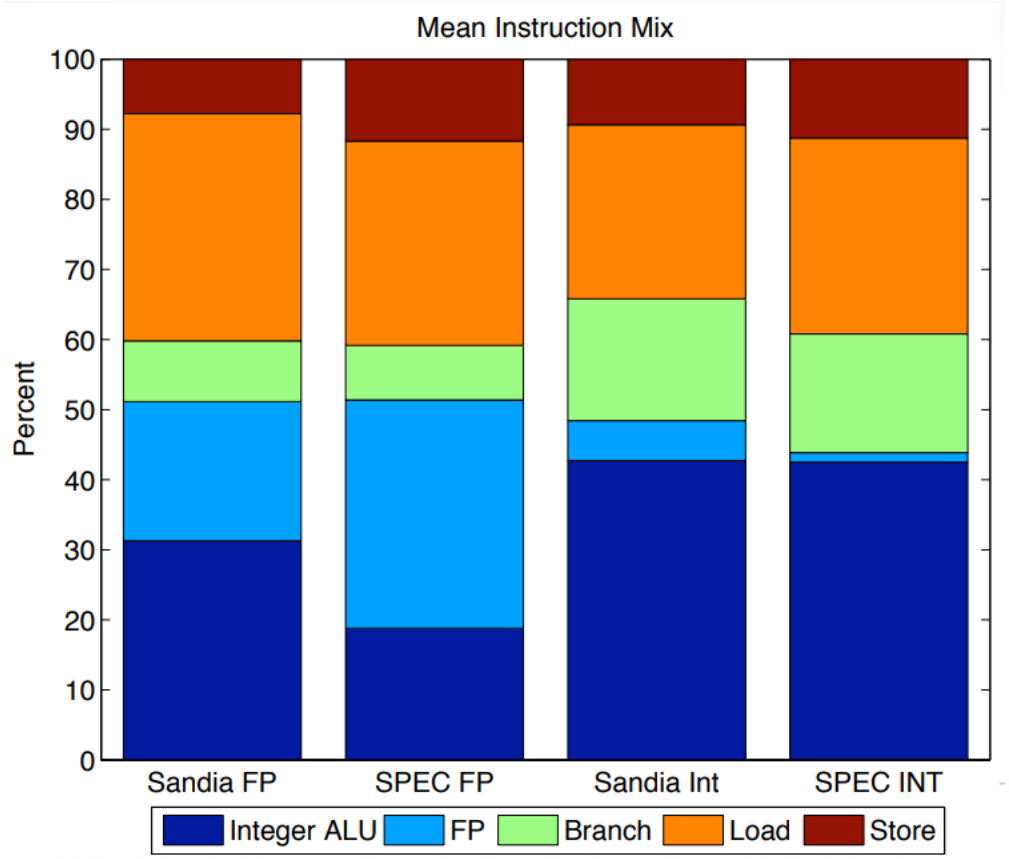
Profiling

Lecture 23

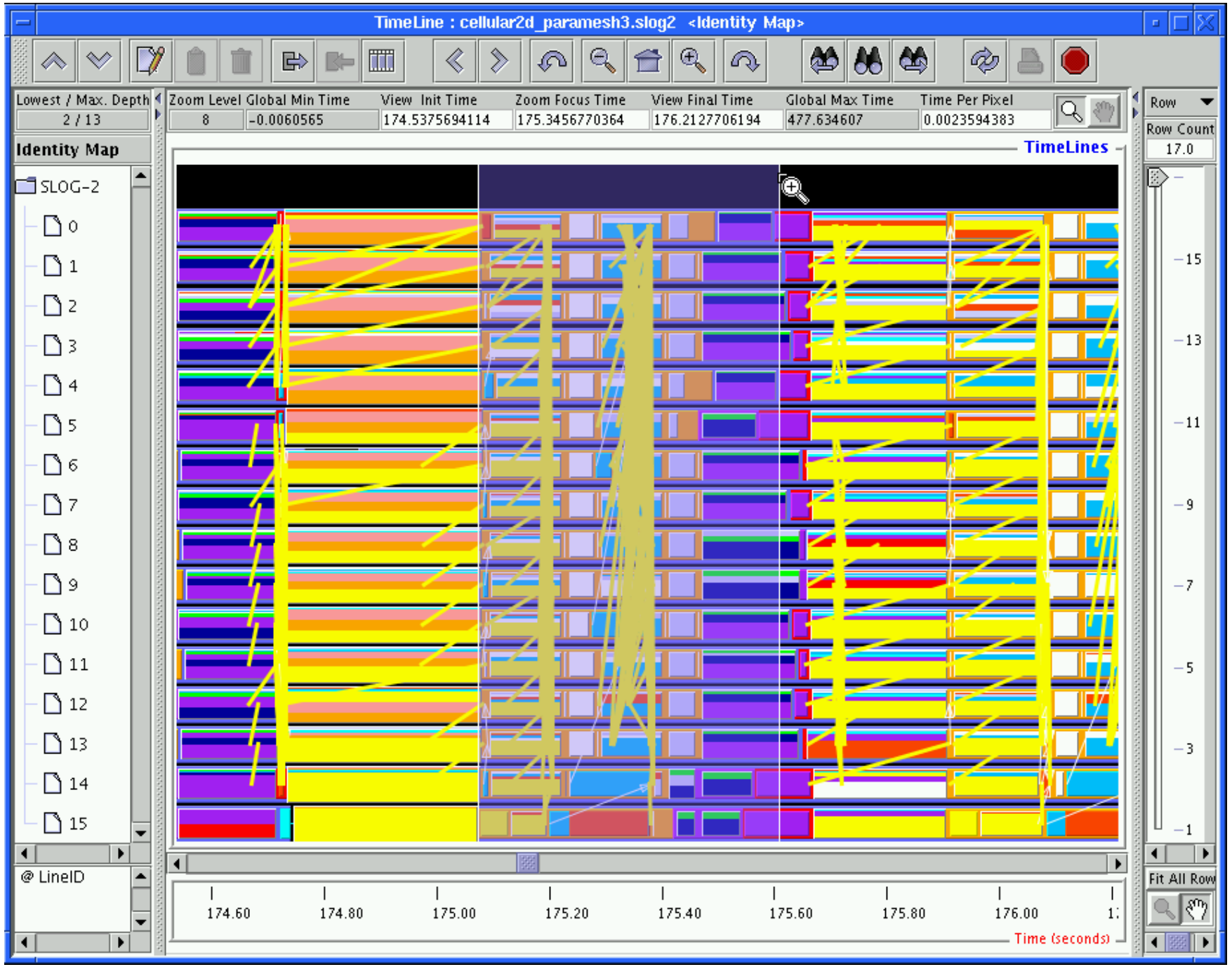
April 13, 2024

Analyzing Performance

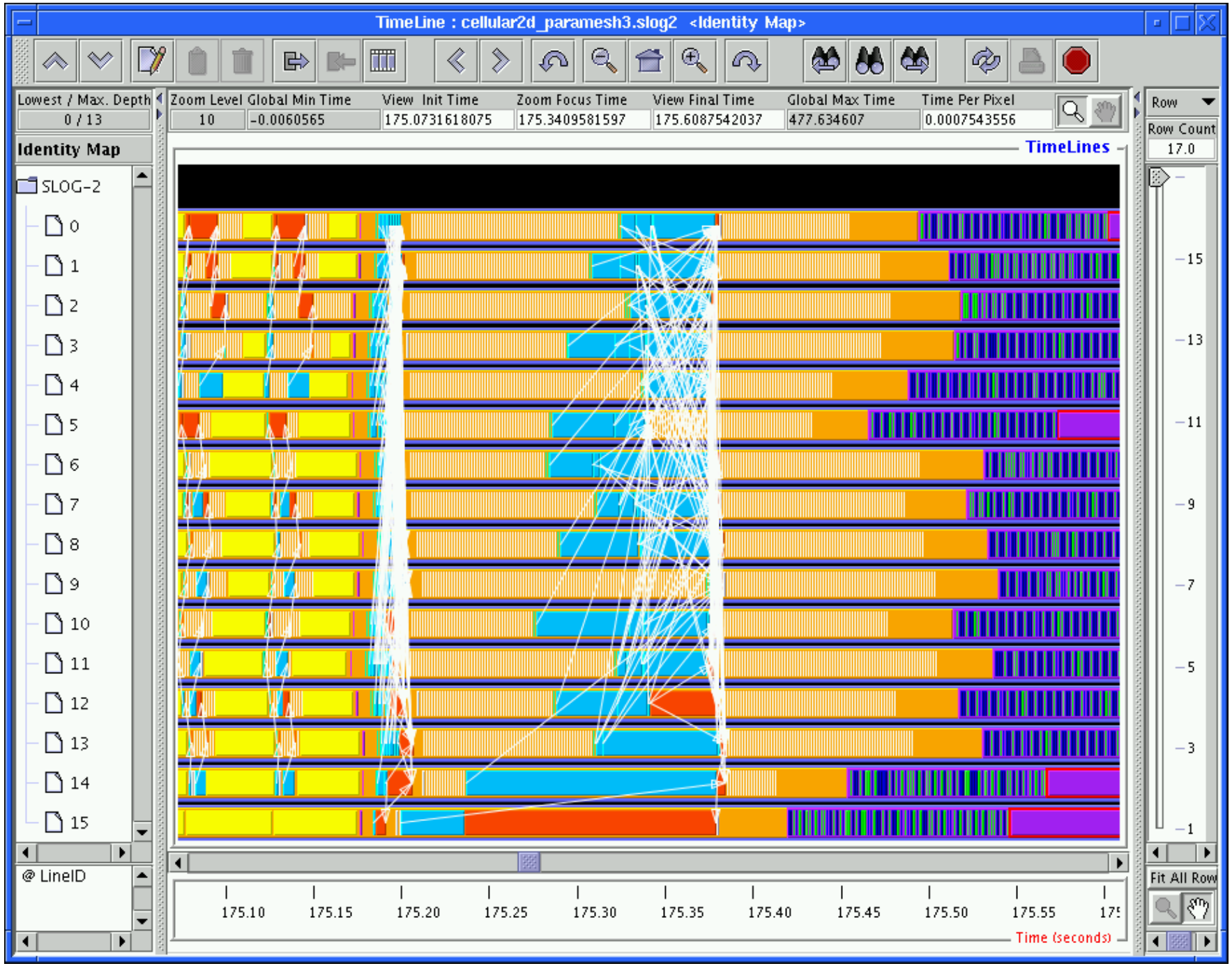
LINPACK Benchmark: solve a dense system of linear equations



Reference: “Introducing the Graph 500”, Richard C. Murphy et al., Cray User’s Group (CUG), May 5, 2010.



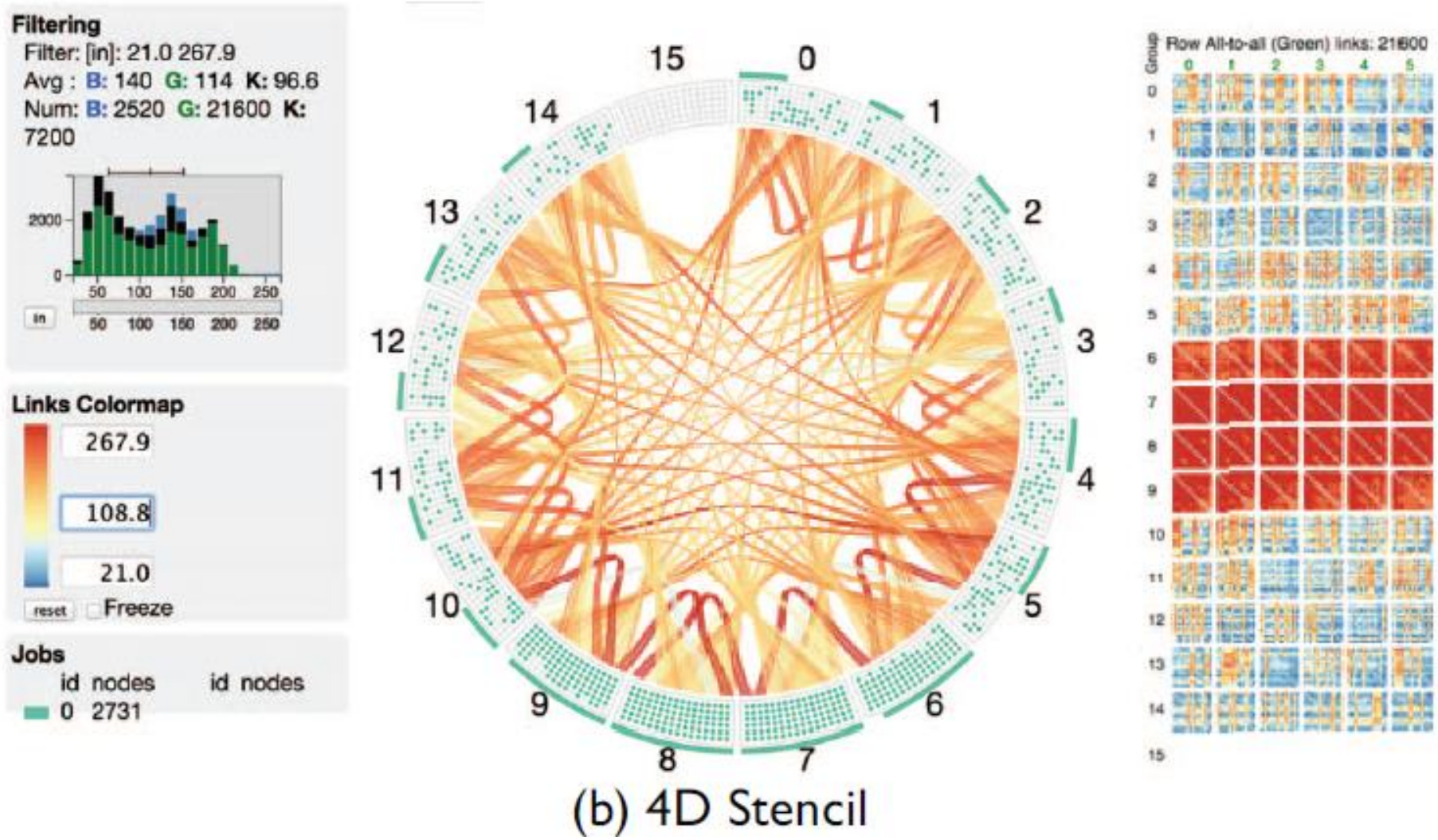
https://www.mcs.anl.gov/research/projects/perfvis/software/viewers/


TimeLine : cellular2d_paramesh3.slog2 <Identity Map>
Lowest / Max. Depth
0 / 13
Zoom Level
10
Global Min Time
-0.0060565
View Init Time
175.0731618075
Zoom Focus Time
175.3409581597
View Final Time
175.6087542037
Global Max Time
477.634607
Time Per Pixel
0.0007543556
Identity Map
SLOG-2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
@ LineID
TimeLines
Row
Row Count
17.0
15
13
11
9
7
5
3
1
Fit All Row
175.10
175.15
175.20
175.25
175.30
175.35
175.40
175.45
175.50
175.55
Time (seconds)

Performance Analysis

- Identify hot spots
- Identify where resources are being used inefficiently
- Difference between peak and average performance
- What matters is proportion of events
  - Floating point arithmetic vs. memory operations
- Performance analysis tools must collect and present relevant information on application performance in a scalable manner
  - Enable developers to easily identify and determine the causes of performance bottlenecks
  - Crucial to be not immersed in sea of data

Example (Custom Analysis)



(b) 4D Stencil

Profiling

- Elapsed time for the program and procedures
- Collection of statistical summaries of various performance metrics broken down by program entities such as routines and nested loops
- Performance metrics include time as well as hardware counter metrics such as operation counts and cache and memory event counts
- Profiling can identify regions of a program that are consuming the most resources

Tracing

- Timestamp attached with each event
- Collection of a timestamped sequence of events
  - Entering and exiting program regions
  - Sending and receiving messages.
- Can help identify the causes of performance problems
- Event traces record the temporal and spatial relationships between individual runtime events
- Cons: Huge space requirement

Performance Counters

Operation count data - Hardware

- Number of cache misses
- Number of loads/stores
- Main memory stalls

Operation count data - Network

- Number of packets sent and received
- Number of packets waiting to be sent

Profiling

- Source instrumentation
  - Know the location of bottleneck a priori for source code instrumentation
  - Huge endeavor to instrument every procedure in a large application
- Compile-time instrumentation
- Run-time instrumentation

Source Code Instrumentation

start = MPI_Wtime()

…

…

…

end = MPI_Wtime()

Source Code Instrumentation

```
/* Initialize the PAPI library and get the number of counters available */
if ((num_hwcntrs = PAPI_num_counters()) <= PAPI_OK)
        handle_error(1);


/* Start counting events */
if (PAPI_start_counters(Events, num_hwcntrs) != PAPI_OK)
        handle_error(1);
```

Compile-time/Run-time Instrumentation

- Code recompilation
  - -L<path-to-lib> -l<libname>
- Code recompilation not required
  - LD_PRELOAD, LD_LIBRARY_PATH..

Some Factors

Scalability

- Large number of processes
- Many functions
- Different inputs

Data Format

- Some common log formats
- Most are tool-specific

Presentation

- Present data in a hierarchical fashion
- Prioritize – present first what happens to be important

THE TAU PARALLEL PERFORMANCE SYSTEM

Sameer S. Shende
Allen D. Malony

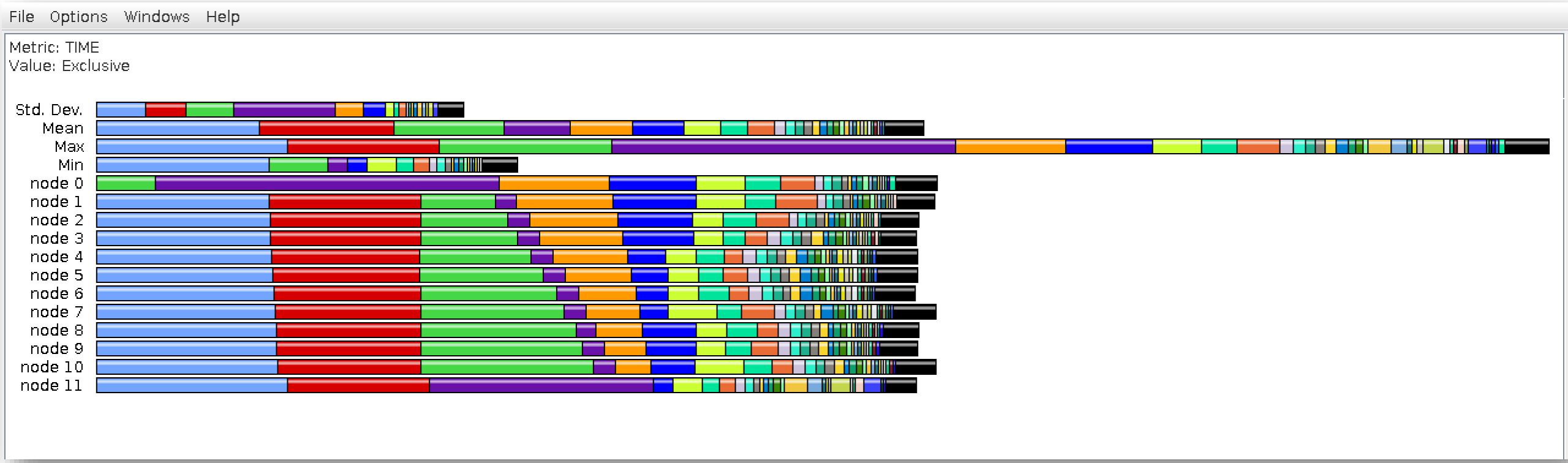
Tuning and Analysis Utilities (TAU)

- Supports tracing and profiling
- Supports different automated instrumentation options
  - Source code
  - Compiler based
- Uses call-path profiling to determine relevant routines for tracing
- Data management framework
  - Relational database
  - Graphical display
- Search traces to automatically identify inefficiency

TAU Profiling

- Performance measurements for functions, basic blocks, statements.
- Exclusive and inclusive time spent in functions (ns)
- #times each function was called, mean inclusive time per call, etc.

TAU Snapshot



HPCTOOLKIT: Tools for performance analysis of optimized parallel programs

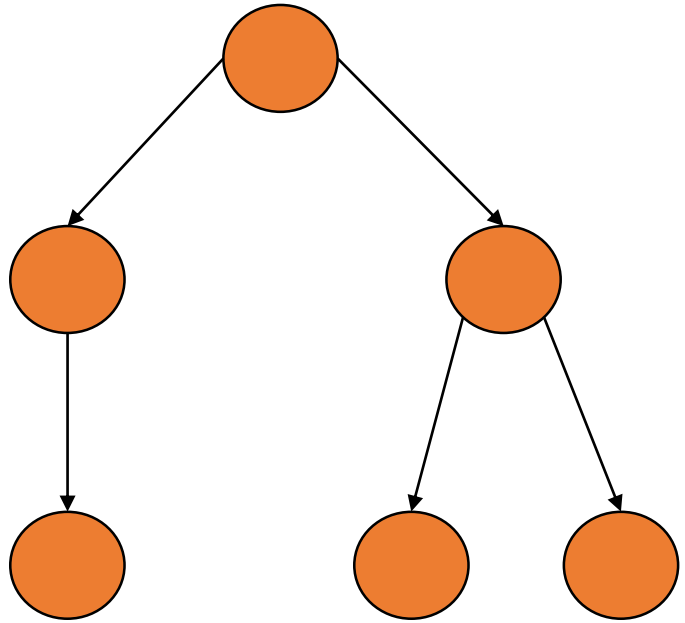
L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey , N. R. Tallent

HPCToolkit Overview

- Does not use instrumentation
  - gprof dilates by 82%
  - Intel’s Vtune dilates by 31x
- Uses statistical sampling
- Supports call path profiling: attribute cost incurred to different contexts in which a procedure is called
- Supports collection, correlation of multiple performance metrics

Calling Context Tree

- Each vertex in a tree represents single procedure activation
  - Metrics recorded for that invocation
- Sample count recorded at each node
- Space-inefficient
- Tracing – Record the timestamp as well



Analysis

Correlation of performance metrics

- Mapping between object code and source code (“recovering program structure”)

Compute useful metrics

- Knowing where the most cycles are spent
- Derived metrics

Scalability

- Combine call path profiles with program structure
- Estimate idleness, parallel efficiency etc.

Hands-on

- Login to HPC2010
- wget www.cse.iitk.ac.in/users/cs633/13-04.tar.gz
- tar –xzf 13-04.tar.gz
- qsub -I
- cd 13-04

NAS Parallel Benchmarks

https://www.nas.nasa.gov/publications/npb.html

- Five kernels
  - IS - Integer Sort, random memory access
  - EP - Embarrassingly Parallel
  - CG - Conjugate Gradient, irregular memory access and communication
  - MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT - discrete 3D fast Fourier Transform, all-to-all communication
- Three pseudo applications
  - BT - Block Tri-diagonal solver
  - SP - Scalar Penta-diagonal solver
  - LU - Lower-Upper Gauss-Seidel solver

Profiling NPB with mpiP profiler

- https://github.com/LLNL/mpiP
- cd NPB

- Small problem, small core count
- qsub sub.A.4.sh
- Small problem, higher core count
- qsub sub.A.16.sh
- Large problem, higher core count
- qsub sub.C.16.sh

mpiP Profiles – Strong scaling

You will obtain *.mpiP files after the runs (for each mpirun)

```
18 ---------------------------------------
19 @--- MPI Time (seconds) ---------------
20 ---------------------------------------
21 Task    AppTime    MPITime     MPI%
22    0       18.4      0.564     3.06
23    1       18.4      0.502     2.72
24    2       18.4      0.329     1.78
25    3       18.4      0.499     2.71
26    *       73.8       1.89     2.57
27 ---------------------------------------
```

sp.A.4.<>.mpiP

sp.A.16.<>.mpiP

```
30 ---------------------------------------
31 @--- MPI Time (seconds) ---------------
32 ---------------------------------------
33 Task    AppTime    MPITime     MPI%
34    0       5.54      0.534     9.64
35    1       5.54      0.556    10.04
36    2       5.54      0.544     9.82
37    3       5.54       0.55     9.92
38    4       5.54      0.524     9.45
39    5       5.54      0.593    10.69
40    6       5.54      0.606    10.93
41    7       5.54      0.584    10.53
42    8       5.54      0.538     9.71
43    9       5.54      0.601    10.84
44   10       5.54      0.572    10.32
45   11       5.54      0.542     9.77
46   12       5.54      0.573    10.34
47   13       5.54       0.63    11.37
48   14       5.54      0.637    11.50
49   15       5.54      0.614    11.08
50    *       88.7        9.2    10.37
51 ---------------------------------------
```

Profiling using IPM

- Integrated Performance Monitoring (IPM) profiler
- http://ipm-hpc.sourceforge.net/userguide.html

Using IPM

- Edit Makefile (-L<library path> -lipm)
- export LD_LIBRARY_PATH=<library path>:$LD_LIBRARY_PATH
- export IPM_REPORT=full
- export IPM_LOG=full

Profiling Intel MPI Benchmark using IPM

- cd imb_run
- qsub subR.sh
- qsub subG.sh
- Increase process count and compare

Parsing IPM Profiles

- Transfer the xml file and bin/ipm_parse, bin/ipm_key_mpi to your local system
- Install ploticus in your local system
- $IPM_HOME/bin/ipm_parse –html <xml file name>
- firefox <directory>/index.html