# Performance Analysis and HPC

Philip J. Mucci
Visiting Scientist, PDC, KTH
Research Consultant, ICL, UT Knoxville

**mucci at cs.utk.edu**

**http://www.cs.utk.edu/~mucci**


Parallelldatorcentrum

August 26th, 2005


High Performance Computing Summer School at PDC

# Outline

- Why Performance Analysis?
- Types of Performance Analysis
- Different types of Tools
- Hardware Performance Analysis
- Tools on Lucidor and Lenngren
  - You should be trying them this afternoons.

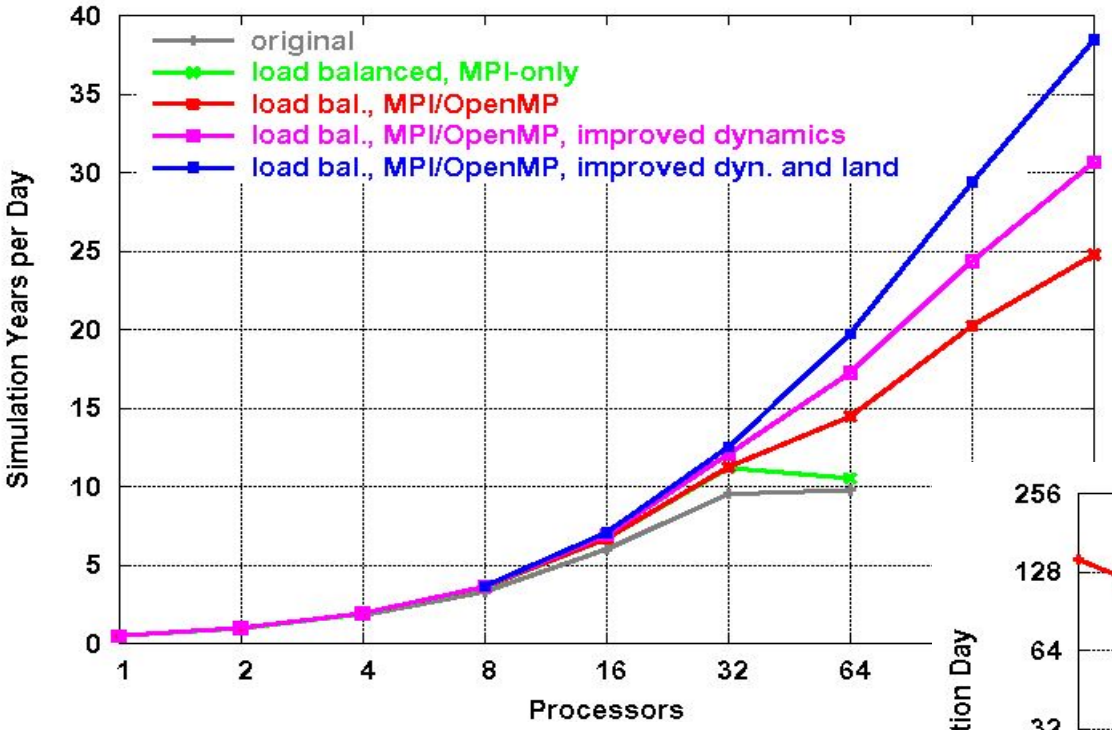
# Why Performance Analysis?

- 2 reasons: Economic & Qualitative
- Economic: TIME IS MONEY
  - Average lifetime of these large machines is 4 years before being decommissioned.
  - Consider the cost per day of a 4 Million Dollar machine, with annual maintenance/electricity cost of $300,000 (US). That's $1500.00 (US) per hour of compute time.
  - Many HPC centers charge departments by the CPU hour

# Why Performance Analysis? (2)

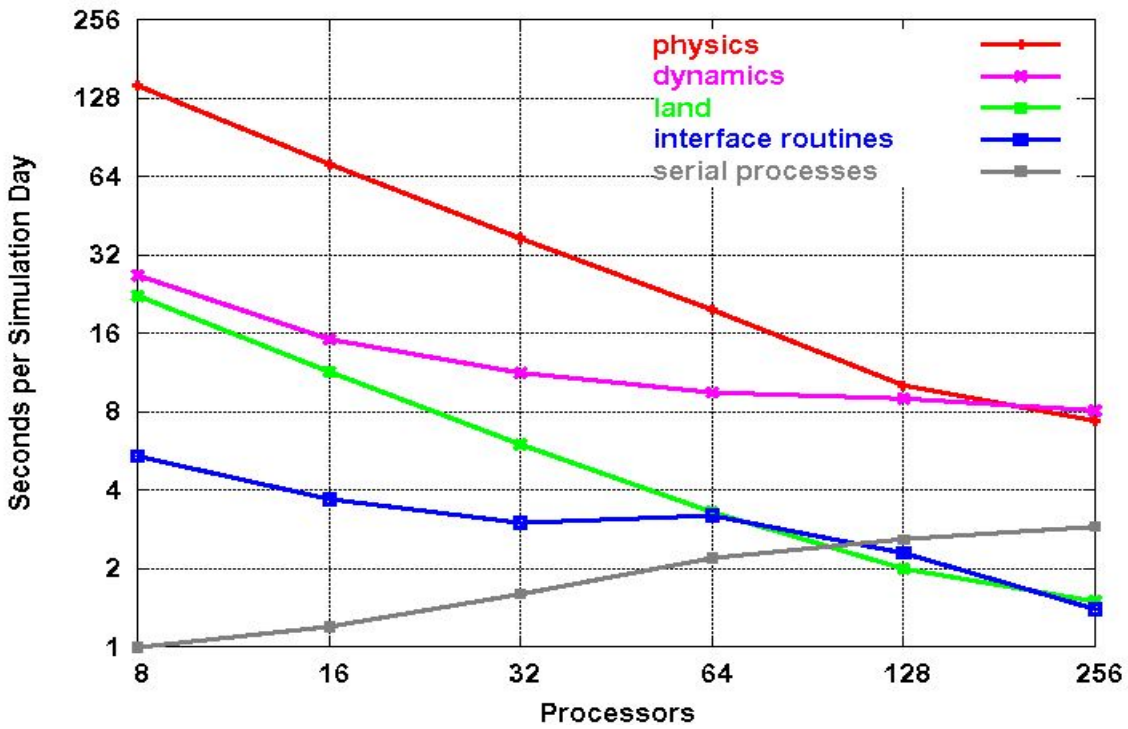
- Qualitative Improvements in Science
  - Consider: Poorly written code can easily run 10 times worse than an optimized version.
  - Consider a 2-dimension domain decomposition of a Finite Difference formulation simulation.
  - For the same amount of time, the code can do 10 times the work. 400x400 elements vs. 1300x1300 elements
  - Or it can do 400x400 for 10 times more time-steps.
  - These could be the difference in resolving the phenomena of interest!

# Biology and Environmental Sciences
# CAM




CAM performance measurements on IBM p690 cluster (and other platforms) were used to direct development process. Graph shows performance improvement from performance tuning and recent code modifications.

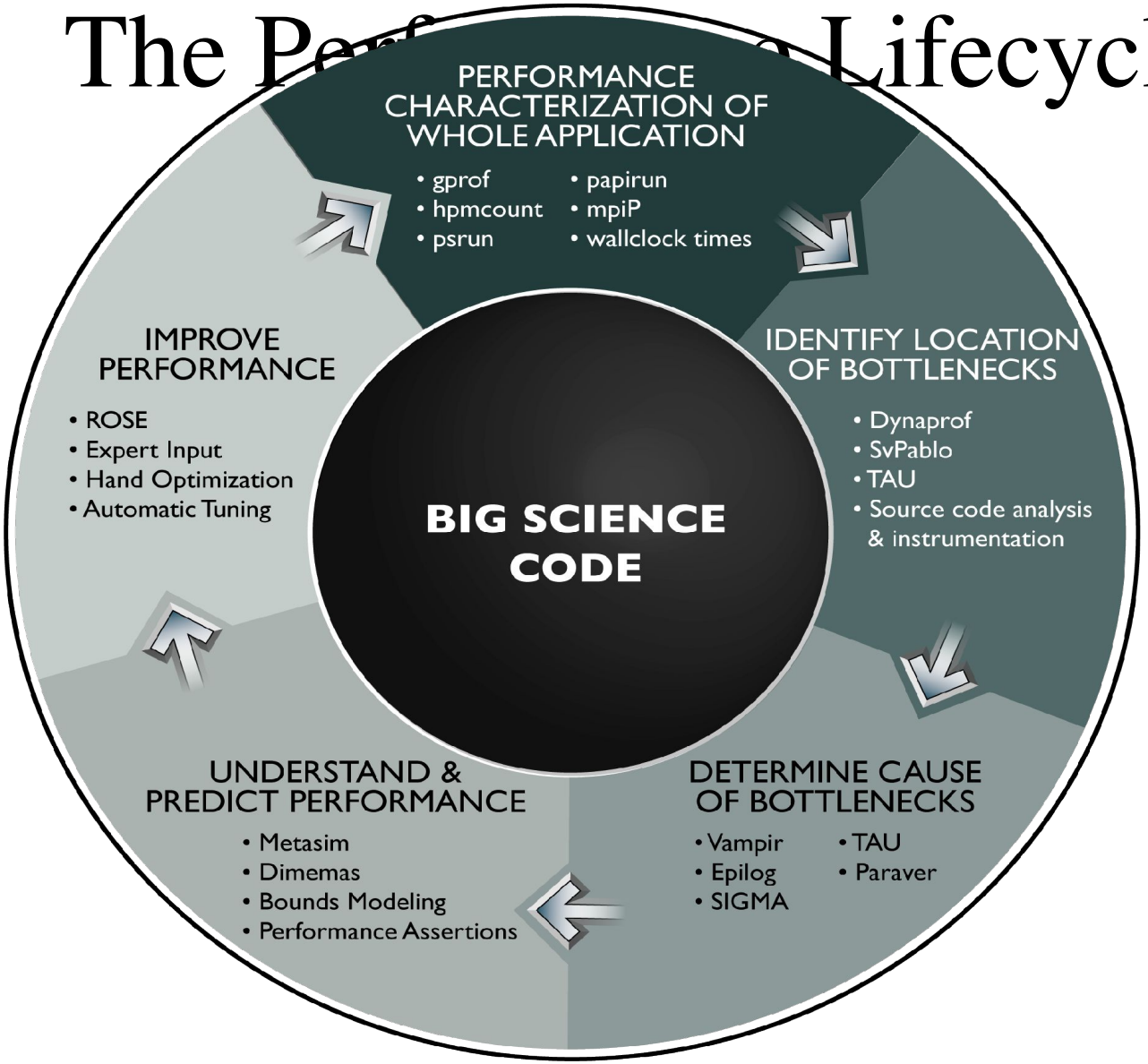
Profile of current version of CAM indicates that improving the serial performance of the physics is the most important optimization for small numbers of processors, and introducing a 2D decomposition of the dynamics (to improve scalability) is the most important optimization for large numbers of processors.



# Why Performance Analysis? (3)

- So, we must strive to evaluate how our code is running.
- Learn to think of performance during the entire cycle of your design and implementation.
- Systems will be in place at PDC to recognize a 'slow' code that is consuming large amounts of CPU time.

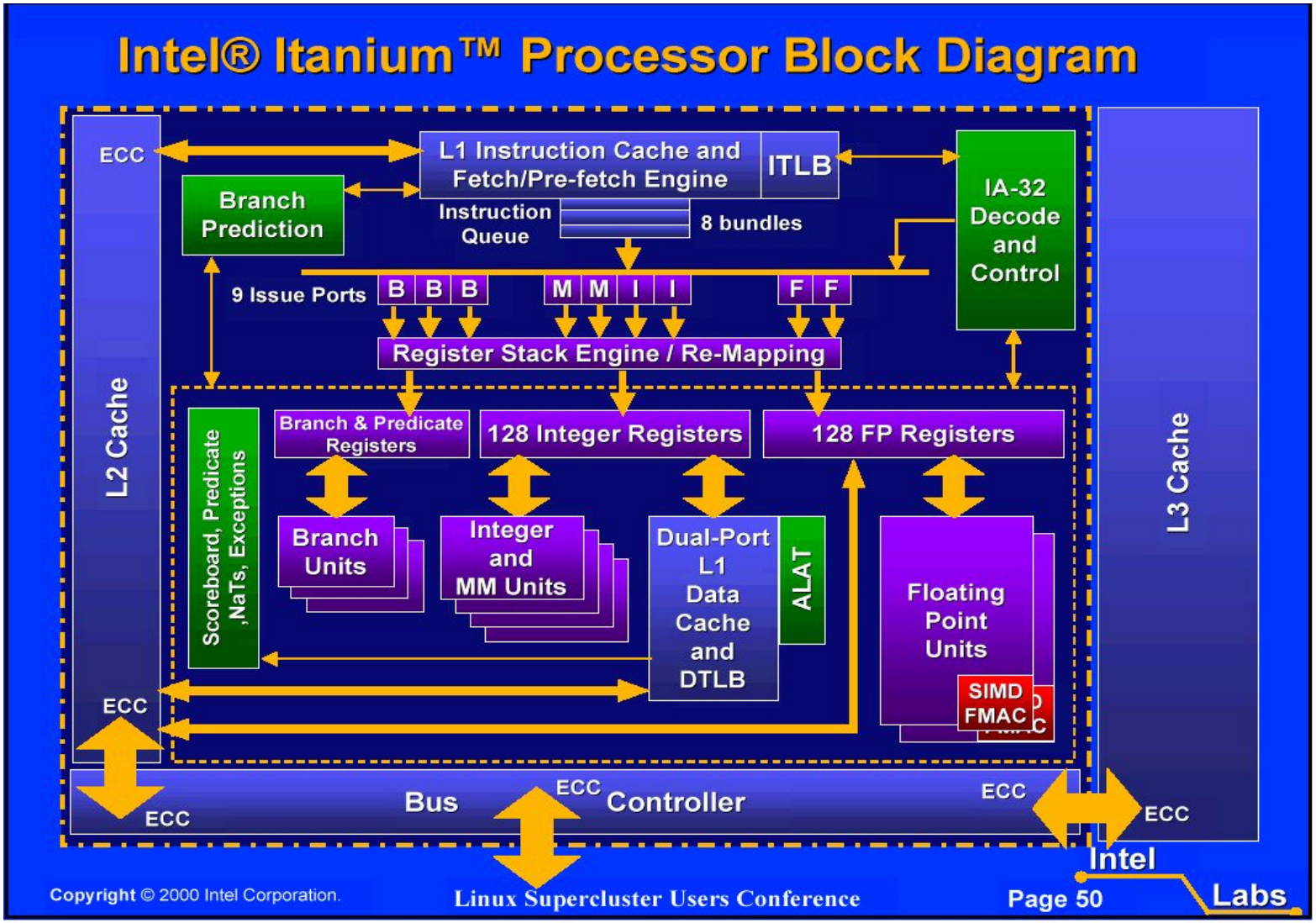
# The Performance Lifecycle



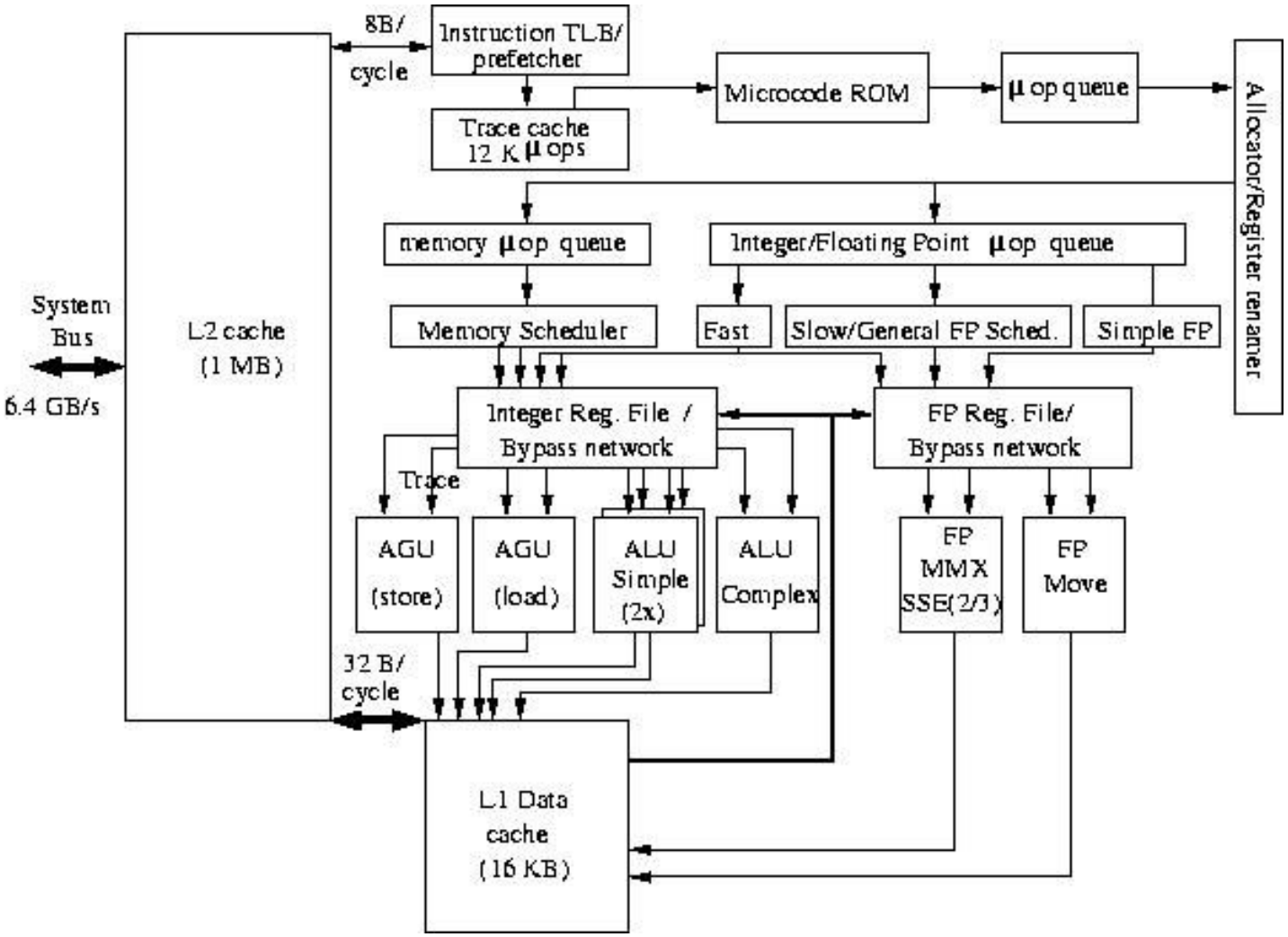
# Rising Processor Complexity

- No longer can we easily trace the execution of a segment of code.
  - Static/Dynamic Branch Prediction
  - Hardware Prefetching
  - Out-of-order scheduling
  - Predication
- So, just a measure of 'wallclock' time is not enough.
- Need to know what's really happening under the hood.

# Processor Complexity (IA64)



# Processor Complexity (EM64T)



# Performance Evaluation

- Traditionally, performance evaluation has been somewhat of an art form:
  - Limited set of tools (**`time`** & **`gprof`** with **`-p/-pg`**)
  - Major differences between systems
  - Lots of guesswork as to what was 'behind the numbers'
- Today, the situation is different.
  - Hardware support for performance analysis
  - A wide variety of quality Open Source tools to choose from.

# Measurement Methods

- Direct methods requires explicit instrumentation in some form.
  - Tracing
    - Generate a record for each measured event.
    - Useful only when evidence of performance anomalies is present due to the large volume of data generated.
  - Aggregate
    - Reduce data at run-time avg/min/max measurements.
    - Useful for application and architecture characterization and optimization.

# Measurement Methods (2)

- Indirect methods requires no instrumentation and can be used on unmodified applications.
- The reality is that the boundary between indirect and direct is somewhat fuzzy.
  - **`gprof`** (no source mods, but requires recompile or relink)

# External Timers

- **`/usr/bin/time <command>`** returns 3 kinds.
  - Real time: Time from start to finish
  - User: CPU time spent executing your code
  - System:  CPU time spent executing system calls
- Warning! The definition of CPU time is different on different machines.
  - Multi-threaded executables add each processes CPU time to the total.

# External Timers (2)

- Sample output (from Linux)

```
0.56user 0.12system 0:03.80elapsed 18%CPU
  (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (55major+2684minor)pagefaults 0swaps
```

1) User

2) System

3) Real

4) Percent of time spent on behalf of this process, not including waiting.

5) Text size, data size, max memory

6) 0 input, 0 output operations

7) Page faults (major, minor), swaps.

# Internal Timers

- `gettimeofday()`, part of the C library obtains seconds and microseconds since Jan 1, 1970.
- `second()`, Fortran 90.
- `MPI_Wtime()`, MPI
- Latency is not the same as resolution.
  - Many calls to this function will affect your wall clock time.

# Internal Timers

- `clock_gettime()` for POSIX, usually implemented as `gettimeofday()`.
- `MPI_Wtime()` returns elapsed wall clock time in seconds as a double.

# The Trouble with Timers

- They depend on the load on the system.
  - Elasped wall clock time does not reflect the actual time the program is doing work due to:
    - OS interference.
    - Daemon processes.
- The solution?
  - We need measurements that are accurate yet independent of external factors. (Help from the OS)

# Hardware Performance Counters

- Performance Counters are hardware registers dedicated to counting certain types of events within the processor or system.
  - Usually a small number of these registers (2,4,8)
  - Sometimes they can count a lot of events or just a few
  - Symmetric or asymmetric
  - May be on or off chip
- Each register has an associated control register that tells it what to count and how to do it.
  - Interrupt on overflow
  - Edge detection (cycles vs. events)
  - User vs. kernel mode

# Hardware Performance Counter Virtualization

- Every process appears to have its own counters.
- OS accumulates counts into 64-bit quantities for each thread and process.
  - Saved and restored on context switch.
- Both user and kernel modes can be measured.
- Explicit counting or statistical histograms based on counter overflow.
- Counts are largely independent of load.

# Performance Counters

- Most high performance processors include hardware performance counters.
  - AMD Athlon and Opteron
  - Compaq Alpha EV Series
  - CRAY T3E,  X1
  - IBM Power Series
  - Intel Itanium, Itanium 2, Pentium
  - SGI MIPS R1xK Series
  - Sun UltraSparc II+
  - And many others...

# Available Performance Data

- Cycle count
- Instruction count
  - All instructions
  - Floating point
  - Integer
  - Load/store
- Branches
  - Taken / not taken
  - Mispredictions
- Pipeline stalls due to
  - Memory subsystem
  - Resource conflicts
- Cache
  - I/D cache misses for different levels
  - Invalidations
- TLB
  - Misses
  - Invalidations

# PAPI

- **P**erformance **A**pplication **P**rogramming **I**nterface
- The purpose of PAPI is to implement a standardized portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- The goal of PAPI is to facilitate the optimization of parallel and serial code performance by encouraging the development of cross-platform optimization tools.

# PAPI Preset Events

- Proposed set of events deemed most relevant for application performance tuning
- Preset events are mappings from symbolic names to machine specific definitions for a particular hardware resource.
  - Total Cycles is PAPI_TOT_CYC
- Mapped to native events on a given platform
- PAPI also supports presets that may be derived from the underlying hardware metrics.

# PAPI Native Events

- These are the performance metrics as specified by the hardware.
- Usually require a more detailed understanding of the architecture.
- Names are different for every architecture.
- Example:
  - EM64T/Pentium4 Instruction TLB misses are specified as ITLB_reference_MISS

# PAPI Presets on Lucidor (IA64)

| Preset | Description |
|---|---|
| PAPI_L1_DCM | Level 1 data cache misses |
| PAPI_L1_ICM | Level 1 instruction cache misses |
| PAPI_L2_DCM | Level 2 data cache misses |
| PAPI_L2_ICM | Level 2 instruction cache misses |
| PAPI_L3_DCM | Level 3 data cache misses |
| PAPI_L3_ICM | Level 3 instruction cache misses |
| PAPI_L1_TCM | Level 1 cache misses |
| PAPI_L2_TCM | Level 2 cache misses |
| PAPI_L3_TCM | Level 3 cache misses |
| PAPI_CA_SNP | Requests for a snoop |
| PAPI_CA_INV | Requests for cache line invalidation |
| PAPI_L3_LDM | Level 3 load misses |
| PAPI_L3_STM | Level 3 store misses |
| PAPI_TLB_DM | Data translation lookaside buffer misses |
| PAPI_TLB_IM | Instruction translation lookaside buffer misses |
| PAPI_TLB_TL | Total translation lookaside buffer misses |
| PAPI_L1_LDM | Level 1 load misses |
| PAPI_L2_LDM | Level 2 load misses |
| PAPI_L2_STM | Level 2 store misses |
| PAPI_L3_DCH | Level 3 data cache hits |
| PAPI_STL_ICY | Cycles with no instruction issue |
| PAPI_STL_CCY | Cycles with no instructions completed |
| PAPI_BR_MSP | Conditional branch instructions mispredicted |
| PAPI_BR_PRC | Conditional branch instructions correctly predicted |
| PAPI_TOT_IIS | Instructions issued |
| PAPI_TOT_INS | Instructions completed |
| PAPI_LD_INS | Load instructions |
| PAPI_SR_INS | Store instructions |
| PAPI_BR_INS | Branch instructions |
| PAPI_RES_STL | Cycles stalled on any resource |
| PAPI_FP_STAL | Cycles the FP unit(s) are stalled |
| PAPI_TOT_CYC | Total cycles |

# PAPI Presets on Lucidor (2)

| Preset | Description |
|---|---|
| PAPI_L1_DCH | Level 1 data cache hits |
| PAPI_L2_DCH | Level 2 data cache hits |
| PAPI_L1_DCA | Level 1 data cache accesses |
| PAPI_L2_DCA | Level 2 data cache accesses |
| PAPI_L3_DCA | Level 3 data cache accesses |
| PAPI_L1_DCR | Level 1 data cache reads |
| PAPI_L2_DCR | Level 2 data cache reads |
| PAPI_L3_DCR | Level 3 data cache reads |
| PAPI_L2_DCW | Level 2 data cache writes |
| PAPI_L3_DCW | Level 3 data cache writes |
| PAPI_L3_ICH | Level 3 instruction cache hits |
| PAPI_L1_ICA | Level 1 instruction cache accesses |
| PAPI_L2_ICA | Level 2 instruction cache accesses |
| PAPI_L3_ICA | Level 3 instruction cache accesses |
| PAPI_L1_ICR | Level 1 instruction cache reads |
| PAPI_L2_ICR | Level 2 instruction cache reads |
| PAPI_L3_ICR | Level 3 instruction cache reads |
| PAPI_L2_TCH | Level 2 total cache hits |
| PAPI_L3_TCH | Level 3 total cache hits |
| PAPI_L1_TCA | Level 1 total cache accesses |
| PAPI_L2_TCA | Level 2 total cache accesses |
| PAPI_L3_TCA | Level 3 total cache accesses |
| PAPI_L1_TCR | Level 1 total cache reads |
| PAPI_L2_TCR | Level 2 total cache reads |
| PAPI_L3_TCR | Level 3 total cache reads |
| PAPI_L2_TCW | Level 2 total cache writes |
| PAPI_L3_TCW | Level 3 total cache writes |
| PAPI_FP_OPS | Floating point operations |

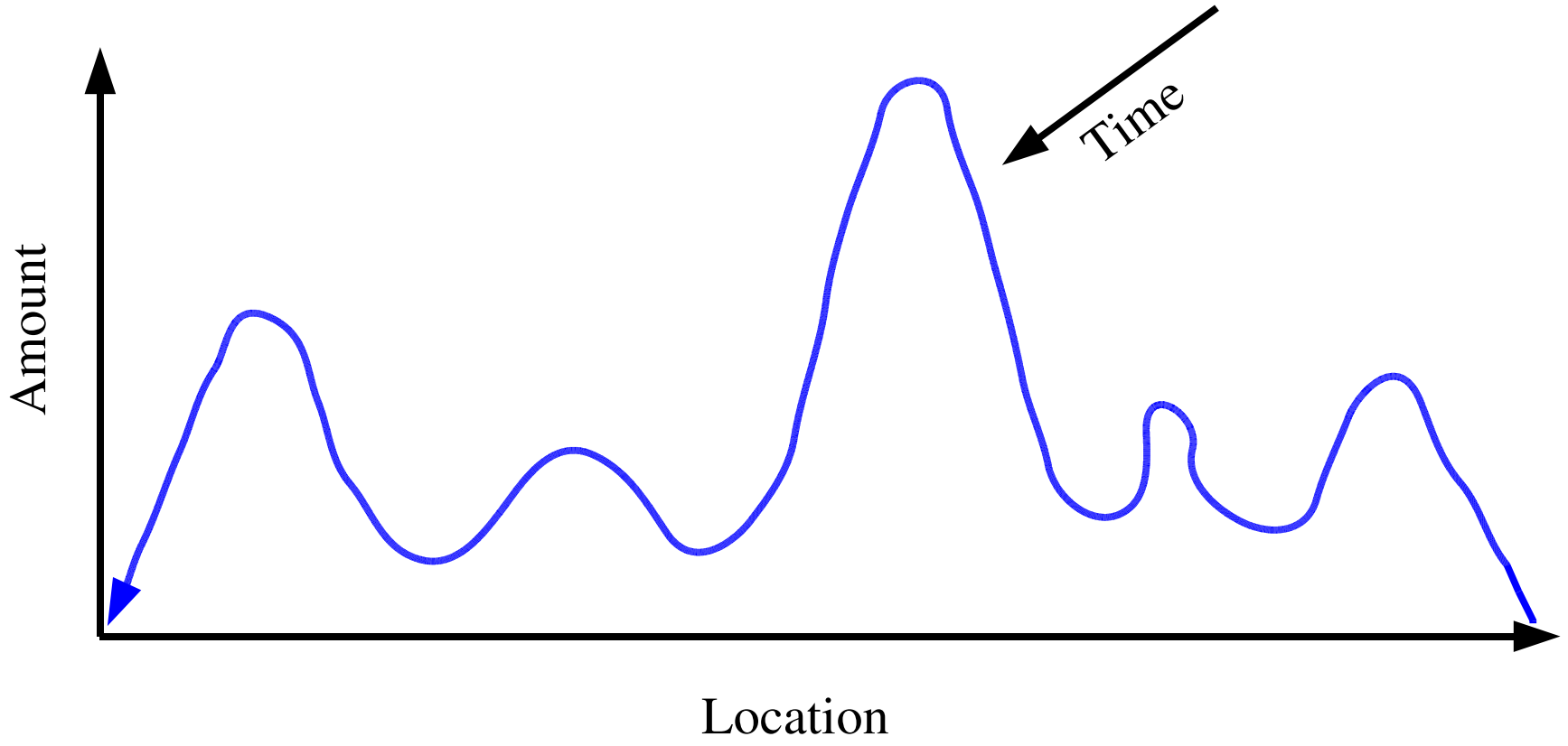
# PAPI Presets on Lenngren (EM64T)

| Preset | Description |
|---|---|
| PAPI_L1_DCM | Level 1 data cache misses |
| PAPI_L1_ICM | Level 1 instruction cache misses |
| PAPI_L2_TCM | Level 2 cache misses |
| PAPI_TLB_DM | Data translation lookaside buffer misses |
| PAPI_TLB_IM | Instruction translation lookaside buffer misses |
| PAPI_TLB_TL | Total translation lookaside buffer misses |
| PAPI_L1_LDM | Level 1 load misses |
| PAPI_L2_LDM | Level 2 load misses |
| PAPI_L2_STM | Level 2 store misses |
| PAPI_BR_TKN | Conditional branch instructions taken |
| PAPI_BR_NTK | Conditional branch instructions not taken |
| PAPI_BR_MSP | Conditional branch instructions mispredicted |
| PAPI_BR_PRC | Conditional branch instructions correctly predicted |
| PAPI_TOT_IIS | Instructions issued |
| PAPI_TOT_INS | Instructions completed |
| PAPI_FP_INS | Floating point instructions |
| PAPI_LD_INS | Load instructions |
| PAPI_SR_INS | Store instructions |
| PAPI_BR_INS | Branch instructions |
| PAPI_VEC_INS | Vector/SIMD instructions |
| PAPI_RES_STL | Cycles stalled on any resource |
| PAPI_TOT_CYC | Total cycles |
| PAPI_LST_INS | Load/store instructions completed |
| PAPI_L1_ICA | Level 1 instruction cache accesses |
| PAPI_FP_OPS | Floating point operations |

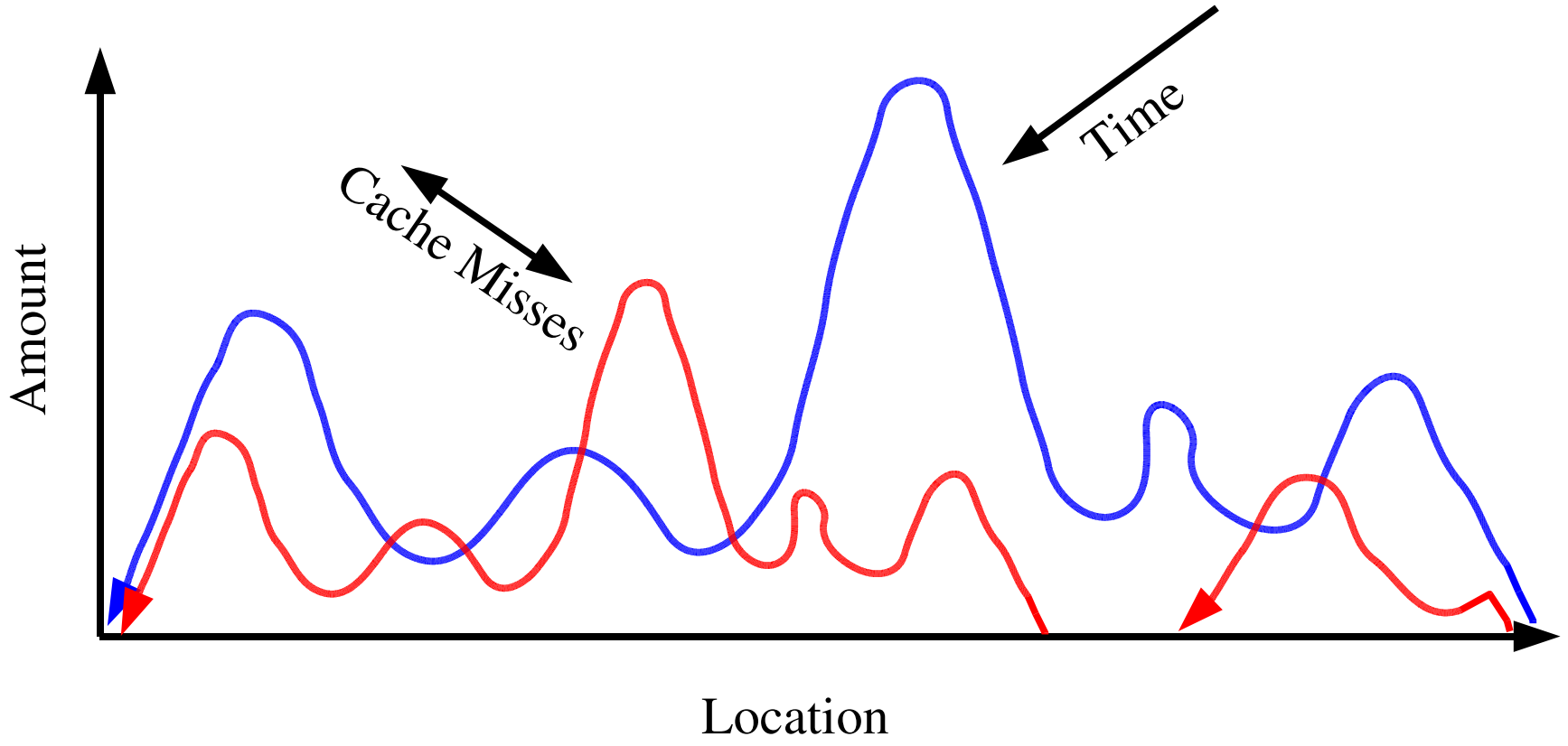
# Statistical Profiling

- Dr. Andersson introduced you to gprof.
- At a defined interval (interrupts), record WHERE in the program the CPU is.
- Data gathered represents a probabilistic distribution in the form of a histogram.
- Interrupts can be based on time or hardware counter events with the proper infrastructure like...

# Statistical Profiling



# Hardware Statistical Profiling



# Parallel Performance

"The single most important impediment to good parallel performance is *still* poor single-node performance."

- William Gropp
Argonne National Lab

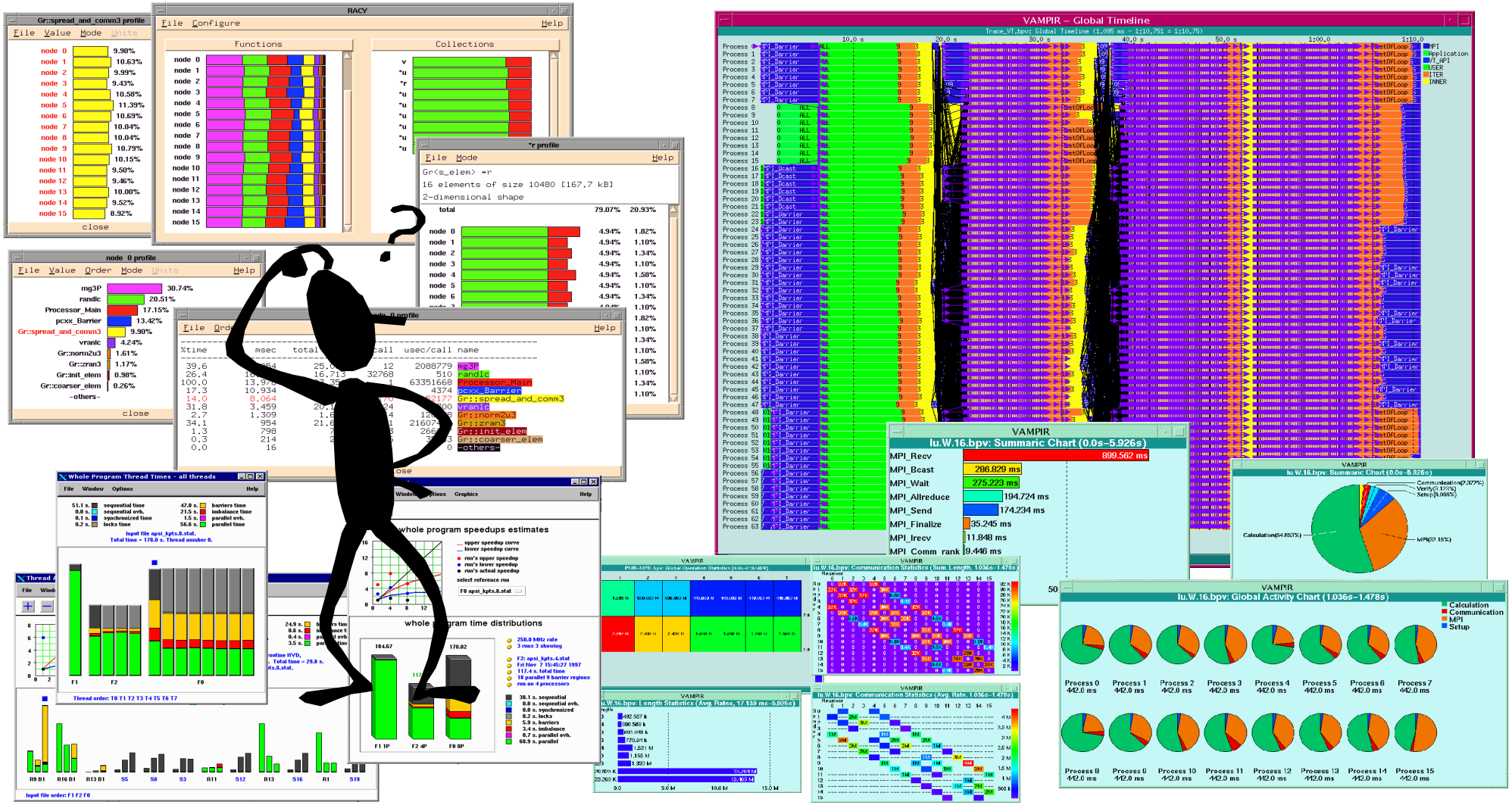
# What is Good Parallel Performance?

- Single CPU performance is high.
- The code is scalable out to more than a few nodes.
- The network is not the bottleneck.
- In parallel computation, algorithm design is the key to good performance.
- You must reduce the amount of data that needs to be sent around.

# Beware the Fallacy of Reported Linear Scalability

- But what about per/PE performance?
- With a slow code, overall performance of the code is not vulnerable to other system parameters like communication bandwidth, latency.
- Very common on tightly integrated systems where you can simple add PE's for performance.

# Which Tool?


Gr::spread_and_comm3 profile
RACY
*r profile
node_0 profile
VAMPIR – Global Timeline
VAMPIR
lu.W.16.bpv: Summaric Chart (0.0s–5.926s)
MPI_Recv 899.562 ms
MPI_Bcast 286.829 ms
MPI_Wait 275.223 ms
MPI_Allreduce 194.724 ms
MPI_Send 174.234 ms
MPI_Finalize 35.245 ms
MPI_Irecv 11.848 ms
MPI_Comm_rank 9.446 ms
lu.W.16.bpv: Global Activity Chart (1.036s–1.478s)
Whole Program Thread Times - all threads
whole program speedups estimates
whole program time distributions

# The Right Performance Tool

- You must have the right tool for the job.
- What are your needs? Things to consider:
  - User Interface
    - Complex Suite
    - Quick and Dirty
  - Data Collection Mechanism
    - Aggregate
    - Trace based
    - Statistical

# The Right Performance Tool (2)

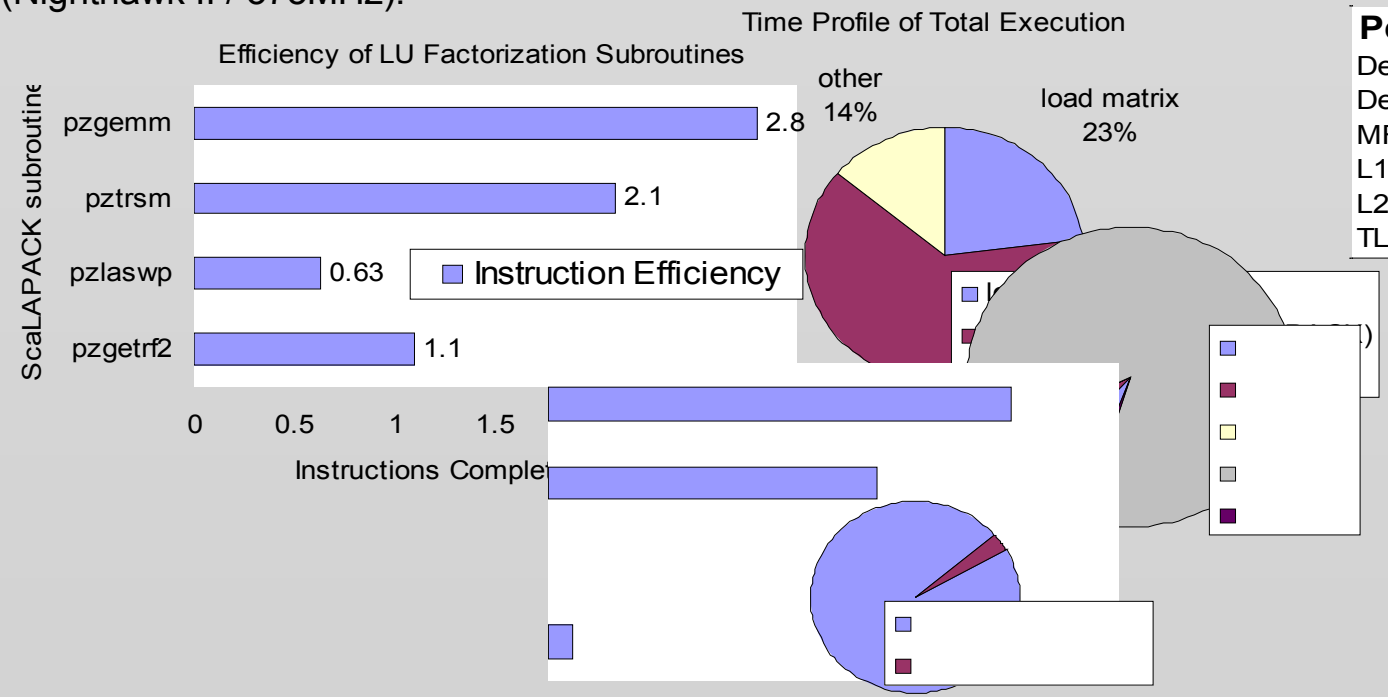
- Performance Data
  - Communications (MPI)
  - Synchronization (Threads and OpenMP)
  - External Libraries
  - User code
- Data correlation
  - Task Parallel (MPI)
  - Thread Parallel
- Instrumentation Mechanism
  - Source
  - Binary (DPCL/DynInst)
  - Library interposition

# The Right Performance Tool (3)

- Data Management
  - Performance Database
  - User (Flat file)
- Data Visualization
  - Run Time
  - Post Mortem
  - Serial/Parallel Display
  - ASCII

# Fusion Sciences
# AORSA3D

AORSA3D was ported and benchmarked on IBM and Compaq platforms. A detailed performance analysis has begun using SvPablo and PAPI. The results below are for a 400 Fourier mode run on 16 processors and 1 node of an IBM SP (Nighthawk II / 375MHz).







| Performance for ZGemm | |
|---|---|
| Denisty of Mem Access | 1 |
| Denisty of FLOPs | 1.8 |
| MFLOP/s | 664 |
| L1 cache hit rate | 0.98 |
| L2 cache hit rate | 0.96 |
| TLB misses | 285034 |

# Linux Performance Infrastructure

- Contrary to popular belief, the Linux infrastructure is well established.
- PAPI is +8 years old.
- Wide complement of tools from which to choose.
- Some are production quality.
- Sun, IBM and HP are now focusing on Linux/HPC which means a focus on performance.

# Available Performance Tools on PDC Linux Clusters

- Lucidor: IA64
- Lenngren: EM64T
- Exactly the same performance tools.
- 2 Completely different architectures.
- The magic of portable software.

# Available Performance Tools on PDC Linux Clusters (2)

- We will cover 4 simple tools, although many more complex ones are available.
  - papiex: hardware performance measurement.
  - hpctoolkit: hardware statistical profiling.
  - mpip: MPI performance and usage.
  - jumpshot: MPI message tracing.

# Available Performance Tools on PDC Clusters (3)

- All performance tools discussed here are available via:

```
module load perftools/1.2
```

# Hardware Event Measurement and papiex

- A simple tool that generates performance measurements for the entire run of a code.
- Requires no recompilation.
- Monitors all subprocesses/threads.
- Output goes to stderr or a file.
- Can monitor memory and resource usage.
- Provides hooks for simple instrumentation of user source code if desired.

# PapiEx Features

- Automatically detects multi-threaded executables.
- Counter multiplexing with -m, use more counters than available hardware. Check -i.
- Full memory usage information with -x.
- Simple instrumentation API.
  - Called PapiEx Calipers.

# papiex Usage

```
Usage: papiex [-lihVmukord] [-L event] [-f[prefix]] [-F file] [-e
   event] ... -- <cmd> <cmd options>

 -l              List the available events.

 -L event        List information about specific event.

 -i              Print information about the host machine.

 -h              Print this message.

 -V              Print version information.

 -m              Enable multiplexing of hardware counters.

 -u              Monitor user mode events. (default)

 -k              Monitor kernel mode events.

 -o              Monitor transient mode events.

 -r              Report getrusage() information.

 -x              Report memory information.

 -d              Enable debugging output.

 -f[prefix]      Output to <prefix><cmd>.papiex.<host>.<pid>.<tid>.

 -e event        Monitor this hardware event.
```

# Papiex Example

```
> module load monitor papi papiex

> papiex <application>

> papiex -e PAPI_TOT_CYC -e PAPI_TOT_INS --
  <application>

> mpirun -np 4 `which papiex` -f --
  <application>
```

# PapiEx Output

```
PapiEx Version:         0.99rc2
Executable:             /afs/pdc.kth.se/home/m/mucci/summer/a.out
Processor:              Itanium 2
Clockrate:              900.000000
Parent Process ID:      8632
Process ID:             8633
Hostname:               h05n05.pdc.kth.se
Options:                MEMORY
Start:                  Wed Aug 24 14:34:18 2005
Finish:                 Wed Aug 24 14:34:19 2005
Domain:                 User
Real usecs:             1077497
Real cycles:            969742309
Proc usecs:             970144
Proc cycles:            873129600
PAPI_TOT_CYC:           850136123
PAPI_FP_OPS:            40001767
Mem Size:               4064
Mem Resident:           2000
Mem Shared:             1504
Mem Text:               16
Mem Library:            2992
Mem Heap:               576
Mem Locked:             0
Mem Stack:              32

Event descriptions:
Event: PAPI_TOT_CYC
        Derived: No
        Short Description: Total cycles
        Long Description: Total cycles
        Developer's Notes:
Event: PAPI_FP_OPS
        Derived: No
        Short Description: FP operations
        Long Description: Floating point operations
        Developer's Notes:
```

# PapiEx Caliper Fortran Example

```
#include "papiex.h"

      program zero

      real a, b, c;
      a = 0.1
      b = 1.1
      c = 2.1

      PAPIEX_START_ARG(1,"write")

      print *, "Doing 10000000 iters. of a += b * c on doubles."

      PAPIEX_STOP_ARG(1)

      PAPIEX_START_ARG(2,"do loop")

      do i=1,100000000
         a = a + b * c
      end do

      PAPIEX_STOP_ARG(2)

      end
```

# PapiEx Caliper C/C++ Example

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "papiex.h"

volatile double a = 0.1, b = 1.1, c = 2.1;

int main(int argc, char **argv)
{
  int i;

  PAPIEX_START_ARG(1,"printf");

  printf("Doing 100000000 iters. of a += b * c on doubles.\n");

  PAPIEX_STOP_ARG(1);

  PAPIEX_START_ARG(2,"for loop");

  for (i=0;i<100000000;i++)
    a += b * c;

  PAPIEX_STOP_ARG(2);

  exit(0);
}
```

# PapiEx Caliper Example

```
bash-3.00$ papiex -e PAPI_L1_DCM ./a.out
 Doing 10000000 iters. of a += b * c on doubles.

[normal papiex output...]
PAPI_L1_DCM:            6864

Caliper 1: write
        Executions:             1
        Real usecs:             42
        Real cycles:            144432
        Proc usecs:             42
        Proc cycles:            144304
        PAPI_L1_DCM:            667
Caliper 2: do loop
        Executions:             1
        Real usecs:             769107
        Real cycles:            2608043669
        Proc usecs:             769019
        Proc cycles:            2607743748
        PAPI_L1_DCM:            4167

Event descriptions:
Event: PAPI_L1_DCM
        Derived: No
        Short Description: L1D cache misses
        Long Description: Level 1 data cache misses
        Developer's Notes:
```

# HPCToolkit

- A statistical profiling package based on interrupts from the performance monitoring hardware.
- No instrumentation required, but compiling with -g helps.
- Does not work on statically linked programs.
- 2 phase, collections and visualization.

# HPCToolkit (2)

- This package is reasonably sophisticated, including an HTML and Java GUI for the data.
- Here we only cover the basics.
  - Collecting profiles.
  - Displaying the profiles as text.

# hpcrun Usage

```
hpcrun [-lLVhr] [-t each,all] [-e event:[period]] [-o path] [-f flag]
   <cmd> -- <cmd options>

 -l              List the available events.

 -L              List detailed information about all events.

 -V              Print version information.

 -h              Print this message.

 -r              Do not follow subprocesses.

 -t[each,all]    Profile threaded applications.

 -e event[:period] Sample event every period counts.

 -o path         Directory for output data.

 -f flag         PAPI profile mode.


Default is to profile every 32768 cycles (PAPI_TOT_CYC).
```

**You need to use -t on Lenngren because Scali MPI is threaded!**

# hpcprof Usage

```
hpcprof [-hefrl] [-H dir] <cmd> <profile> ... <profile>
 -h              Print this message.
 -e              Dump all information.
 -f              Dump results by file.
 -r              Dump results by function.
 -l              Dump results by line.
 -H dir       Dump HTML into dir.

There are more options.
```

# HPCToolkit Serial Example

```
> module load hpctoolkit
> gcc -g main.c
> hpcrun ./a.out
> ls ./a.out.*
./a.out.PAPI_TOT_CYC.h05n05.pdc.kth.se.13255.0
> hpcprof -e ./a.out
  a.out.PAPI_TOT_CYC.h05n05.pdc.kth.se.13255.0
```

# hpcprof Output

```
[mucci@h05n05:~]$ hpcprof -e ./a.out ./a.out.PAPI_TOT_CYC.h05n05.pdc.kth.se.13255.0
Columns correspond to the following events [event:period (events/sample)]
  PAPI_TOT_CYC:32767 - Total cycles (24755 samples)

Load Module Summary:
 100.0% /afs/pdc.kth.se/home/m/mucci/a.out

File Summary:
 100.0% <</afs/pdc.kth.se/home/m/mucci/a.out>>/afs/pdc.kth.se/home/m/mucci/main.c

Function Summary:
 100.0% <</afs/pdc.kth.se/home/m/mucci/a.out>>main

Line Summary:
  90.0% <</afs/pdc.kth.se/home/m/mucci/a.out>>/afs/pdc.kth.se/home/m/mucci/main.c:7
   5.8% <</afs/pdc.kth.se/home/m/mucci/a.out>>/afs/pdc.kth.se/home/m/mucci/main.c:6
   4.2% <</afs/pdc.kth.se/home/m/mucci/a.out>>/afs/pdc.kth.se/home/m/mucci/main.c:4

File <</afs/pdc.kth.se/home/m/mucci/a.out>>/afs/pdc.kth.se/home/m/mucci/main.c with profile
annotations.
     1        main()
     2        {
     3        int i;
     4   4.2% for (i=0;i<10000000;i++)
     5        {
     6   5.8% double a = 1.0, b = 2.0, c = 3.0;
     7  90.0% a += b * c + (double)i;
     8        }}
     9
```

# HPCToolkit Lucidor Example

```
> module load hpctoolkit
> mpicc -g hello-mpi.c -o hello-mpi
> spattach -i -p2
> mpirun -np 2 -machinefile $SP_HOSTFILE `which hpcrun` $PWD/hello-mpi
h05n31-e.pdc.kth.se: hello world
h05n31-e.pdc.kth.se: hello world
> ls ./hello-mpi.*
hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.2545.0
hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.2552.0
> hpcprof -l ./hello-mpi.PAPI_TOT_CYC.*
```

# HPCToolkit Lenngren Example

```
> module load hpctoolkit
> mpicc -g hello-mpi.c -o hello-mpi
> spattach -i -p2
> mpirun -np 2 -machinefile $SP_HOSTFILE `which hpcrun`
  -t $PWD/hello-mpi
h05n31-e.pdc.kth.se: hello world
h05n31-e.pdc.kth.se: hello world
> ls ./hello-mpi.*
hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.2545.0
hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.2552.0
> hpcprof -l ./hello-mpi.PAPI_TOT_CYC.*
```

# hpcprof Output

```
[bash-2.05a$ hpcprof -f ./hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.25*
Columns correspond to the following events [event:period (events/sample)]
  PAPI_TOT_CYC:32767 - Total cycles (21831 samples)

File Summary:
  99.5% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>>/afs/pdc.kth.se/home/m/mucci/hello-mpi.c
   0.1% <</lib/ld-2.2.5.so>><unknown>
   0.1% <</afs/pdc.kth.se/pdc/vol/gm/2.0.6-2.4.25/lib/libgm.so.0.0.0>>/pdc/vol/gm/2.0.6-
2.4.25/src/ia64_deb30/../gm-2.0.6_Linux/libgm/gm_bzero.c
   0.1% <</afs/pdc.kth.se/pdc/vol/gm/2.0.6-2.4.25/lib/libgm.so.0.0.0>>/pdc/vol/gm/2.0.6-
2.4.25/src/ia64_deb30/../gm-2.0.6_Linux/include/gm_crc32.h
   0.1% <</lib/libc-2.2.5.so>><unknown>
   0.1% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>><unknown>

bash-2.05a$ hpcprof -l ./hello-mpi.PAPI_TOT_CYC.h05n31.pdc.kth.se.25*
Columns correspond to the following events [event:period (events/sample)]
  PAPI_TOT_CYC:32767 - Total cycles (21831 samples)

Line Summary:
  55.2% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>>/afs/pdc.kth.se/home/m/mucci/hello-mpi.c:10
  32.2% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>>/afs/pdc.kth.se/home/m/mucci/hello-mpi.c:7
  12.1% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>>/afs/pdc.kth.se/home/m/mucci/hello-mpi.c:9
   0.1% <</lib/ld-2.2.5.so>><unknown>:0
   0.1% <</lib/libc-2.2.5.so>><unknown>:0
   0.1% <</afs/pdc.kth.se/pdc/vol/gm/2.0.6-2.4.25/lib/libgm.so.0.0.0>>/pdc/vol/gm/2.0.6-
2.4.25/src/ia64_deb30/../gm-2.0.6_Linux/libgm/gm_bzero.c:47
   0.1% <</afs/pdc.kth.se/home/m/mucci/hello-mpi>><unknown>:0
```

# MPI Performance Analysis

- There are 2 modes, both accomplished through intercepting the calls to MPI.
  - Aggregate
  - Tracing
- Often aggregate is sufficient.
  - MPIP
- Tracing generates massive amounts of data.
  - Jumpshot and the MPE libraries.

# MPI Profiling with MPIP

- Often we want to see:
  - What % of time we are spending in MPI
  - What is the frequency and size distribution of messages
  - Load imbalance.
- A simple and elegant tool is mpiP, it does online trace reduction to provide a good summary of MPI usage and performance.

# MPIP Example

```
> module load mpip

> mpicc srtest.c -o srtest -lmpiP -lbfd
  -liberty

> spattach -i -p2

> mpirun -np 2 -machinefile $SP_HOSTFILE
  $PWD/srtest

> more srtest*mpiP
```

# MPIP Output

```
@ Command : /afs/pdc.kth.se/home/m/mucci/mpiP-2.7/testing/./sweep-ops-stack.exe
/tmp/SPnodes-mucci-0
@ Version                  : 2.7
@ MPIP Build date          : Aug 17 2004, 17:04:36
@ Start time               : 2004 08 17 17:08:48
@ Stop time                : 2004 08 17 17:08:48
@ MPIP env var             : [null]
@ Collector Rank           : 0
@ Collector PID            : 17412
@ Final Output Dir         : .
@ MPI Task Assignment      : 0 h05n05-e.pdc.kth.se
@ MPI Task Assignment      : 1 h05n35-e.pdc.kth.se
@ MPI Task Assignment      : 2 h05n05-e.pdc.kth.se
@ MPI Task Assignment      : 3 h05n35-e.pdc.kth.se

@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0      0.084     0.0523    62.21
   1     0.0481      0.015    31.19
   2      0.087     0.0567    65.20
   3     0.0495     0.0149    29.98
   *      0.269      0.139    51.69

@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%
Barrier                 1        112   41.57   80.42
Recv                    1       26.2    9.76   18.89
Allreduce               1      0.634    0.24    0.46
Bcast                   1        0.3    0.11    0.22
Send                    1      0.033    0.01    0.02

@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
---------------------------------------------------------------------------
Call                 Site      Count      Total       Avrg  Sent%
Allreduce               1          8    4.8e+03        600  46.15
Bcast                   1          8    4.8e+03        600  46.15
Send                    1          2        800        400   7.69
```

# MPIP Output (2)

```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 16 ---------------------
---------------------------------------------------------------------------
Name              Site Rank  Count      Max     Mean      Min   App%   MPI%
Allreduce            1    0      2    0.105    0.087    0.069   0.21   0.33
Allreduce            1    1      2    0.118     0.08    0.042   0.33   1.07
Allreduce            1    2      2     0.11    0.078    0.046   0.18   0.27
Allreduce            1    3      2    0.102    0.072    0.042   0.29   0.97
Barrier              1    0      3     51.9     17.3    0.015  61.86  99.44
Barrier              1    1      3    0.073   0.0457    0.016   0.29   0.91
Barrier              1    2      3     54.9     18.8    0.031  64.90  99.53
Barrier              1    3      3     1.56     1.02    0.035   6.20  20.68
Bcast                1    0      2    0.073   0.0535    0.034   0.13   0.20
Bcast                1    1      2    0.037    0.023    0.009   0.10   0.31
Bcast                1    2      2    0.084    0.046    0.008   0.11   0.16
Bcast                1    3      2     0.03   0.0275    0.025   0.11   0.37
Recv                 1    1      1     14.6     14.6     14.6  30.48  97.71
Recv                 1    3      1     11.6     11.6     11.6  23.37  77.98
Send                 1    0      1    0.013    0.013    0.013   0.02   0.02
Send                 1    2      1     0.02     0.02     0.02   0.02   0.04
Send                 1    *     32     54.9     4.34    0.008  51.69 100.00
---------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) -------------------
---------------------------------------------------------------------------
Name              Site Rank   Count       Max      Mean       Min       Sum
Allreduce            1    0       2       800       600       400      1200
Allreduce            1    1       2       800       600       400      1200
Allreduce            1    2       2       800       600       400      1200
Allreduce            1    3       2       800       600       400      1200
Bcast                1    0       2       800       600       400      1200
Bcast                1    1       2       800       600       400      1200
Bcast                1    2       2       800       600       400      1200
Bcast                1    3       2       800       600       400      1200
Send                 1    0       1       400       400       400       400
Send                 1    2       1       400       400       400       400
Send                 1    *      18       800     577.8       400  1.04e+04
---------------------------------------------------------------------------
@--- End of Report --------------------------------------------------------
```

# MPI Tracing with Jumpshot

- Sometimes we need to see the exact sequence of messages exchanged between processes.
- For this, we can enable MPI tracing by relinking our application.
- Once run, the application will produce a log file.
- We visualize that log using the Jumpshot tool.

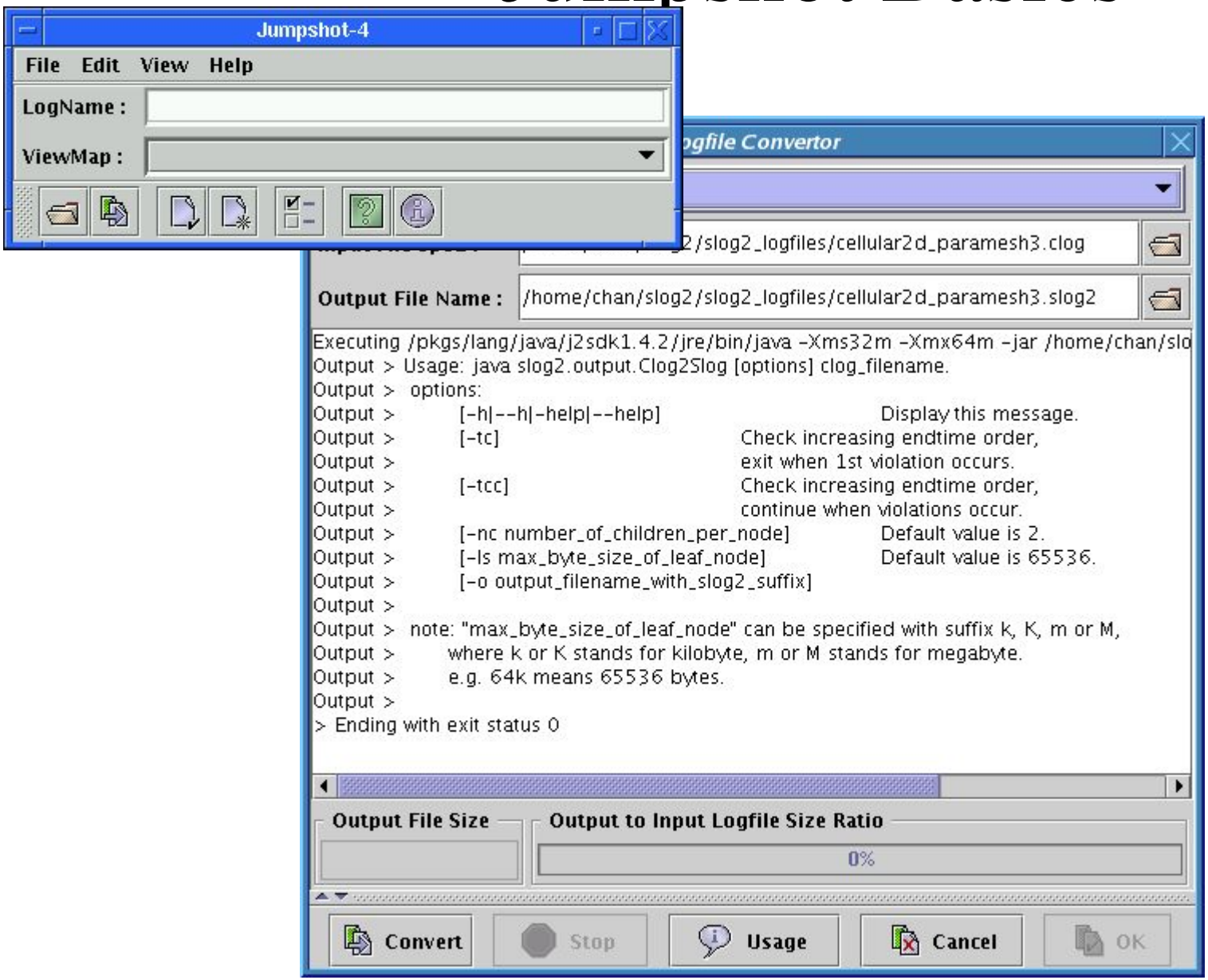
# MPI Tracing with Jumpshot (2)

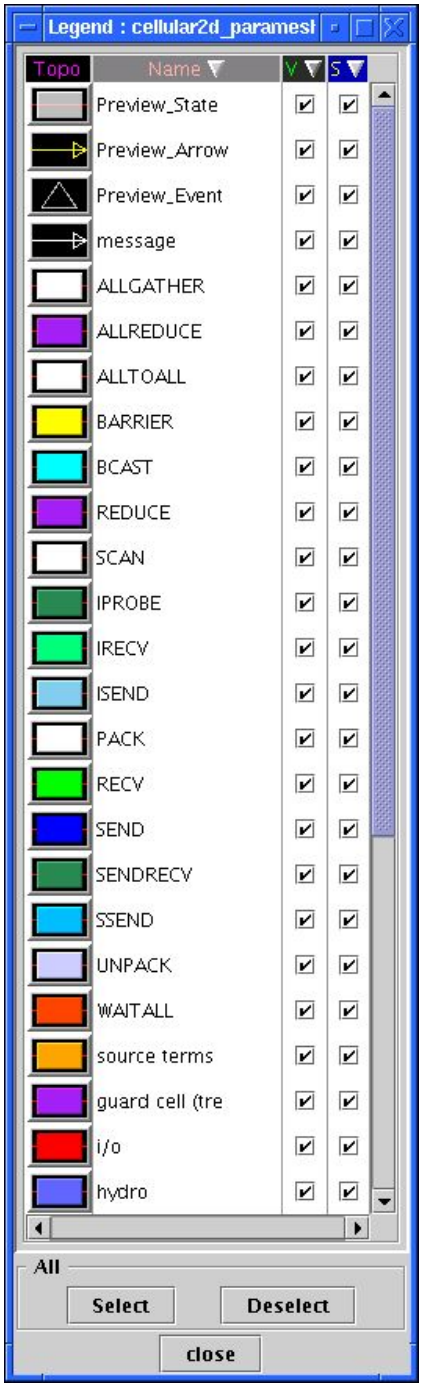
- Relink your codes after loading the module.
- After running, you will see a <app>.clog2
- (Optional) Translate the logfile to the slog2 scalable log format with clog2TOslog2.
  - Jumpshot will give you the option of doing this.
- Left click and drag zooms, right click and drag analyzes.

# Jumpshot Example

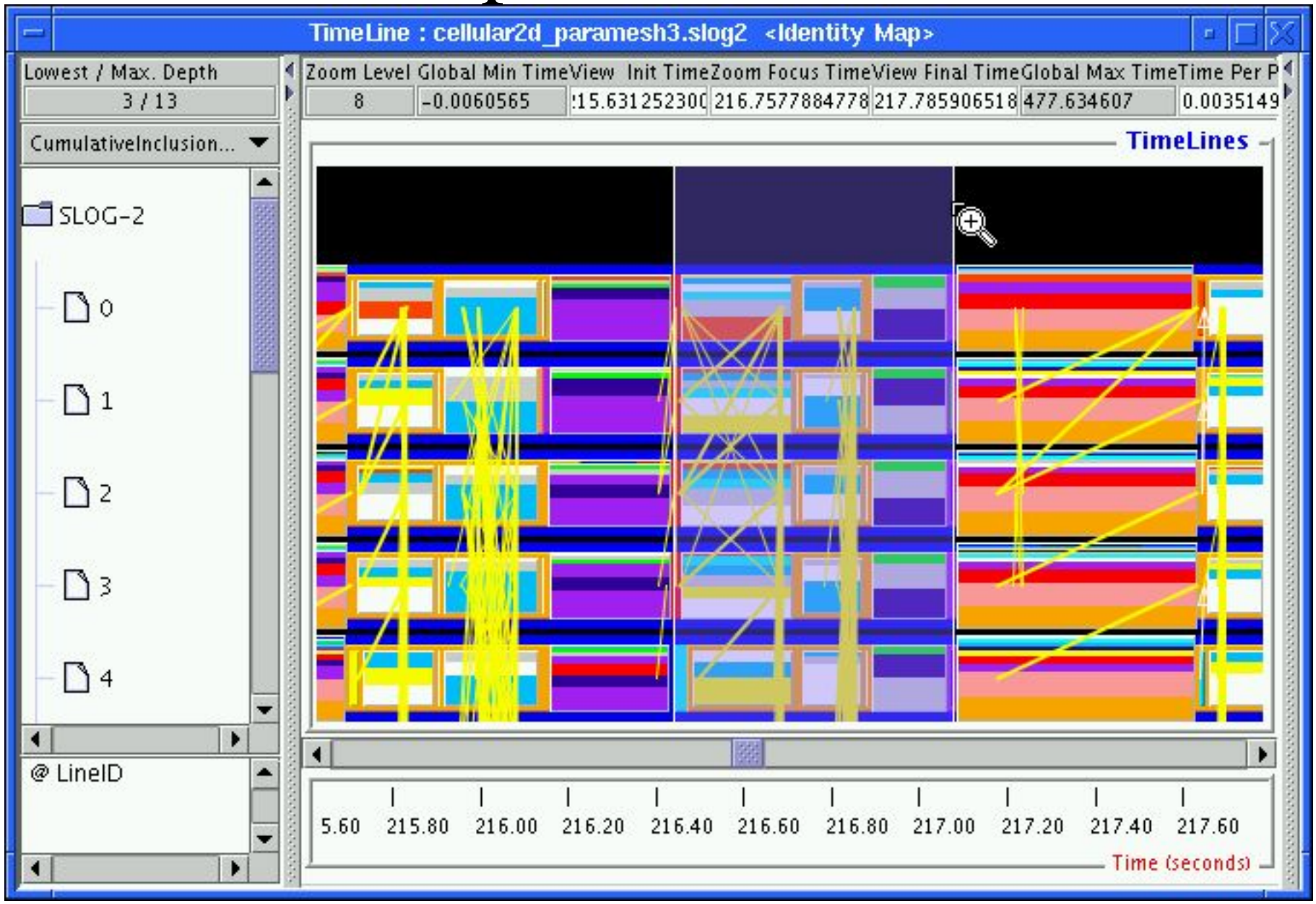
```
> module load jumpshot
> mpicc srtest.c -o srtest -llmpe -lmpe
> spattach -i -p2
> mpirun -np 2 -machinefile $SP_HOSTFILE
  $PWD/srtest
> clog2slog2 srtest.clog2
> jumpshot srtest.slog2
```

# Jumpshot Basics

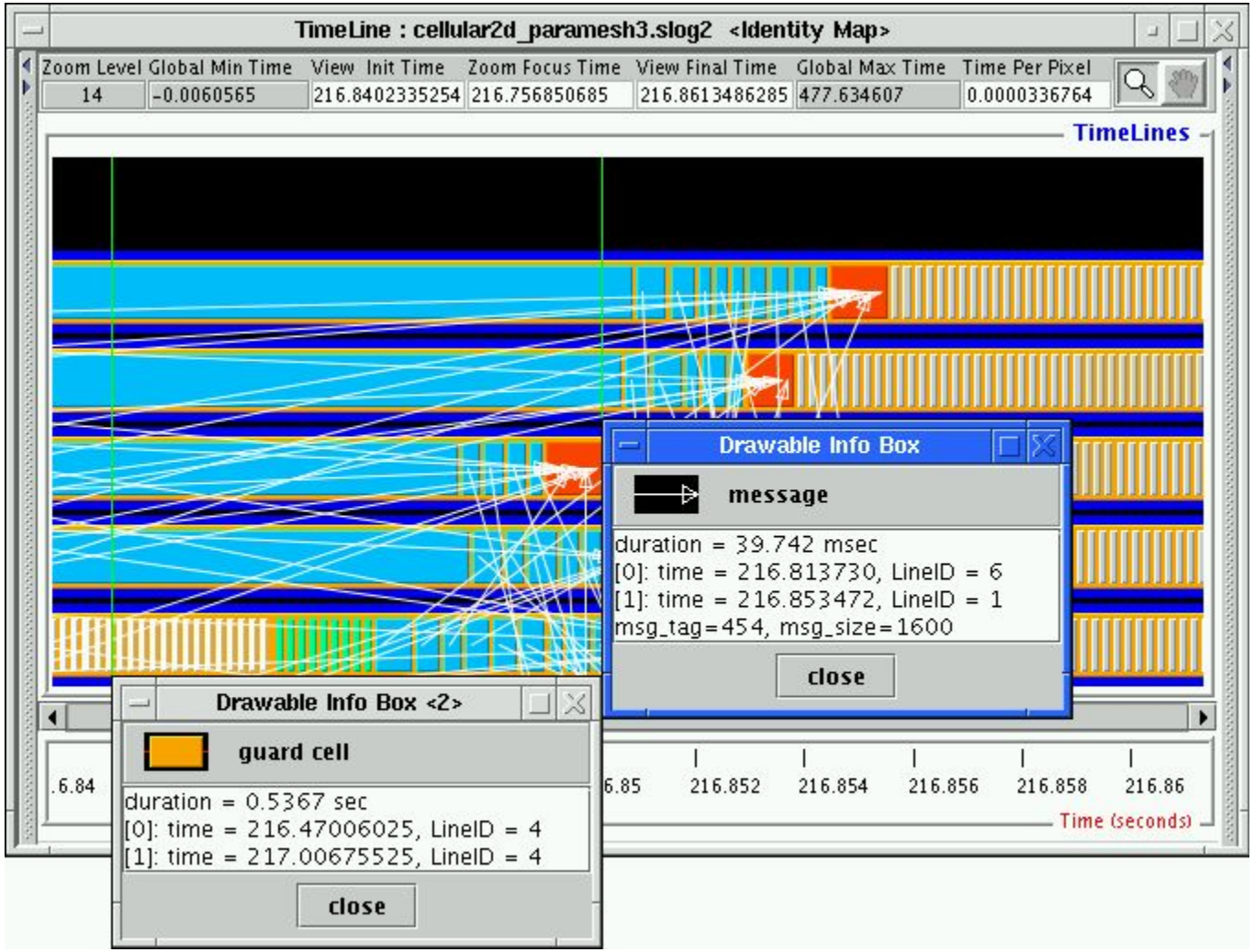


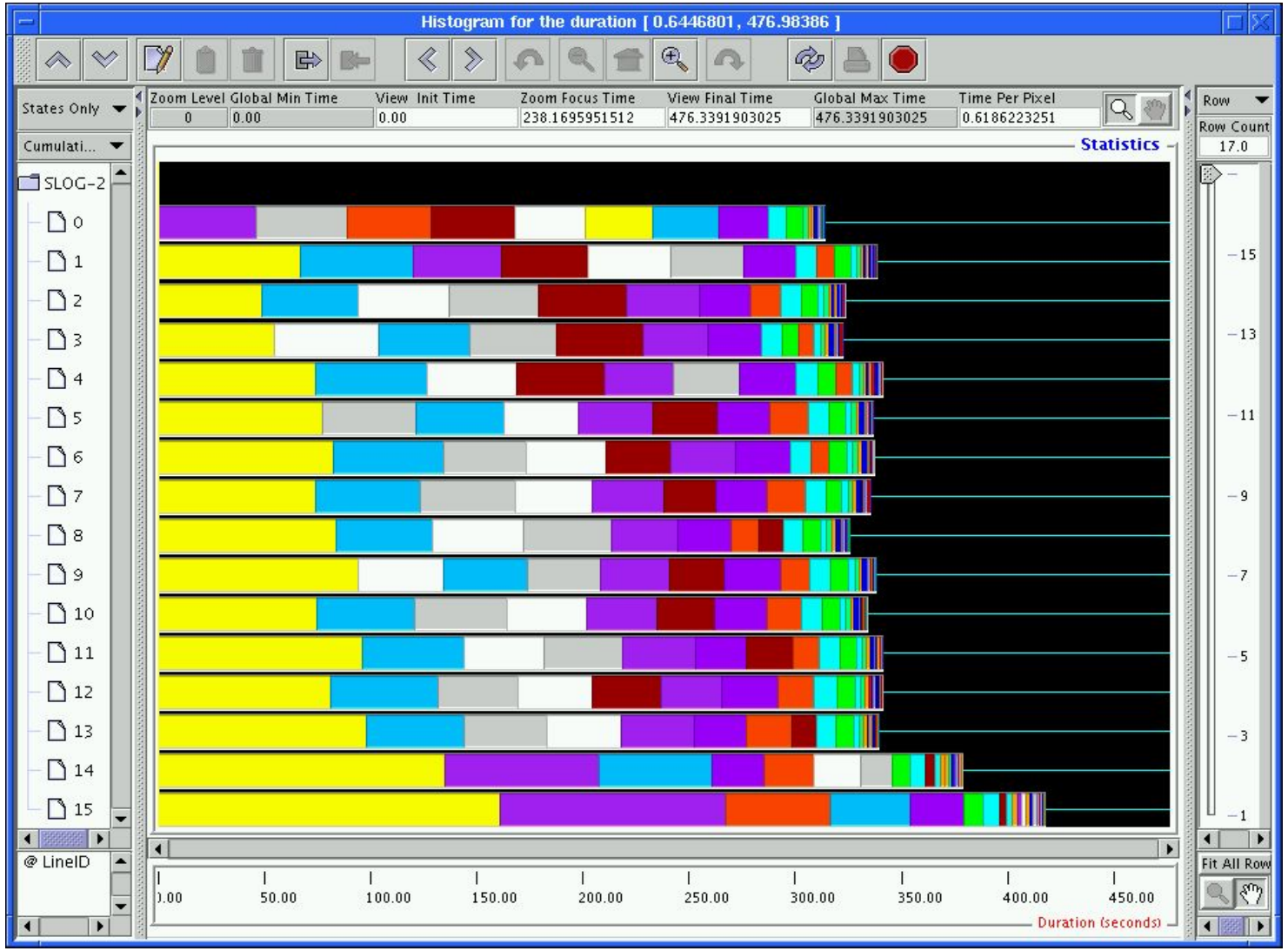
# Jumpshot Timeline



# Jumpshot Zoomed Timeline



# Jumpshot Histogram Window



# Some Other Performance Tools

- TAU (U. Oregon)
  - Source/dynamic instrumentation and tracing system
  - http://www.cs.uoregon.edu/research/tau/home.php
- KOJAK (Juelich, UTK)
  - Automated bottleneck analysis.
  - Instrumentation, tracing and analysis system for MPI, OpenMP and Performance Counters.
  - http://www.fz-juelich.de/zam/kojak/

# 5 Ways to Avoid Performance Problems: (1)

*Never, ever, write your own code unless you absolutely have to.*

- Libraries, libraries, libraries!
- Spend time to do the research, chances are you will find a package that suits your needs.
- Often you just need to do the glue that puts the application together.
- The 90/10 Rule! 90% of time is spent in 10% of code.

# 5 Ways to Avoid Performance Problems: (2)

*Never violate the usage model of your environment.*

- If something seems impossible to accomplish in your language or programming environment, you're probably doing something wrong.
- Consider such anomalies as:
  - Matlab in parallel on a cluster of machines.
  - High performance Java.
- There probably is a better way to do it, ask around.

# 5 Ways to Avoid Performance Problems: (3)

*Always let the compiler do the work.*

- The compiler is much better at optimizing most code than you are.
- Gains of 30-50% are reasonable common when the 'right' flags are thrown.
- Spend some time to read the manual and ask around.

# 5 Ways to Avoid Performance Problems: (4)

*Never use more data than absolutely necessary.*

- C: float vs. double.
- Fortran: REAL*4, REAL*8, REAL*16
- Only use 64-bit precision if you NEED it.
- A reduction in the amount of data the CPU needs ALWAYS translates to a increase in performance.
- Always keep in mind that the memory subsystem and the network are the ultimate bottlenecks.

# 5 Ways to Avoid Performance Problems: (5)

*Make friends with Computer Scientists*

- Learning even a little about modern computer architectures will result in much better code.
- 2 Challenges: Learn why the following statements are almost always horribly slow on modern CPUs when placed inside loops where Index is the loop variable.
- **1) Var = Var + DataArray[IndexArray[Index]]**
- **2) IF (VAR2 .EQ. I) THEN**
- **2) DOESN'T REALLY MATTER**
- **2) ENDIF**

# Questions?

- This talk:
  - http://www.cs.utk.edu/~mucci/latest/mucci_talks.html
  - http://www.cs.utk.edu/~mucci/latest/pubs/PDCOptClass.pdf
- PAPI Homepage:
  - http://icl.cs.utk.edu/papi
- For those here at KTH, many on the PDC staff are well versed in the art of performance. Use them!