Performance Observation

Sameer Shende and Allen D. Malony
{sameer,malony} @ cs.uoregon.edu

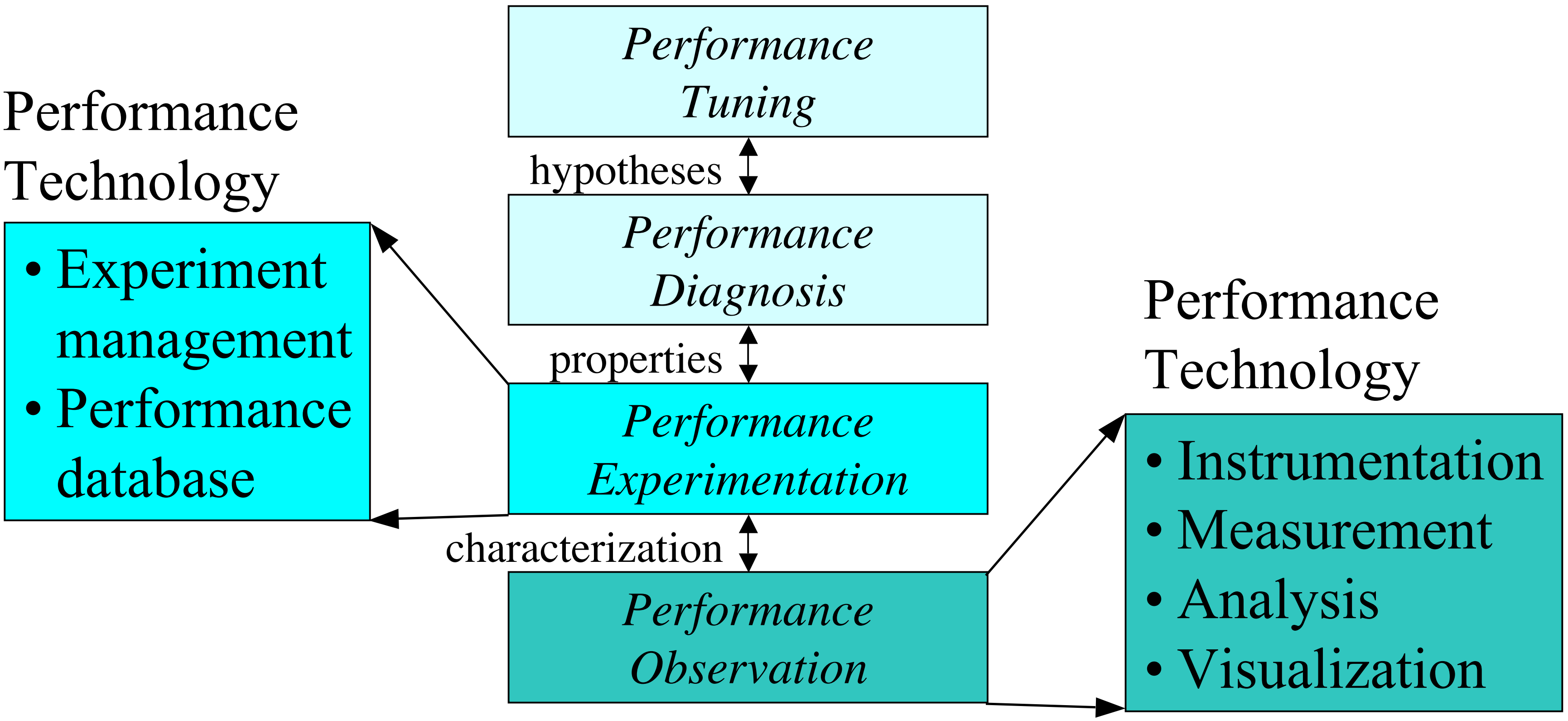
UNIVERSITY
OF OREGON

Outline

- Motivation
- Introduction to TAU
- Optimizing instrumentation: approaches
- Perturbation compensation
- Conclusion

Research Motivation

- Tools for performance problem solving
  - Empirical-based performance optimization process
  - Performance technology concerns



Performance Tuning

hypotheses

Performance Diagnosis

properties

Performance Experimentation

characterization

Performance Observation

Performance Technology

- Experiment management
- Performance database

Performance Technology

- Instrumentation
- Measurement
- Analysis
- Visualization

TAU Performance System

- Tuning and Analysis Utilities (11+ year project effort)
- *Performance system framework* for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
  - nodes / contexts / threads
  - Multi-level: system / software / parallelism
  - Measurement and analysis abstraction
- *Integrated toolkit* for performance instrumentation, measurement, analysis, and visualization
  - Portable performance profiling and tracing facility
  - Open software approach with technology integration
- University of Oregon , Forschungszentrum Jülich, LANL

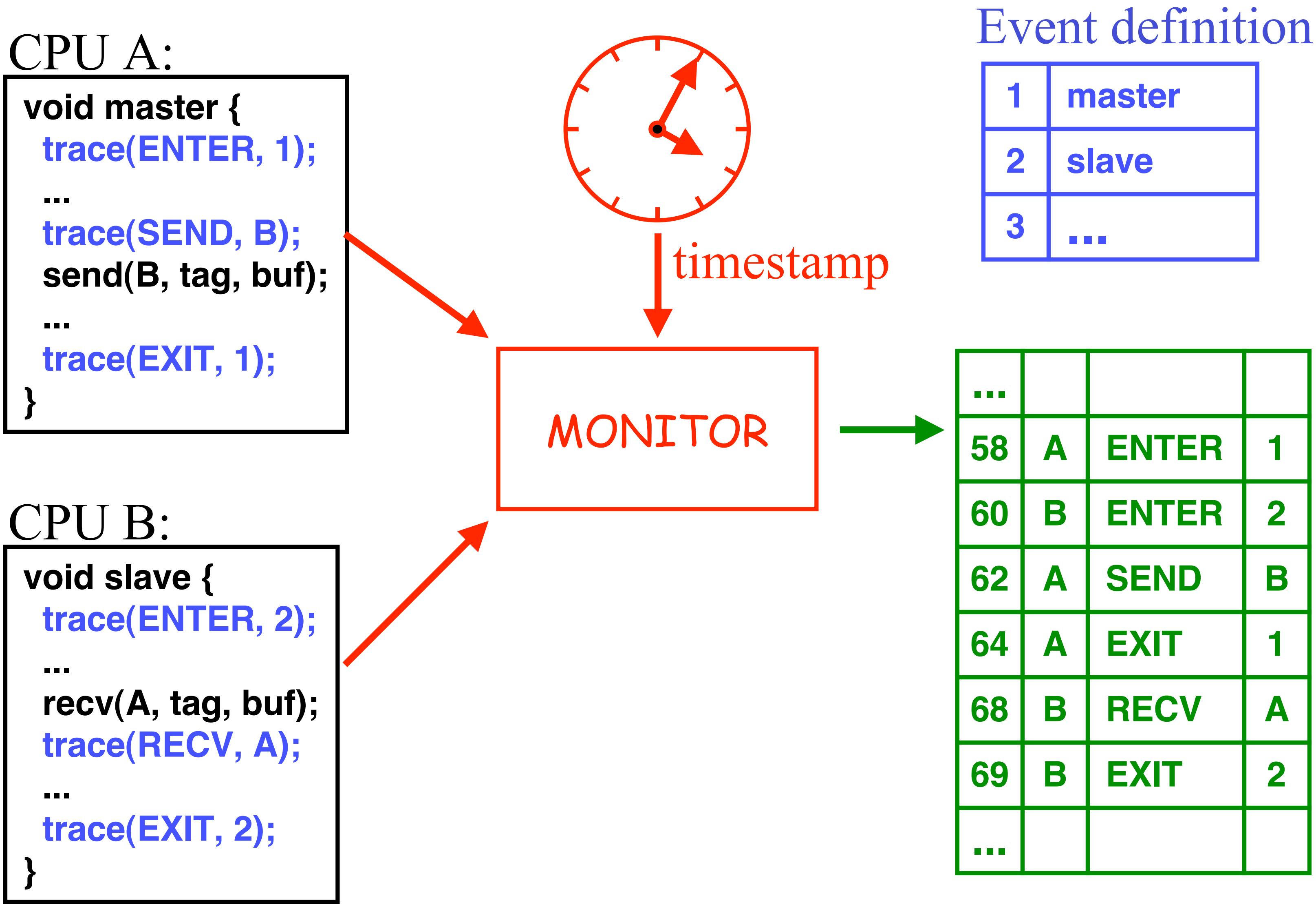
Definitions – Profiling

- Profiling
  - Recording of summary information during execution
    - inclusive, exclusive time, # calls, hardware statistics, …
  - Reflects performance behavior of program entities
    - functions, loops, basic blocks
    - user-defined "semantic" entities
  - Very good for low-cost performance assessment
  - Helps to expose performance bottlenecks and hotspots
  - Implemented through
    - sampling: periodic OS interrupts or hardware counter traps
    - instrumentation: direct insertion of measurement code

Definitions – Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
    - entering/exiting code region (function, loop, block, …)
    - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

Event Tracing: *Instrumentation*, *Monitor*, *Trace*



CPU A:

```
void master {
  trace(ENTER, 1);
  ...
  trace(SEND, B);
  send(B, tag, buf);
  ...
  trace(EXIT, 1);
}
```

CPU B:

```
void slave {
  trace(ENTER, 2);
  ...
  recv(A, tag, buf);
  trace(RECV, A);
  ...
  trace(EXIT, 2);
}
```

Event definition

1	master
2	slave
3	...

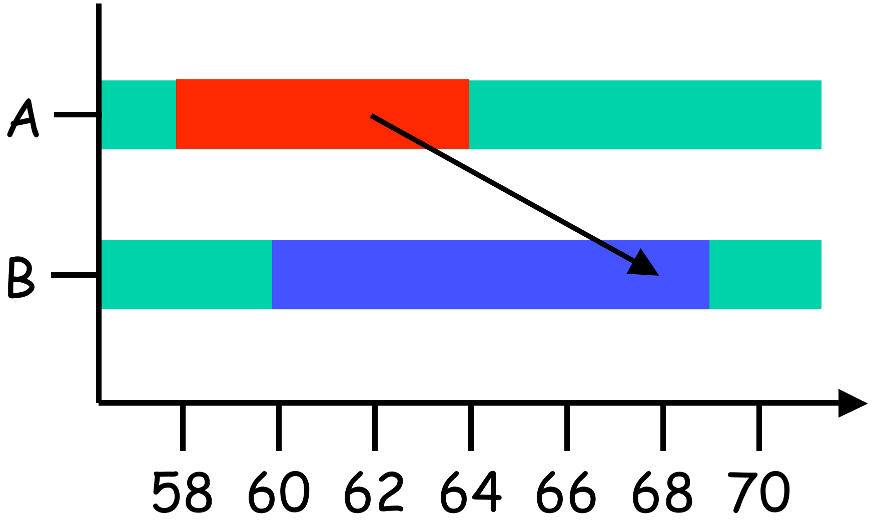
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Event Tracing: "Timeline" Visualization

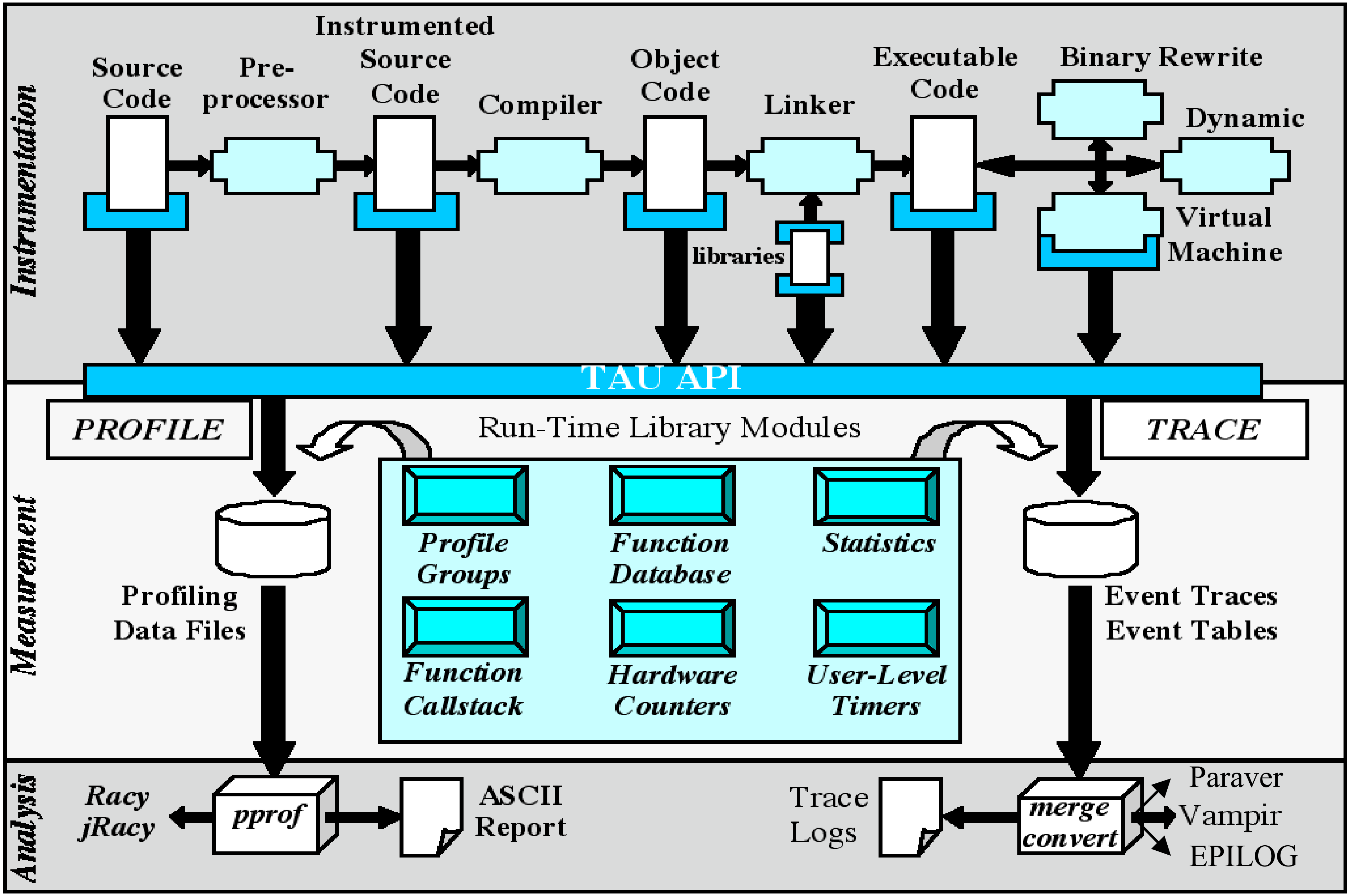
1	master
2	slave
3	...

🟩 main
🟥 master
🟦 slave

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



TAU Performance System Architecture



Instrumentation

Source Code

Pre-processor

Instrumented Source Code

Compiler

Object Code

Linker

libraries

Executable Code

Binary Rewrite

Dynamic

Virtual Machine

TAU API

Measurement

PROFILE

Run-Time Library Modules

TRACE

Profile Groups

Function Database

Statistics

Function Callstack

Hardware Counters

User-Level Timers

Profiling Data Files

Event Traces

Event Tables

Analysis

Racy

jRacy

pprof

ASCII Report

Trace Logs

merge

convert

Paraver

Vampir

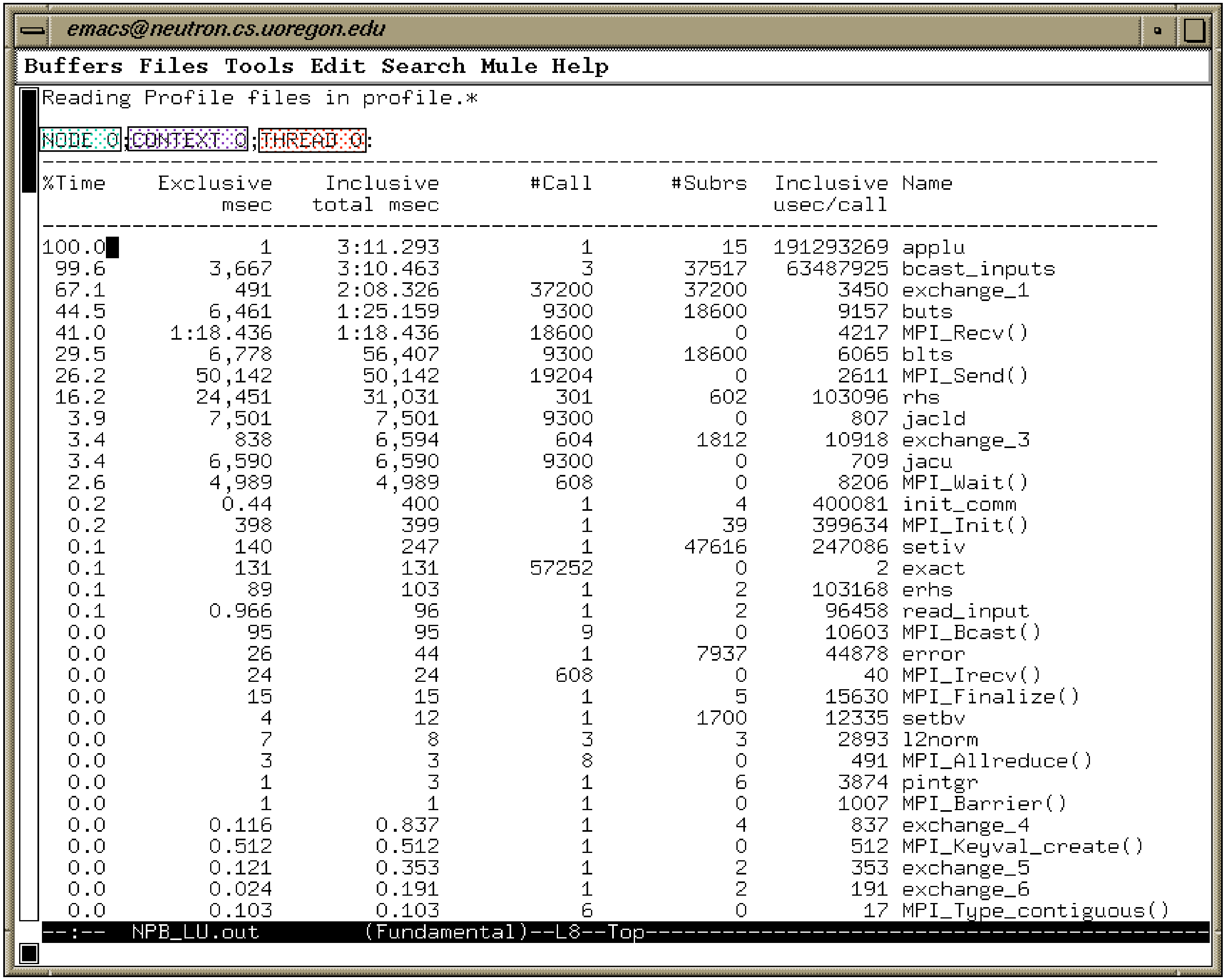
EPILOG

TAU Analysis

- Parallel profile analysis
  - *Pprof*
    - parallel profiler with text-based display
  - *ParaProf*
    - Graphical, scalable, parallel profile analysis and display
- Trace analysis and visualization
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, VTF, Paraver)
  - Trace visualization using *Vampir* (Pallas/Intel)

Pprof Output (NAS Parallel Benchmark – LU)

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI



```
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total  msec                         usec/call
---------------------------------------------------------------------------------------
100.0            1     3:11.293           1          15  191293269 applu
 99.6        3,667     3:10.463           3       37517   63487925 bcast_inputs
 67.1          491     2:08.326       37200       37200       3450 exchange_1
 44.5        6,461     1:25.159        9300       18600       9157 buts
 41.0     1:18.436     1:18.436       18600           0       4217 MPI_Recv()
 29.5        6,778       56,407        9300       18600       6065 blts
 26.2       50,142       50,142       19204           0       2611 MPI_Send()
 16.2       24,451       31,031         301         602     103096 rhs
  3.9        7,501        7,501        9300           0        807 jacld
  3.4          838        6,594         604        1812      10918 exchange_3
  3.4        6,590        6,590        9300           0        709 jacu
  2.6        4,989        4,989         608           0       8206 MPI_Wait()
  0.2         0.44          400           1           4     400081 init_comm
  0.2          398          399           1          39     399634 MPI_Init()
  0.1          140          247           1       47616     247086 setiv
  0.1          131          131       57252           0          2 exact
  0.1           89          103           1           2     103168 erhs
  0.1        0.966           96           1           2      96458 read_input
  0.0           95           95           9           0      10603 MPI_Bcast()
  0.0           26           44           1        7937      44878 error
  0.0           24           24         608           0         40 MPI_Irecv()
  0.0           15           15           1           5      15630 MPI_Finalize()
  0.0            4           12           1        1700      12335 setbv
  0.0            7            8           3           3       2893 l2norm
  0.0            3            3           8           0        491 MPI_Allreduce()
  0.0            1            3           1           6       3874 pintgr
  0.0            1            1           1           0       1007 MPI_Barrier()
  0.0        0.116        0.837           1           4        837 exchange_4
  0.0        0.512        0.512           1           0        512 MPI_Keyval_create()
  0.0        0.121        0.353           1           2        353 exchange_5
  0.0        0.024        0.191           1           2        191 exchange_6
  0.0        0.103        0.103           6           0         17 MPI_Type_contiguous()
--:--  NPB_LU.out        (Fundamental)--L8--Top-----------------------------------------
```

Terminology – Example

- For routine “int main( )”:
- Exclusive time
  - 100-20-50-20=10 secs
- Inclusive time
  - 100 secs
- Calls
  - 1 call
- Subrs (no. of child routines called)
  - 3
- Inclusive time/call
  - 100secs

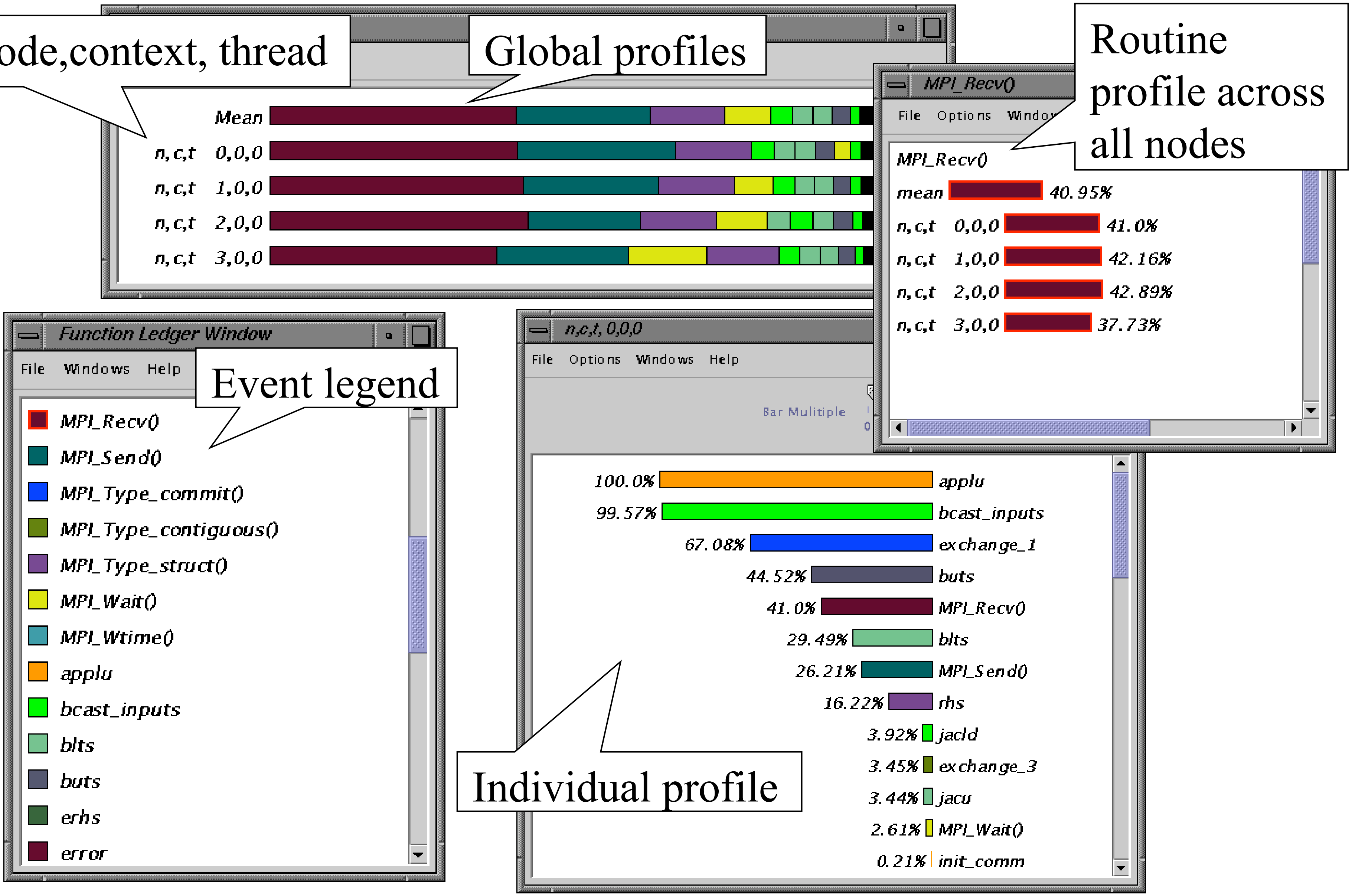
```
int main( )
{ /* takes 100 secs */

  f1(); /* takes 20 secs */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */

  /* other work */
}

/*
Time can be replaced by  counts
*/
```

ParaProf (NAS Parallel Benchmark – LU)



node,context, thread

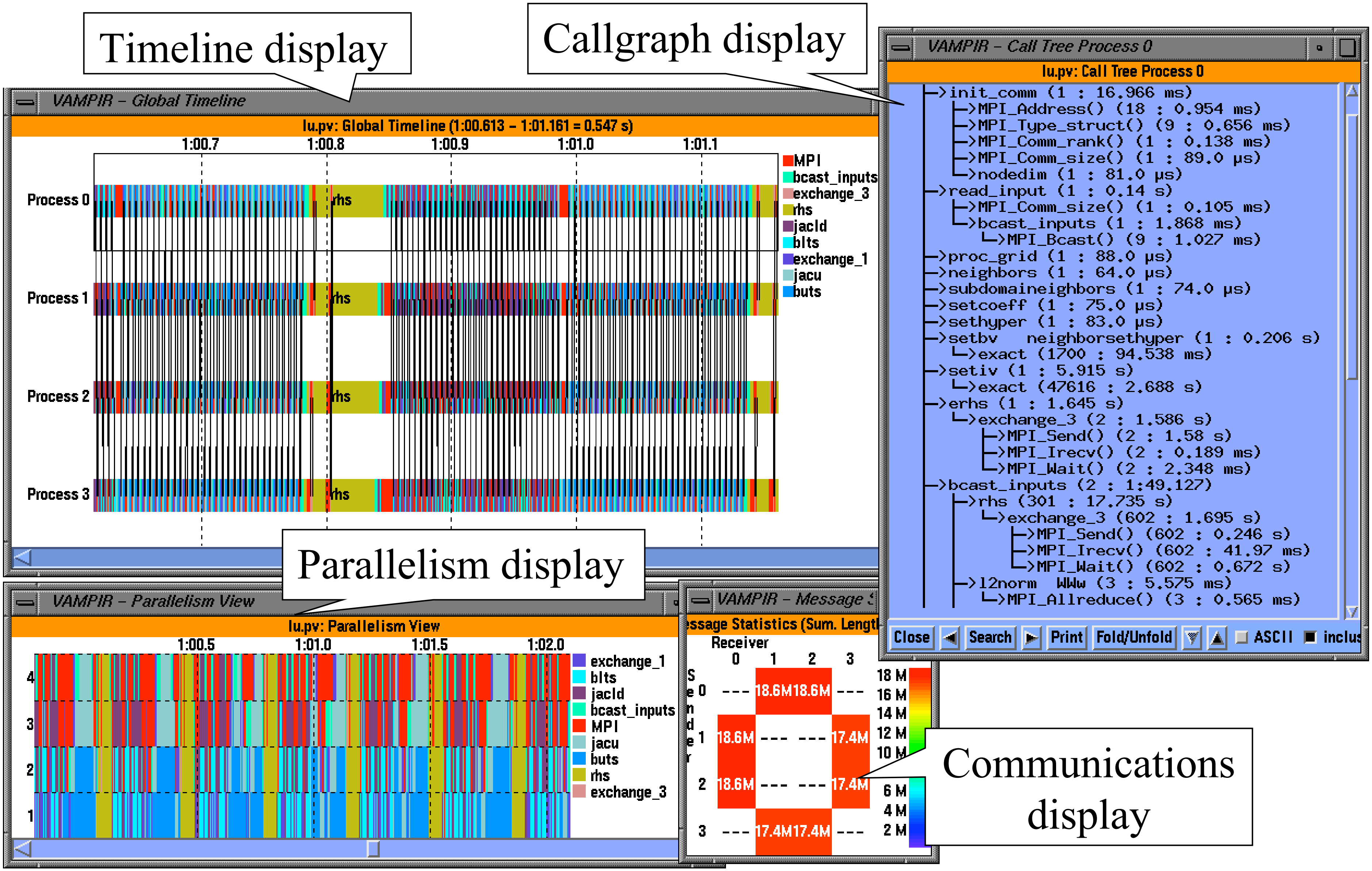
Global profiles

Routine profile across all nodes

Event legend

Individual profile

Trace Visualization using Vampir [Intel/Pallas]



Timeline display

Callgraph display

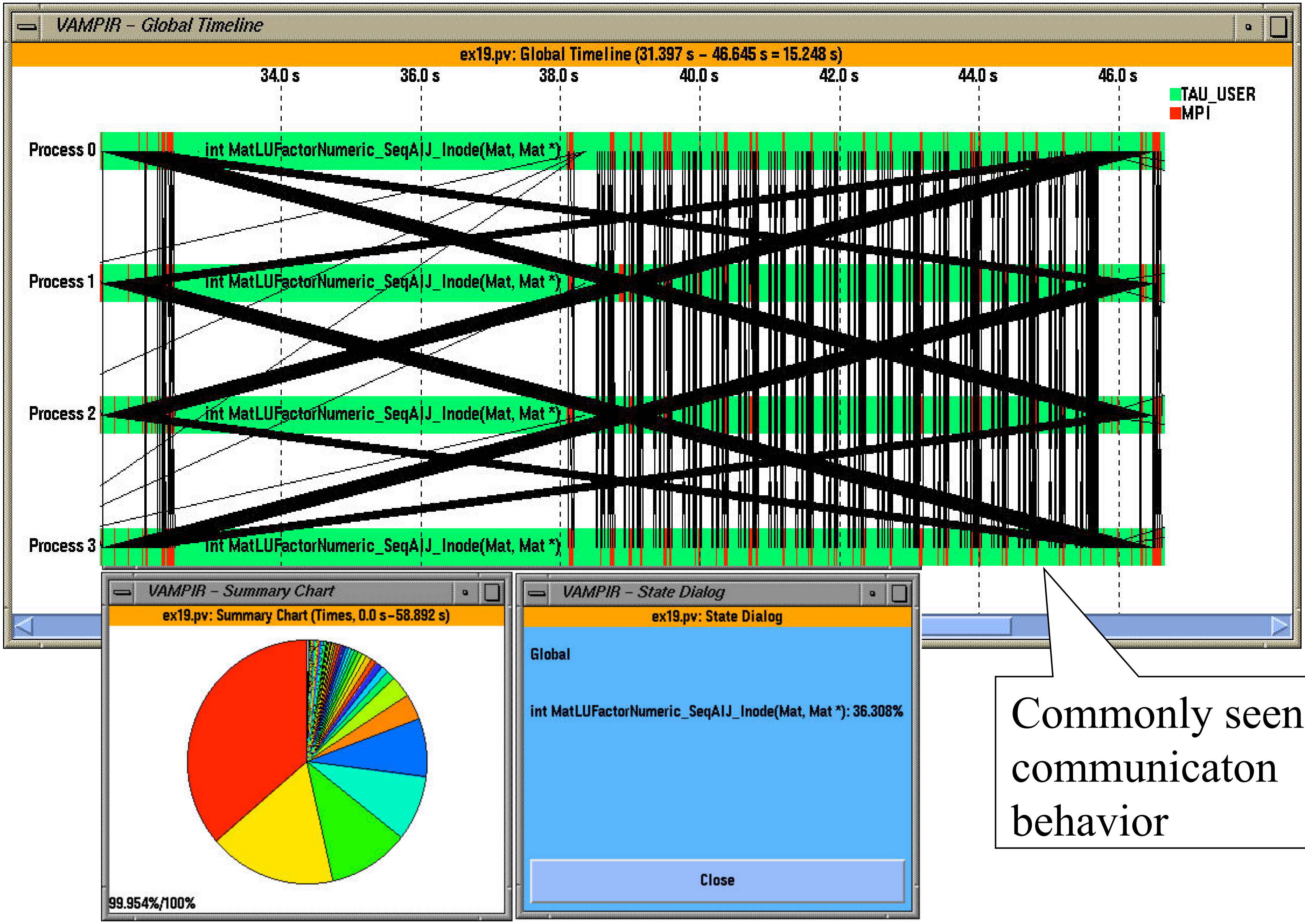
Parallelism display

Communications display

```
──>init_comm (1 : 16.966 ms)
   ├──>MPI_Address() (18 : 0.954 ms)
   ├──>MPI_Type_struct() (9 : 0.656 ms)
   ├──>MPI_Comm_rank() (1 : 0.138 ms)
   ├──>MPI_Comm_size() (1 : 89.0 µs)
   └──>nodedim (1 : 81.0 µs)
──>read_input (1 : 0.14 s)
   ├──>MPI_Comm_size() (1 : 0.105 ms)
   └──>bcast_inputs (1 : 1.868 ms)
      └──>MPI_Bcast() (9 : 1.027 ms)
──>proc_grid (1 : 88.0 µs)
──>neighbors (1 : 64.0 µs)
──>subdomaineighbors (1 : 74.0 µs)
──>setcoeff (1 : 75.0 µs)
──>sethyper (1 : 83.0 µs)
──>setbv   neighborsethyper (1 : 0.206 s)
   └──>exact (1700 : 94.538 ms)
──>setiv (1 : 5.915 s)
   └──>exact (47616 : 2.688 s)
──>erhs (1 : 1.645 s)
   └──>exchange_3 (2 : 1.586 s)
      ├──>MPI_Send() (2 : 1.58 s)
      ├──>MPI_Irecv() (2 : 0.189 ms)
      └──>MPI_Wait() (2 : 2.348 ms)
──>bcast_inputs (2 : 1:49.127)
   ├──>rhs (301 : 17.735 s)
   │  └──>exchange_3 (602 : 1.695 s)
   │     ├──>MPI_Send() (602 : 0.246 s)
   │     ├──>MPI_Irecv() (602 : 41.97 ms)
   │     └──>MPI_Wait() (602 : 0.672 s)
   ├──>l2norm  WWw (3 : 5.575 ms)
      └──>MPI_Allreduce() (3 : 0.565 ms)
```

Sender \ Receiver	0	1	2	3
0	---	18.6M	18.6M	---
1	18.6M	---	---	17.4M
2	18.6M	---	---	17.4M
3	---	17.4M	17.4M	---

PETSc ex19 (Tracing)



ex19.pv: Global Timeline (31.397 s – 46.645 s = 15.248 s)

ex19.pv: Summary Chart (Times, 0.0 s–58.892 s)

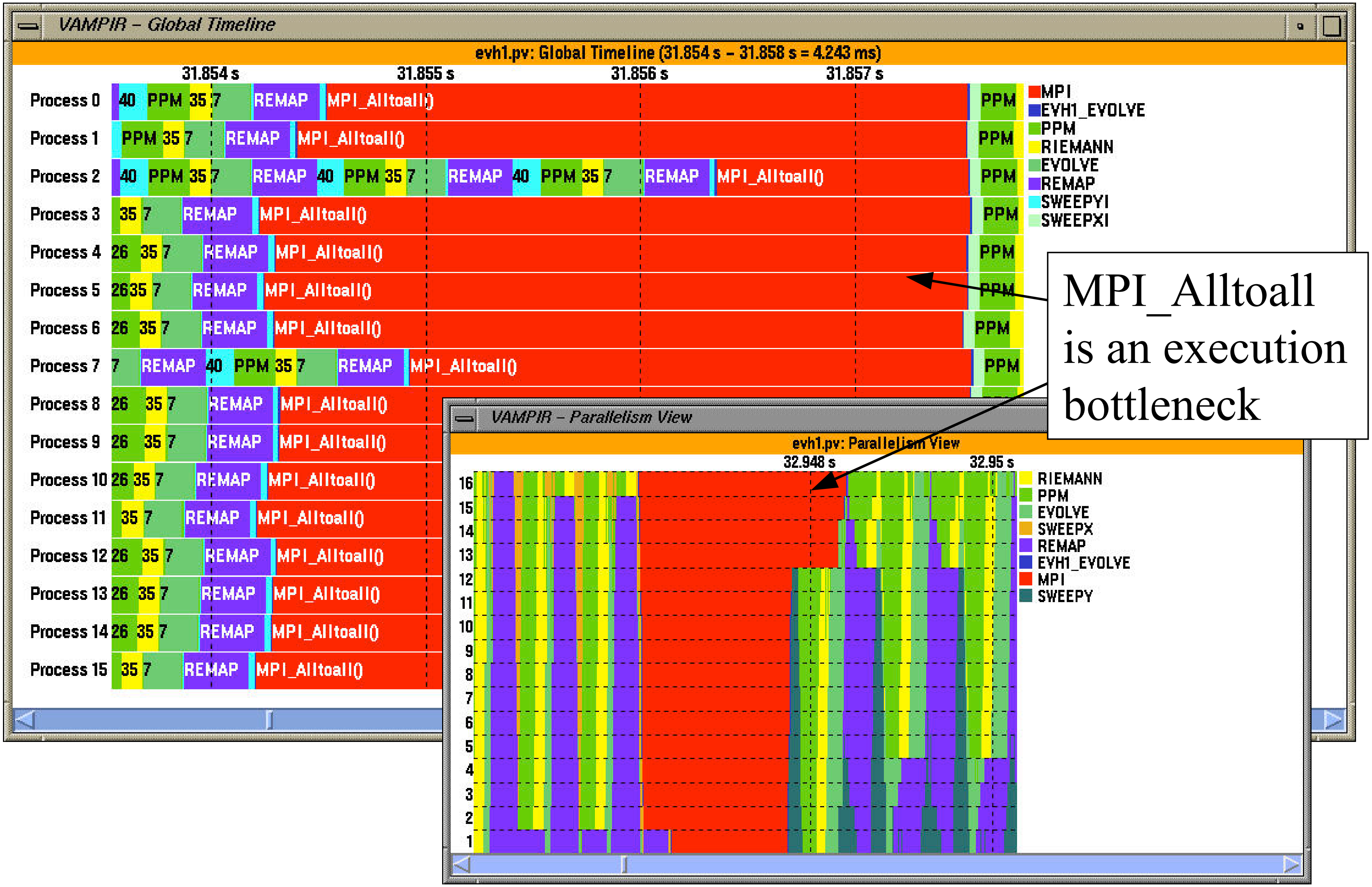
ex19.pv: State Dialog

Global

int MatLUFactorNumeric_SeqAIJ_Inode(Mat, Mat *): 36.308%

Commonly seen communicaton behavior

TAU's EVH1 Execution Trace in Vampir



MPI_Alltoall is an execution bottleneck

Strategies for Empirical Performance Evaluation

- Empirical performance evaluation as a series of performance experiments
  - Experiment trials describing instrumentation and measurement requirements
  - Where/When/How axes of empirical performance space
    - where are performance measurements made in program
      - routines, loops, statements…
    - when is performance instrumentation done
      - compile-time, while pre-processing, runtime…
    - how are performance measurement/instrumentation chosen
      - profiling with hw counters, tracing, callpath profiling…

TAU Instrumentation Approach

- Support for standard program events
  - Routines
  - Classes and templates
  - Statement-level blocks
- Support for user-defined events
  - Begin/End events (“user-defined timers”)
  - Atomic events
  - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual
    - automatic
      - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
      - OpenMP (directive rewriting (*Opari*), *POMP spec*)
  - Object code
    - pre-instrumented libraries (e.g., MPI using *PMPI*)
    - statically-linked and dynamically-linked
  - Executable code
    - dynamic instrumentation (pre-execution) (*DynInstAPI*)
    - virtual machine instrumentation (e.g., Java using *JVMPI*)

Multi-Level Instrumentation

- Targets common measurement interface
  - *TAU API*
- Multiple instrumentation interfaces
  - Simultaneously active
- Information sharing between interfaces
  - Utilizes instrumentation knowledge between levels
- Selective instrumentation
  - Available at each level
  - Cross-level selection
- Targets a common performance model
- Presents a unified view of execution
  - Consistent performance events

TAU Measurement Options

- Parallel profiling
  - Function-level, block-level, statement-level
  - Supports user-defined events
  - TAU parallel profile data stored during execution
  - Hardware counts values
  - Support for multiple counters
  - Support for callgraph and callpath profiling
- Tracing
  - All profile-level events
  - Inter-process communication events
  - Trace merging and format conversion

Optimizing Instrumentation

- Grouping
  - Enable/disable profile groups at runtime
- Selective Instrumentation
  - Include/exclude events (or files) for instrumentation
- Re-instrumentation
  - Profile, overhead analysis, exclude events, re-instrument
- Compensation
  - Overhead calibration, removal

Grouping Performance Data in TAU

- Profile Groups
  - A group of related routines forms a profile group
  - Statically defined
    - TAU_DEFAULT, TAU_USER[1-5], TAU_MESSAGE, TAU_IO, …
  - Dynamically defined
    - group name based on string, such as “adlib” or “particles”
    - runtime lookup in a map to get unique group identifier
    - uses *tau_instrumentor* to instrument
  - Ability to change group names at runtime
  - Group-based instrumentation and measurement control

Selective Instrumentation

- Selection of which performance events to observe
  - Could depend on scope, type, level of interest
  - Could depend on instrumentation overhead
- How is selection supported in instrumentation system?
  - No choice
  - Include / exclude routine and file lists (TAU)
  - Environment variables
  - Static vs. dynamic

Automatic Instrumentation of Source Code

```
% cxxparse file.cpp -I/dir -Dflags      [PDT: Program Database Toolkit]
% tau_instrumentor
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline]
[-g groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.
BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST
# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```

Distortion of Performance Data

- Problem: Controlling instrumentation of small routines
  - High relative measurement overhead
  - Significant intrusion and possible perturbation
- Solution: Re-instrument the application!
  - Weed out frequently executing lightweight routine
  - Feedback to instrumentation system

Re-instrumentation

- ***Tau_reduce***: rule based overhead analysis
- Analyze the performance data to determine events with high (relative) overhead performance measurements
- Create a select list for excluding those events
- Rule grammar (used in *tau_reduce* tool [N. Trebon, UO])

  [GroupName:] Field Operator Number

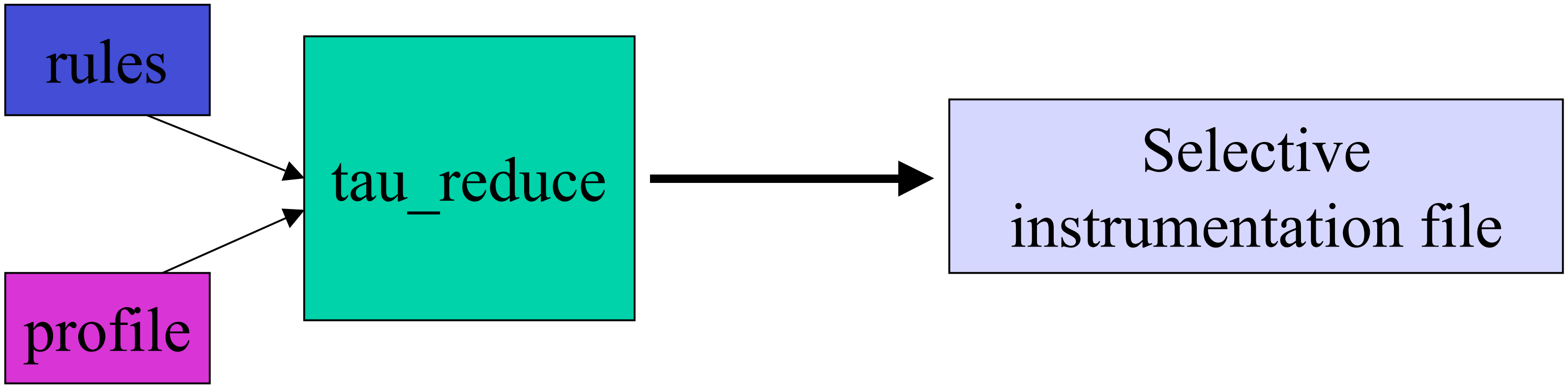
  - *GroupName* indicates rule applies to events in group
  - *Field* is a event metric attribute (from profile statistics)
    - numcalls, numsubs, percent, usec, cumusec, count [PAPI], totalcount, stdev, usecs/call, counts/call
  - *Operator* is one of >, <, or =
  - *Number* is any number
  - Compound rules possible using & between simple rules

Example Rules

- ❐ #Exclude all events that are members of TAU_USER #and use less than 1000 microseconds
  TAU_USER:usec < 1000
- ❐ #Exclude all events that have less than 100 #microseconds and are called only once
  usec < 1000 & numcalls = 1
- ❐ #Exclude all events that have less than 1000 usecs per #call OR have a (total inclusive) percent less than 5
  usecs/call < 1000
  percent < 5
- ❐ Scientific notation can be used
  - ❍ **usec>1000 & numcalls>400000 & usecs/call<30 & percent>25**

TAU_REDUCE

- Reads profile files and rules
- Creates selective instrumentation file
  - Specifies which routines should be excluded from instrumentation



rules

profile

tau_reduce

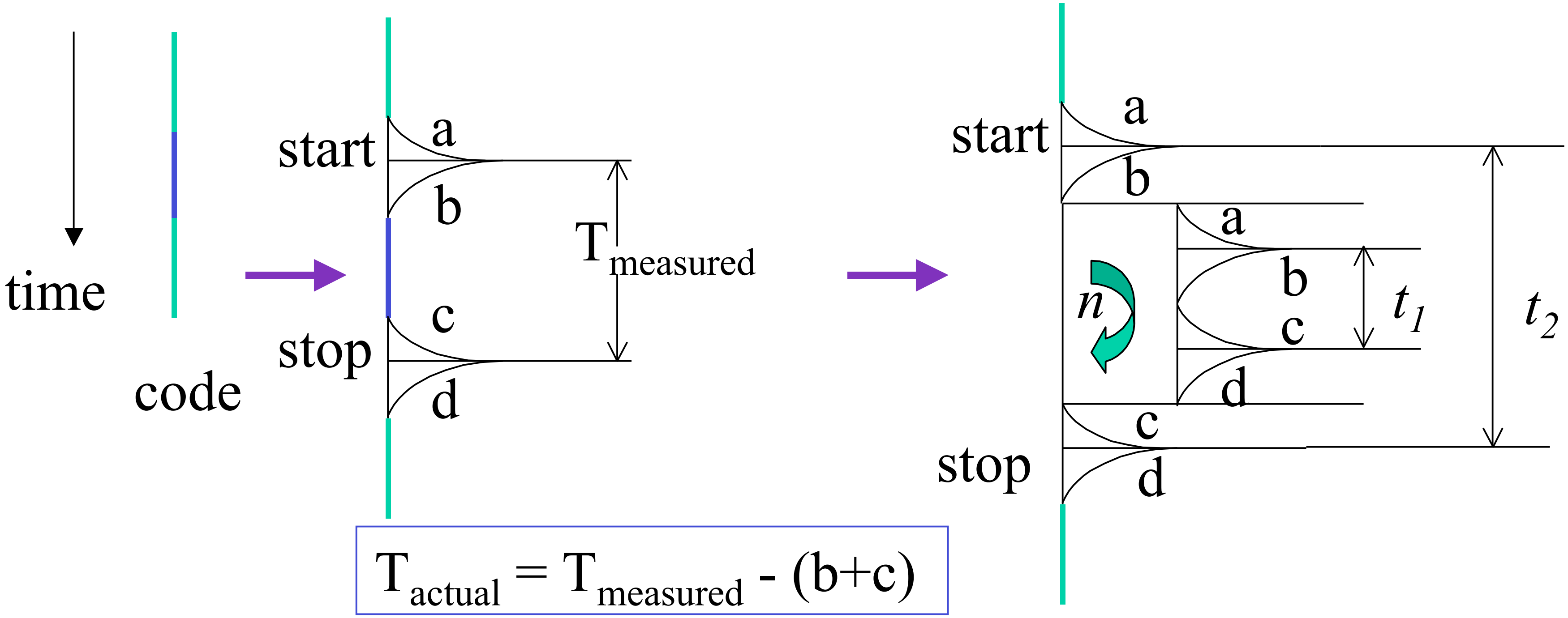
Selective instrumentation file

Compensation of Overhead

- Runtime estimation of a single timer overhead
- Evaluation of number of timer calls along a calling path
- Compensation by subtracting timer overhead
- Recalculation of performance metrics

Estimating Timer Overheads

- Introduce a pair of timer calls (start/stop)



$T_{actual} = T_{measured} - (b+c)$

$t_1 = n * (b+c)$

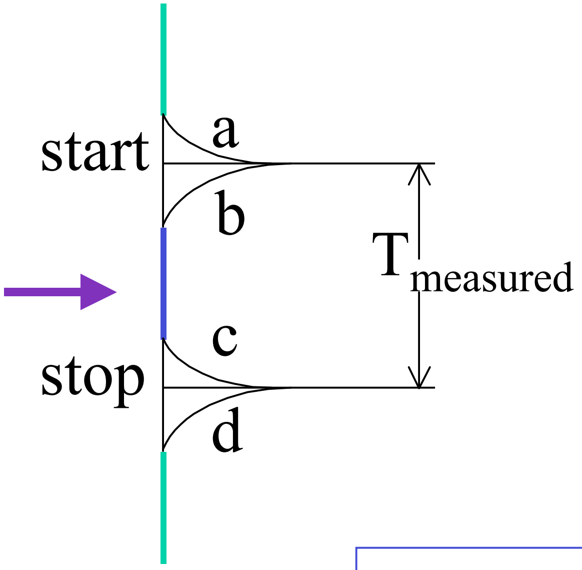
$t_2 = b+n*(a+b+c+d)+c$

$T_{overhead} = a+b+c+d = (t_2 - (t_1/n))/n$

$T_{null} = b+c = t_1/n$

Recalculating Inclusive Time

- Number of children/grandchildren… nodes
- Traverse callstack

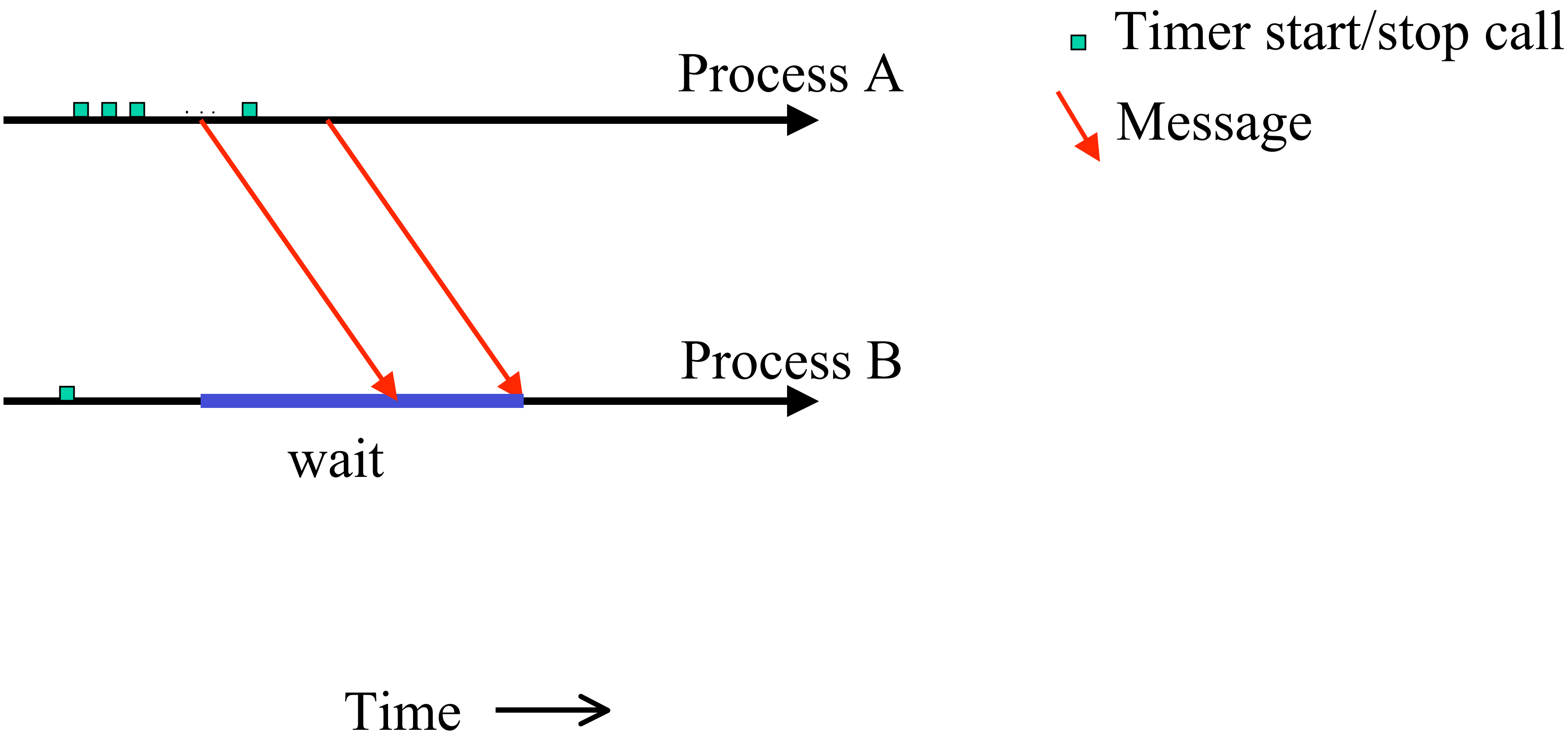


main
=>
f1
=> f2
…
f3
=> f4

$$T_{actual} = T_{measured} - (b+c) - n_{descendants} * T_{overhead}$$

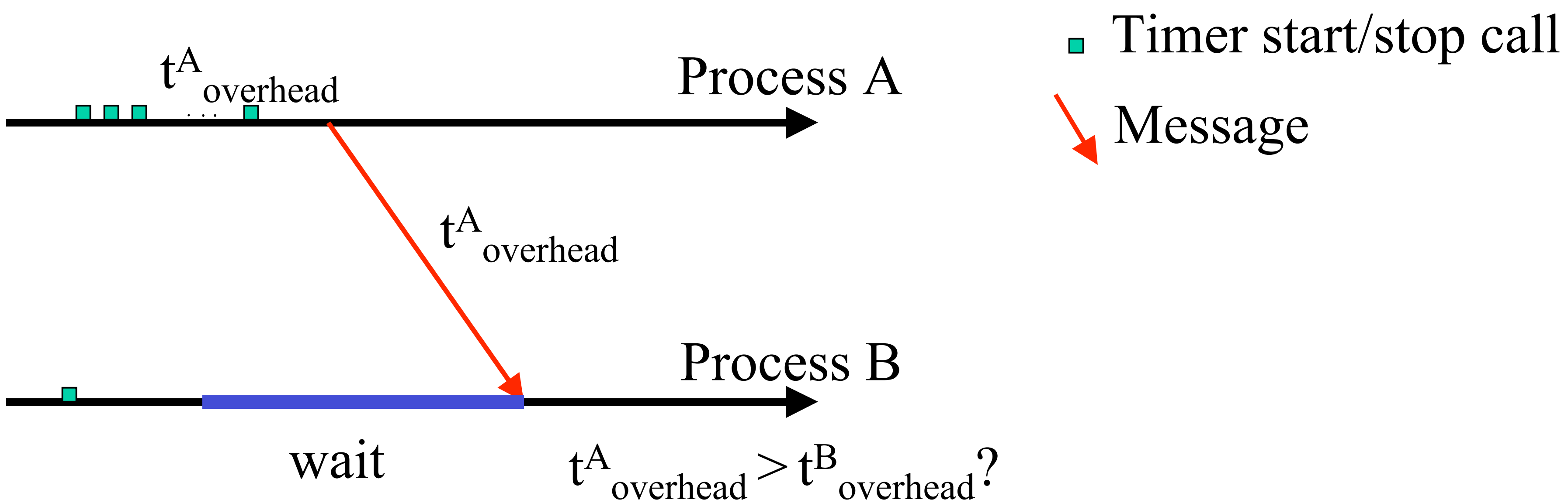
Parallel Performance Compensation

- Compensate for synchronization operations



Lamport's Logical Time [Lamport 1978]

- Logical time incremented by timer start/stop
- Accumulate timer overhead on local process
- Send local timer overhead with message



$t^A_{overhead} > t^B_{overhead}$?

Yes : $t^B_{overhead} = t^A_{overhead}$

$t_{wait}' = t_{wait} - (t^A_{overhead} - t^B_{overhead})$

$= 0$ (if negative)

Time ⟶

Compensation (contd.)

- Message passing programs
  - Adjust wait times (MPI_Recv, MPI_Wait…)
  - Adjust barrier wait times (MPI_Barrier)
    - Each process sends its timer overheads to all other tasks
    - Each task compares its overhead with max overhead
- Shared memory multi-threaded programs
  - Adjust barrier synchronization wait times
    - Each task compares its overhead to max overhead from all participating threads
  - Adjust semaphore/condition variable wait times
    - Each task compares its overhead with other thread's overhead

Conclusions

- Complex software and parallel computing systems pose challenging performance analysis problems that require robust methodologies and tools
- Optimizing instrumentation is a key step towards balancing the volume of performance data with accuracy of measurements
- Present new research in the area of performance perturbation compensation techniques for profiling
- http://www.cs.uoregon.edu/research/paracomp/tau

Support Acknowledgements

- Department of Energy (DOE)
  - Office of Science contracts
  - University of Utah DOE ASCI Level 1 sub-contract
  - DOE ASCI Level 3 (LANL, LLNL)
- NSF National Young Investigator (NYI) award
- Research Centre Juelich
  - John von Neumann Institute for Computing
  - Dr. Bernd Mohr
- Los Alamos National Laboratory