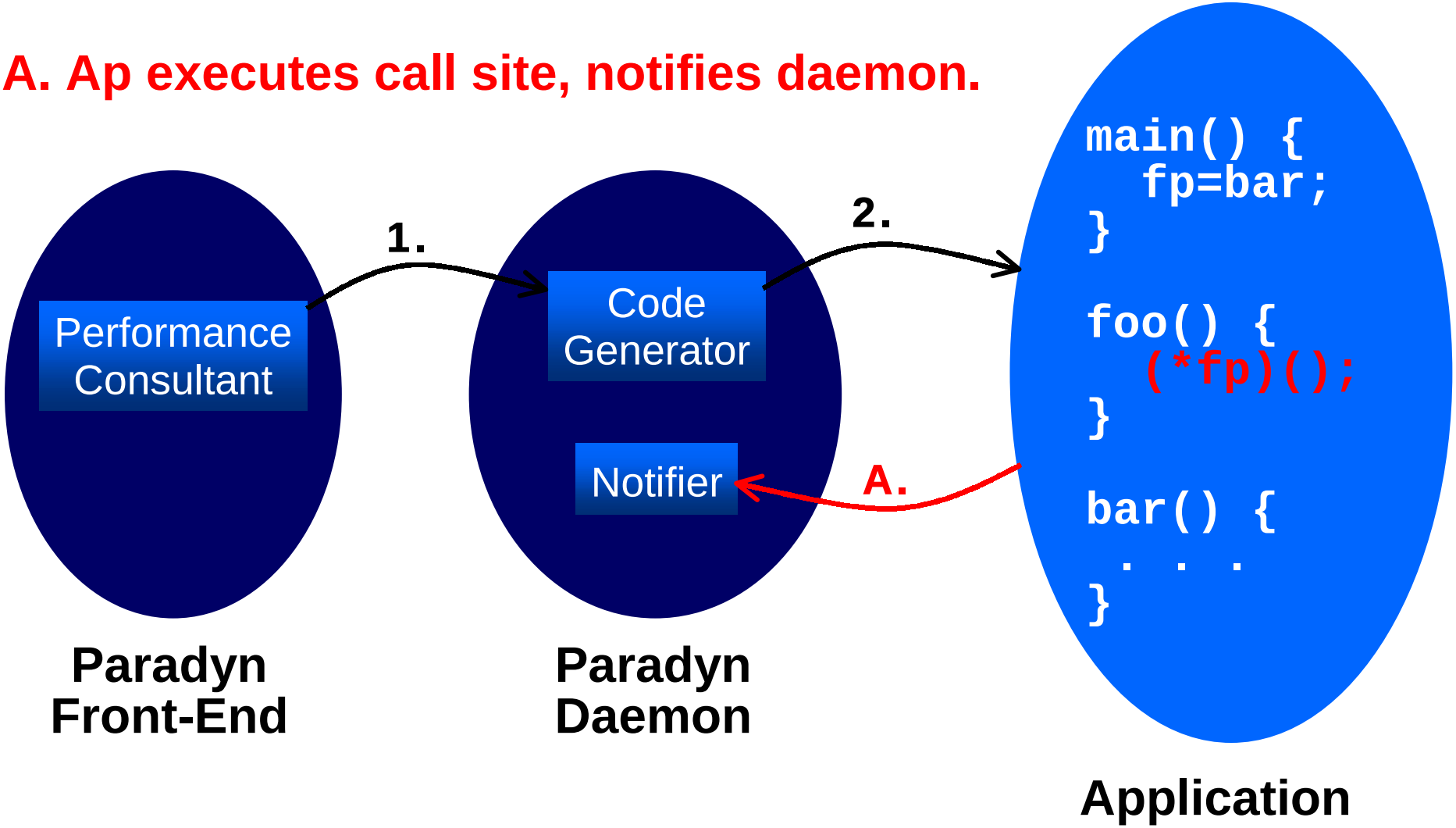
Dynamic Call Site Instrumentation

2. Daemon instruments call sites of foo.



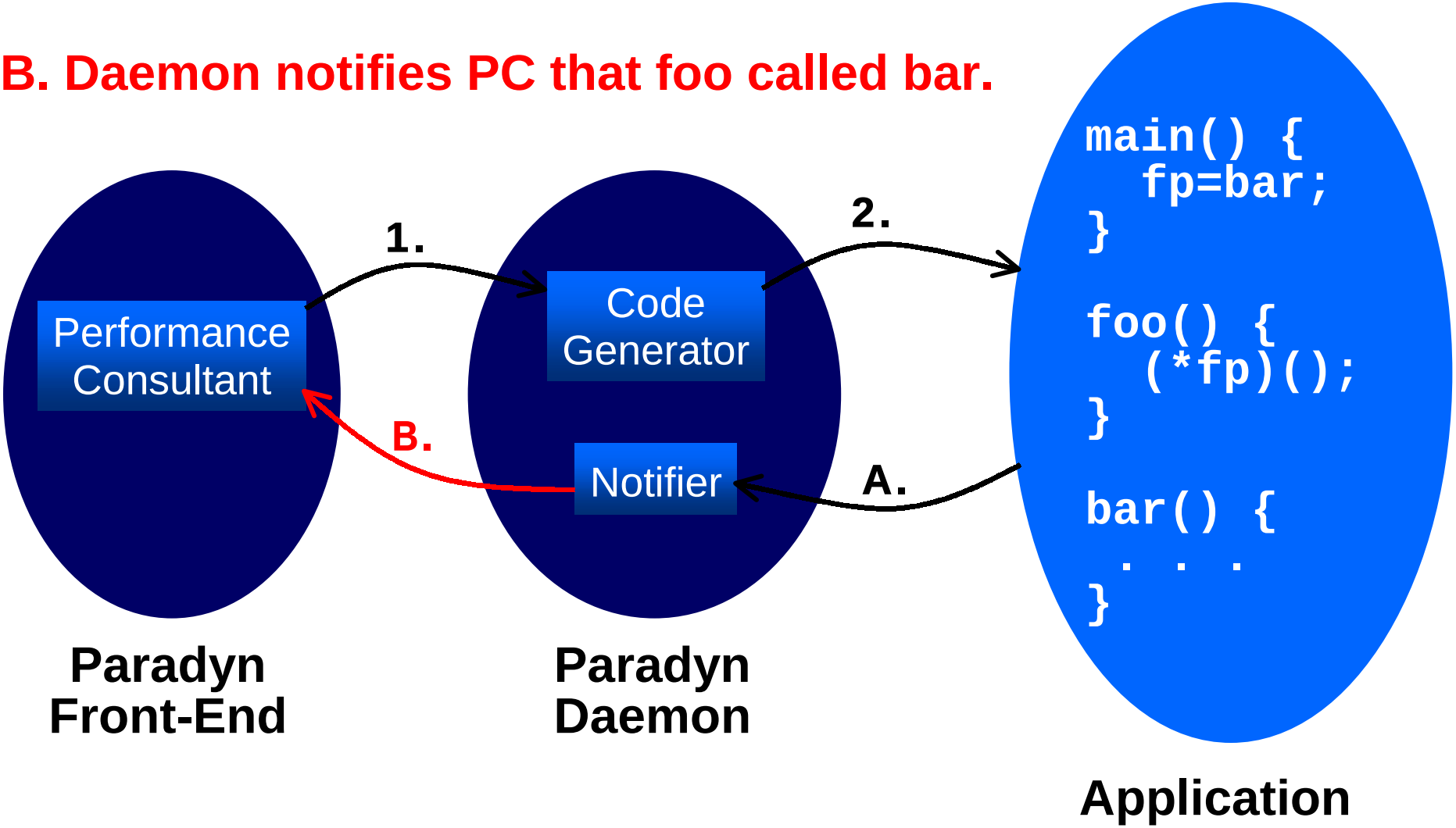
Dynamic Call Site Instrumentation

A. Ap executes call site, notifies daemon.



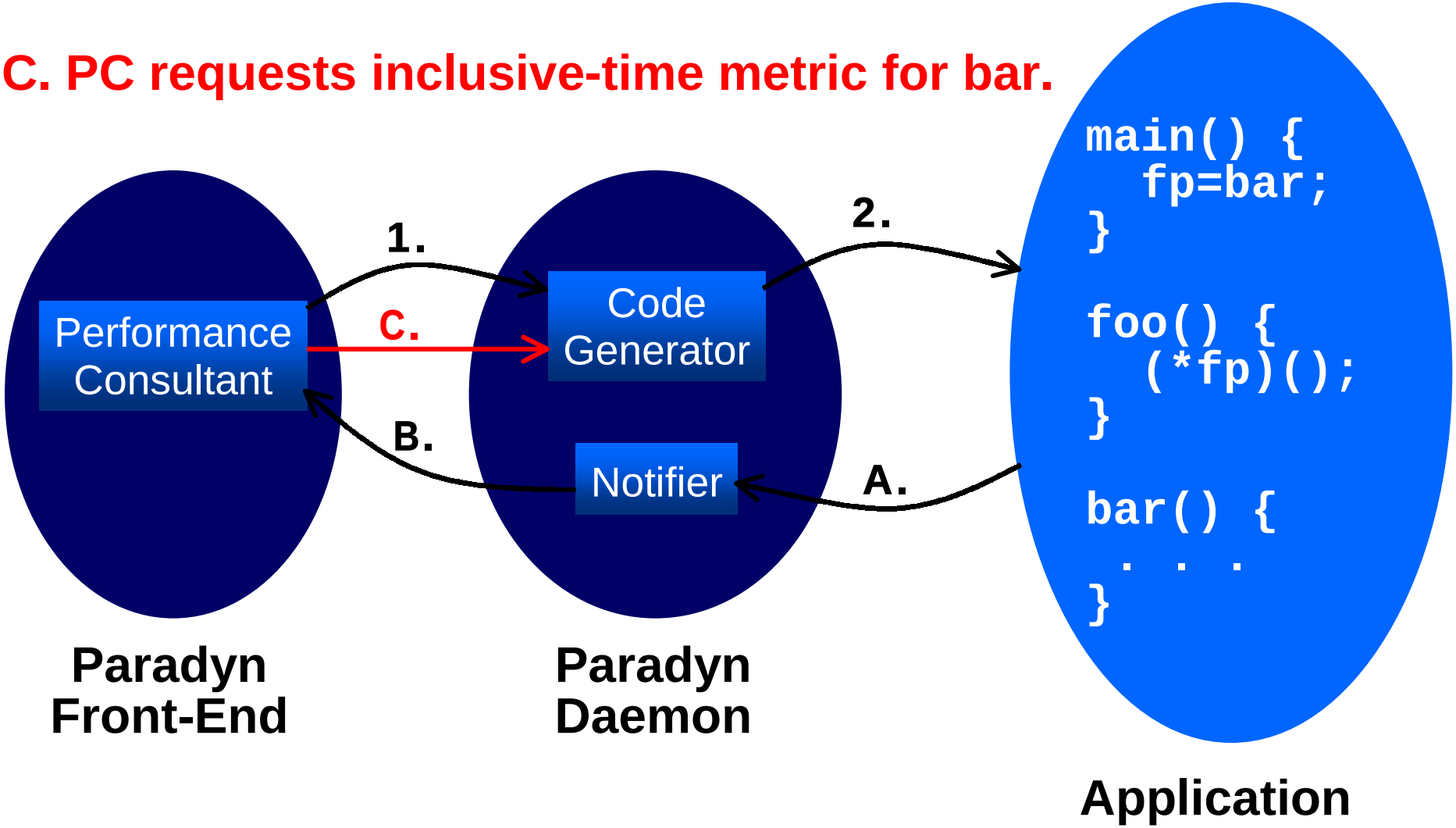
Dynamic Call Site Instrumentation

B. Daemon notifies PC that foo called bar.



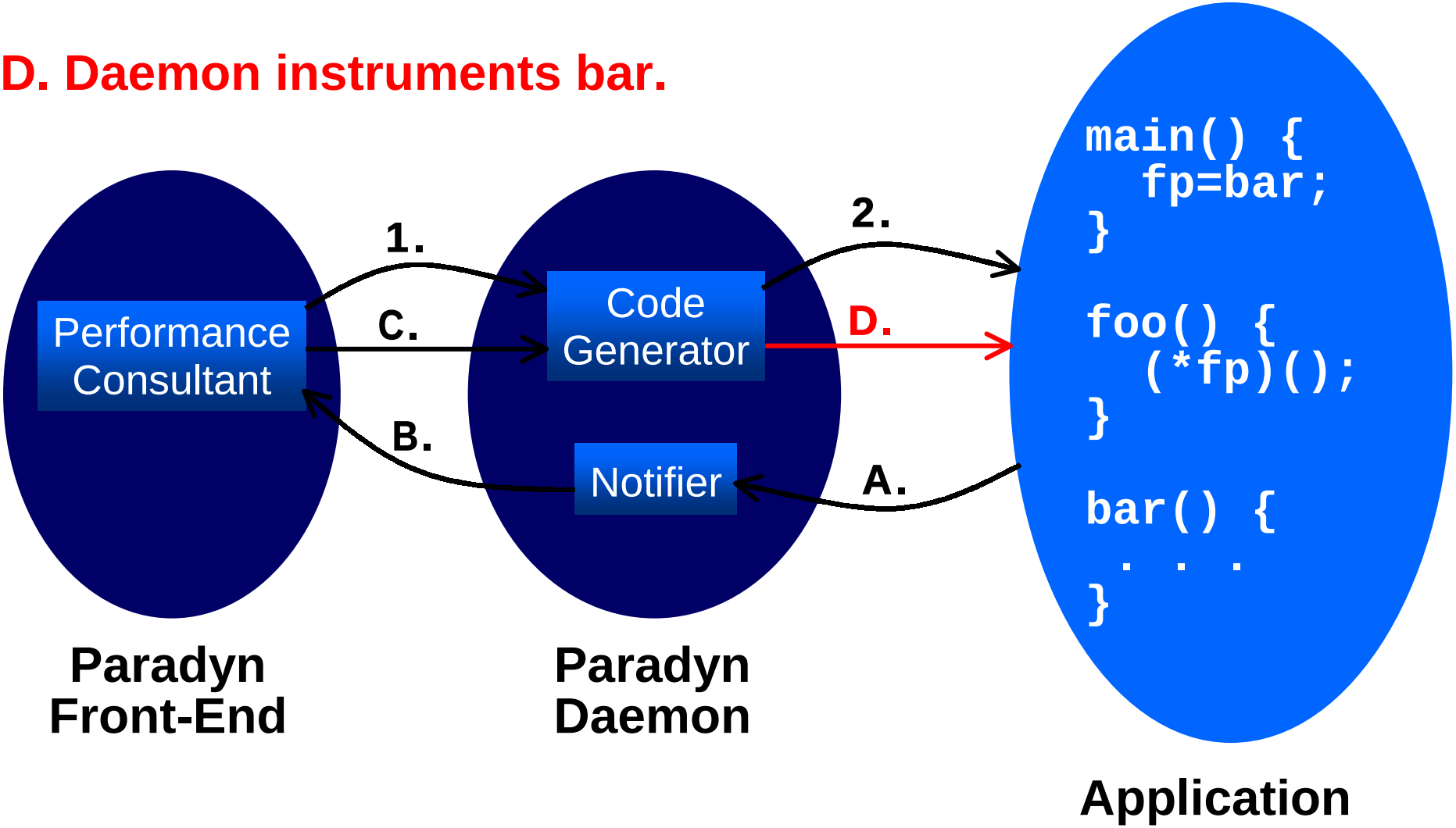
Dynamic Call Site Instrumentation

C. PC requests inclusive-time metric for bar.

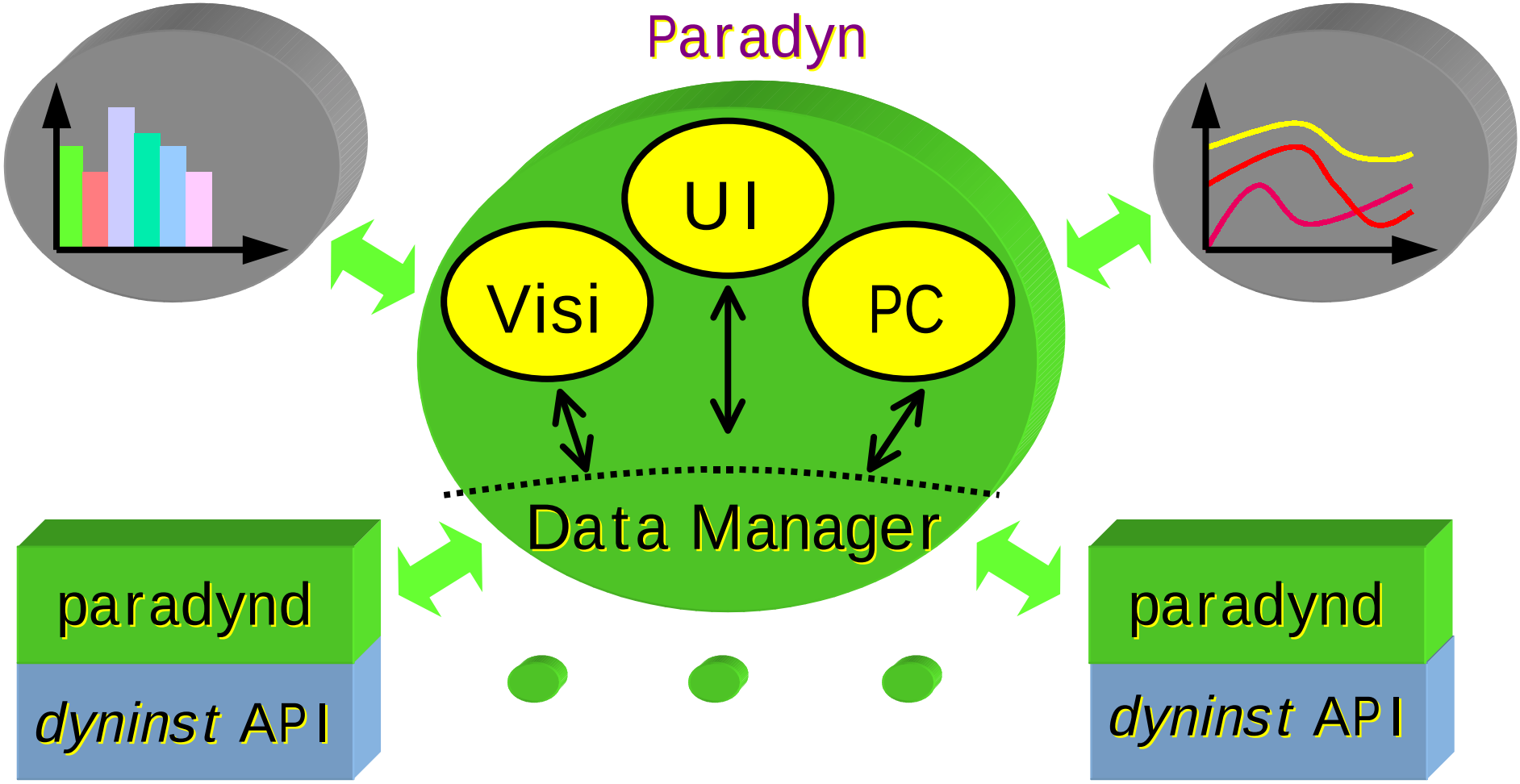


Dynamic Call Site Instrumentation

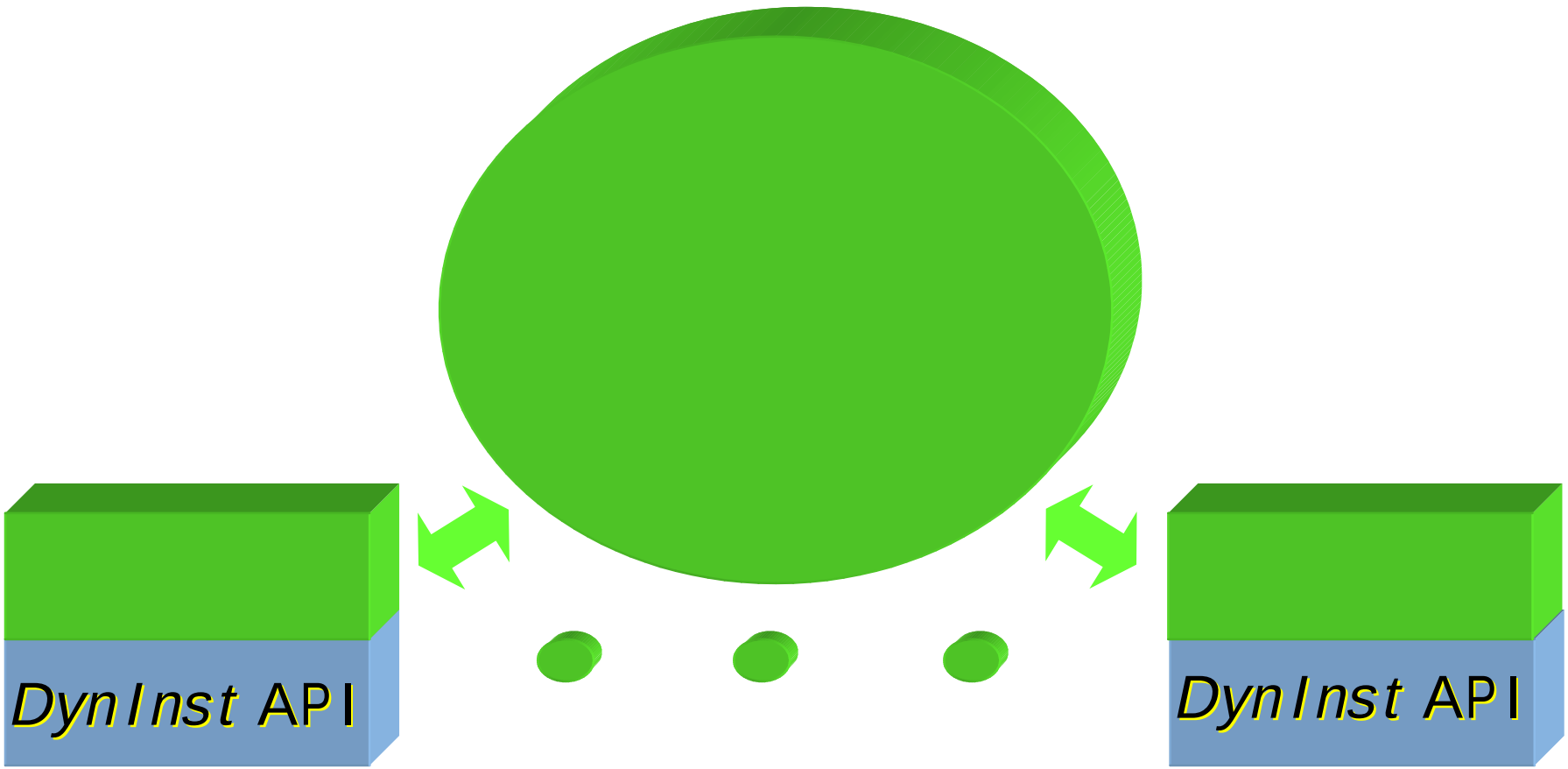
D. Daemon instruments bar.



DynInst API: A Common Substrate



Dyninst API: A Common Substrate



DynInst API

- Dynamic Instrumentation library and API
  - Basic substrate for building new tools.
  - Functions for control and interrogation of subject process execution
  - Runtime code generation and patching
  - Collaboration with University of Maryland.

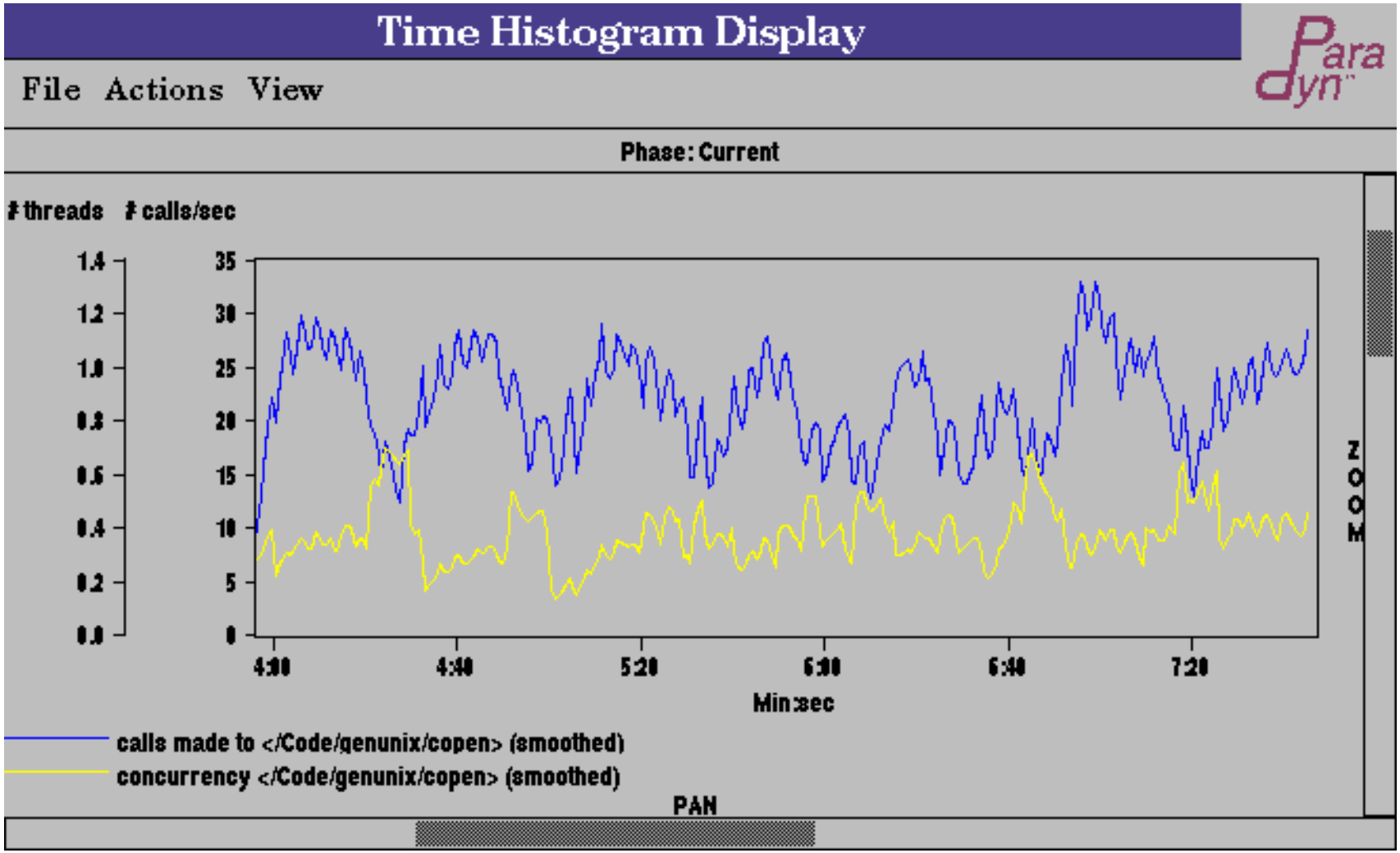
- Basis for IBM's ***Dynamic Probe Class Library*** (DPCL) architecture for secure multi-node, multi-tool support

- Open effort to standardize functionality required for tools:
  - correctness debugging, memory stewardship, computational steering, R/A/S, load balancing, checkpointing, coverage testing, etc.
  - Involves computer vendors, tools researchers and users

Current Research Areas

- Evolving Operating Systems (Ari,Vic)
  - Fine-grained instrumentation.
  - On-the-fly kernel profiling, debugging, testing.
  - Code that adapts to workload.
  - Dynamically customizable kernels.

Profile Any Part of the Kernel at Anytime



Current Research Areas

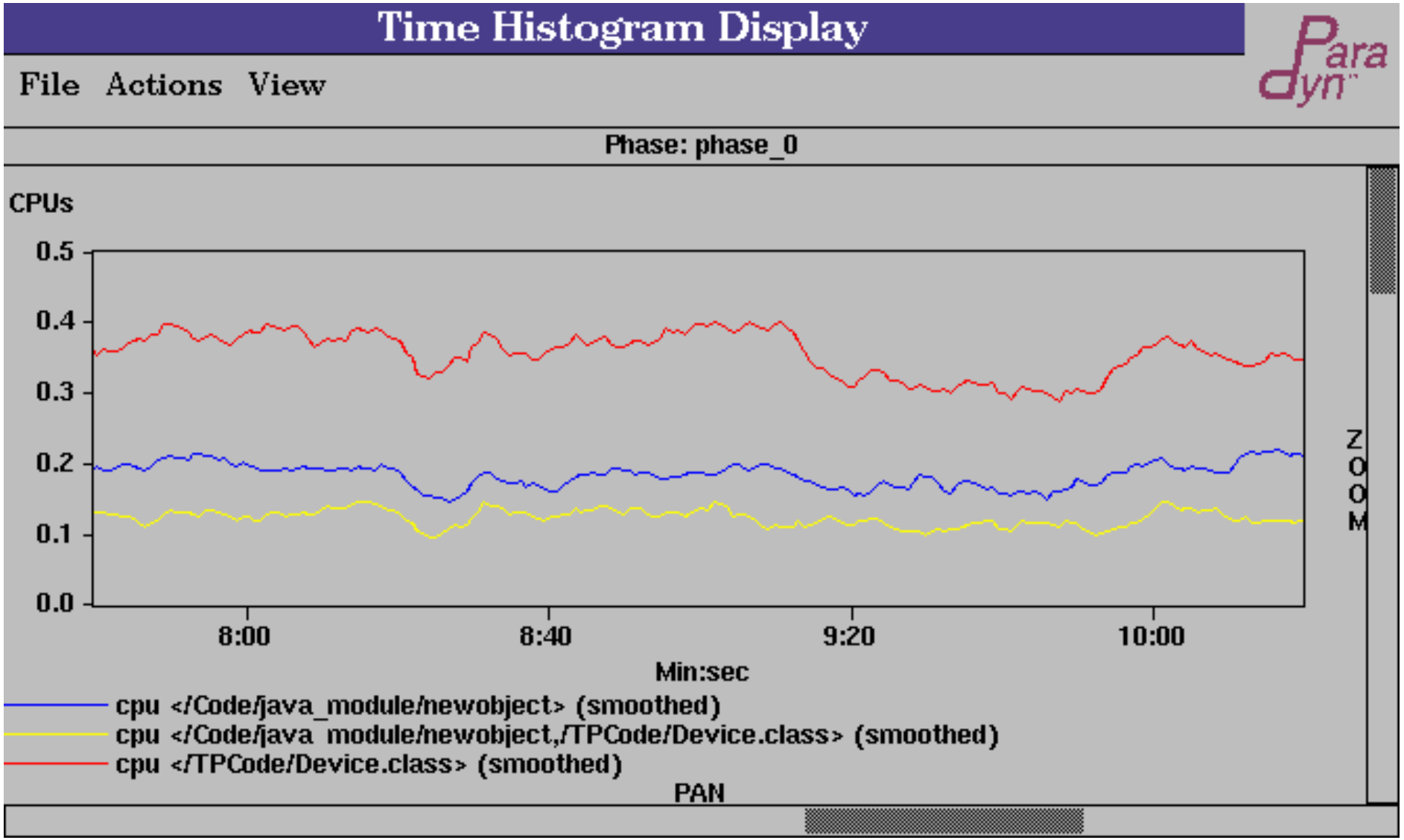
- Profiling Interpreted Code (Tia Newhall, Swarthmore)
  - Performance data in terms of virtual machine:

    CPU time for interpreter function *f*.
  - Performance data in terms of application:

    CPU for Java method *m*.
  - Interaction effects:

    CPU time for function *f* when interpreting method *m*.
  - Multiple execution forms: interpreted and native machine code, and translation costs.

Java Application and VM Profiling



Current Research Areas

- Experiment Management (Karen Karavanic, Portland State Univ.)
  - Performance data from multiple runs (huge multidimensional space).
  - Tuning, benchmarking, regression testing, etc.
  - Provides infrastructure for manipulation and management of execution data
  - Automatically compare data from multiple runs.
  - Use historical information to speed performance diagnosis.
  - A “laboratory notebook” for performance studies.

Current Research Areas

- Security Applications of DynInst

  Process Hijacking
  - Step 1: Submitting “vanilla” jobs to Condor resource management system.
  - Step 2: Grabbing a running process and submitting it to Condor:
    - Replace running process’ system call library.
    - Checkpoint new version of process.
    - Generate new “shadow” process.

  License Server Bypass
  - Use DynInst-based tools to analyze program’s communication and control flow.
  - Modify program to believe that it has valid license credentials.

How to Get a Copy of Paradyn

- Documentation: Installation Guide, Tutorial, Users Guide, Developers Guide, Visi/MDL/libthread Programmers Guides.
- Includes *DynInst* API, tests and Programmers Guide
- Free for research use.
- Runs on
  - Solaris (SPARC & x86)
  - Linux (x86)
  - Irix (MIPS)
  - Windows NT (x86)
  - AIX/SP2 (Power2/3)
  - Tru64 Unix (Alpha)

http://www.cs.wisc.edu/paradyn

paradyn@cs.wisc.edu

Paradyn/DynInst Developers

The Wisconsin & Maryland Team:

- Will Benton
- Drew Bernat
- Ning Li
- Alex Mirgorodskii
- Nilu Motiwala
- Nick Rasmussen
- Phil Roth
- Brandon Schendel
- Ari Tamches
- Zhichen Xu
- Vic Zandy

- Bryan Buck
- Jeff Hollingsworth
- Mustafa Tikir

Colleagues:

- Phil Mucci (Univ. of Tennessee)
- Ted Hoover (IBM Poughkeepsie)
- Karen Karavanic (Portland State)
- Emilio Luque (UAB)
- John Kewley (CSCS)
- Jeff Brown (LANL)