Par Lab Parallel Boot Camp

Performance Tools

Karl Fuerlinger

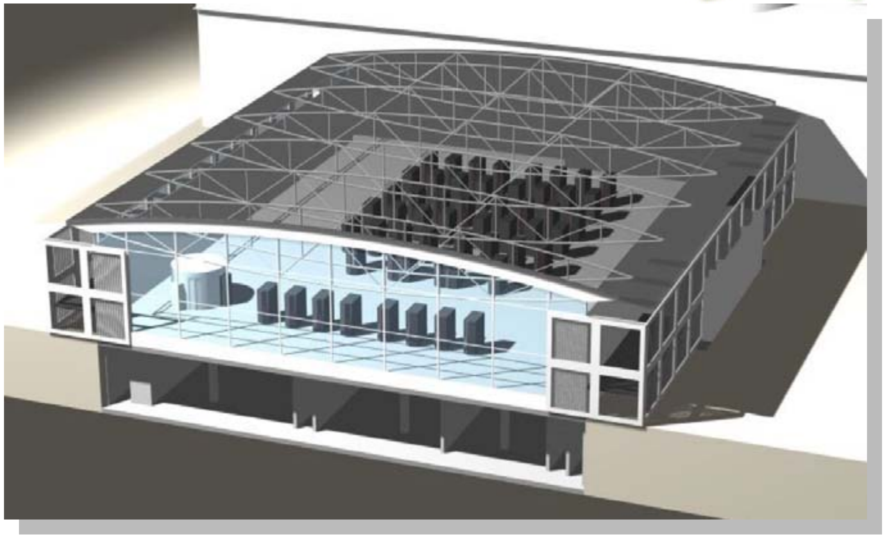
fuerling@eecs.berkeley.edu

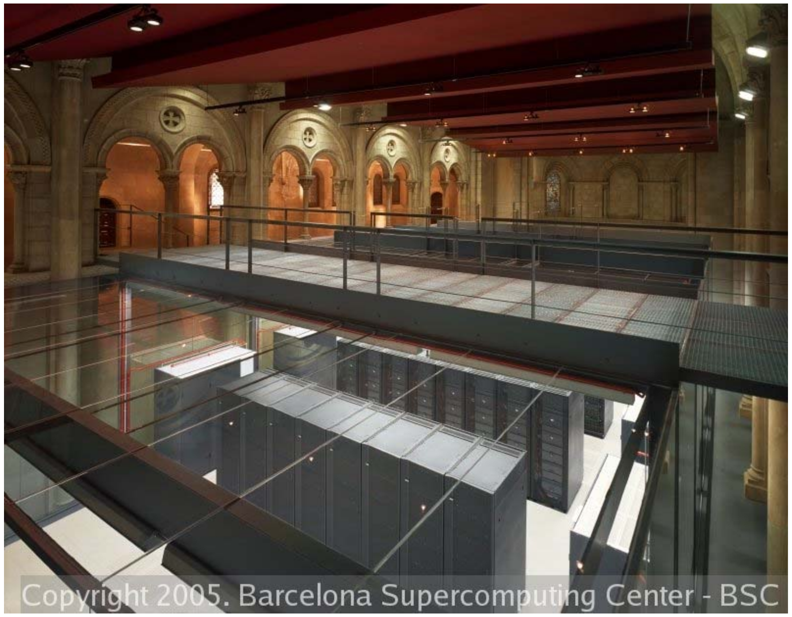
Outline

- Motivation
- Concepts and Definitions
  - Instrumentation, monitoring, analysis
- Specific examples and their functionality
  - PAPI – access to hardware performance counters
  - ompP – profiling OpenMP code
  - IPM – monitoring message passing applications

Motivation

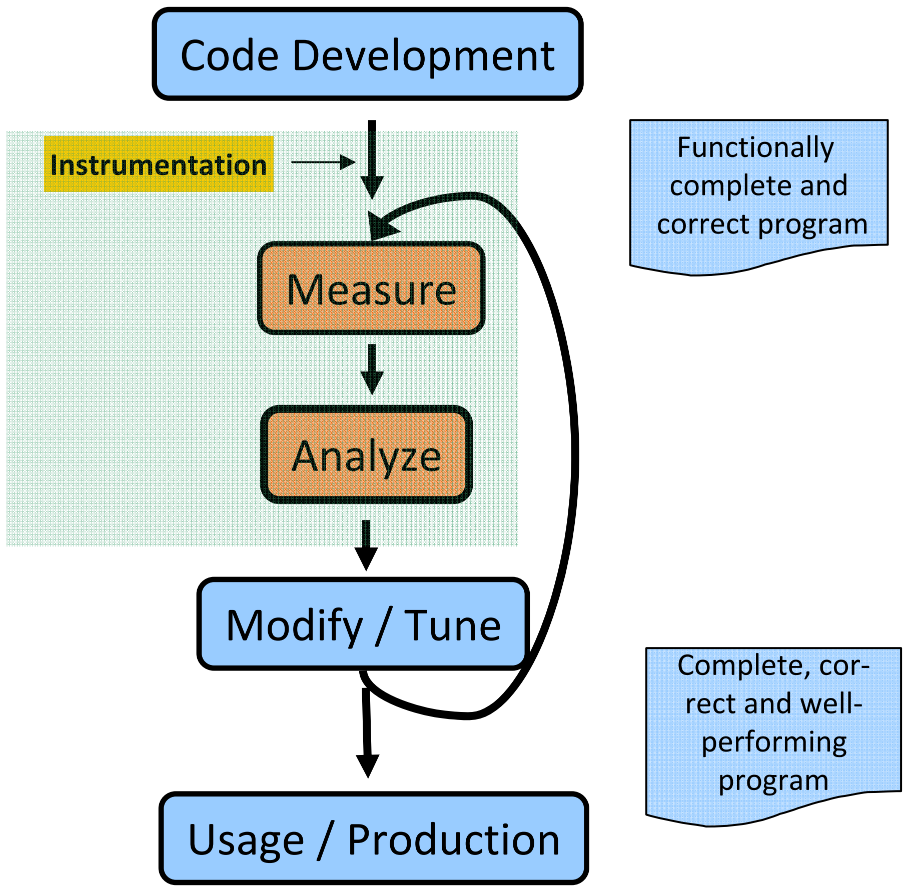
- Performance analysis is important
  - For HPC: computer systems are large investments
    - » Procurement: O($40 Mio)
    - » Operational costs: ~$5 Mio per year
    - » Power: 1 MW/year ~$1 Mio
  - Goals:
    - » Solve **larger** problems (new science)
    - » Solve problems **faster** (turn-around time)
    - » Improve **error** bounds on solutions (confidence)
- Same is true on smaller scale, down to a handheld devices as well: Parallelism enables new kinds of applications but need to take full advantage of resources





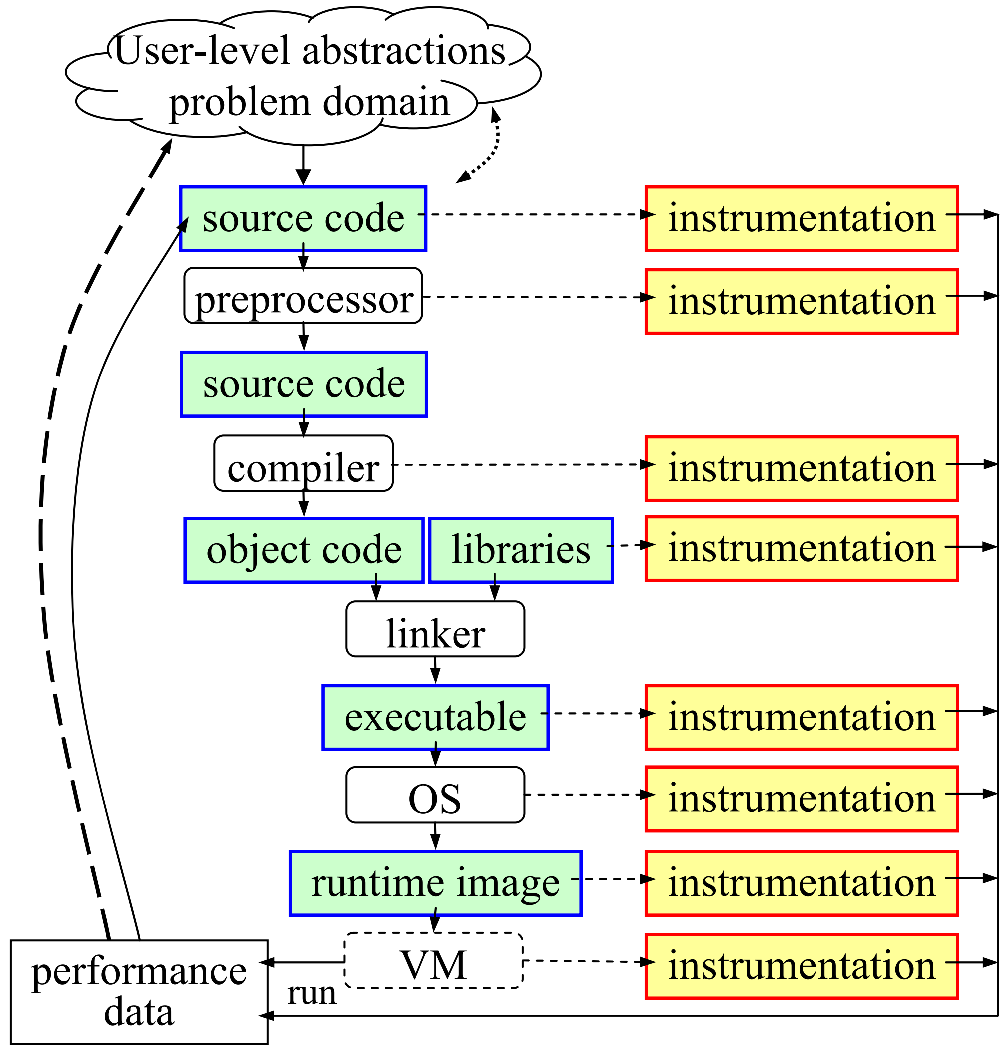
Concepts and Definitions

- The typical performance optimization cycle



Instrumentation

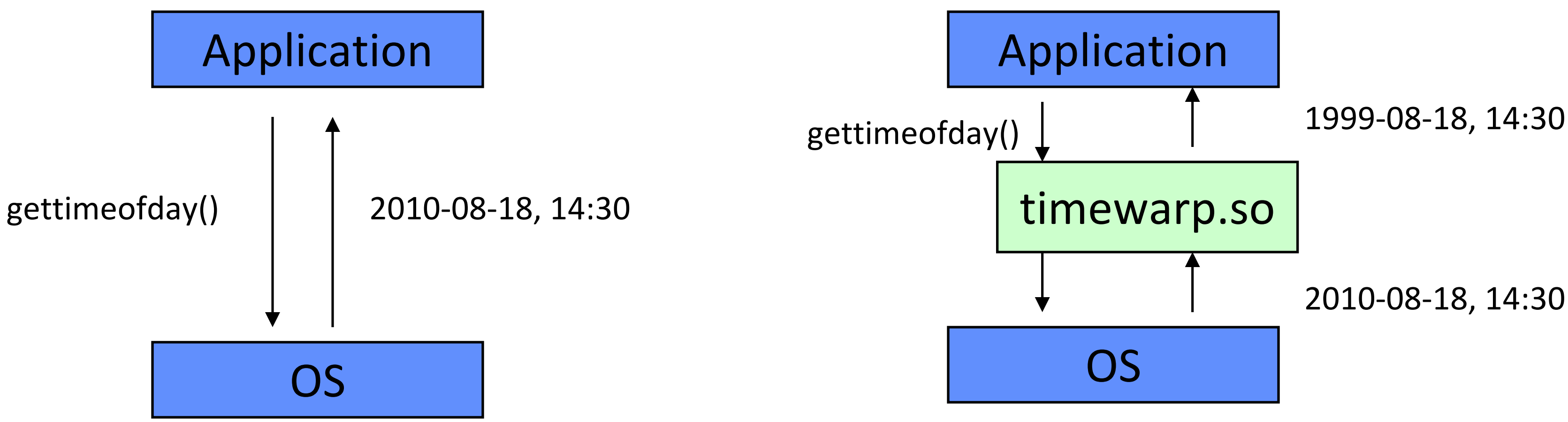
- **Instrumentation** := adding measurement probes to the code in order to observe its execution
- Can be done on several levels and different techniques for different levels
- Different overheads and levels of accuracy with each technique
- No application instrumentation needed: run in a simulator. E.g., Valgrind, SIMICS, etc. but slowdown and scalability are issues



Instrumentation – Examples (1)

- Dynamic Library Interposition
  - Standard technique for dynamically linked executables
  - No changes to the application required

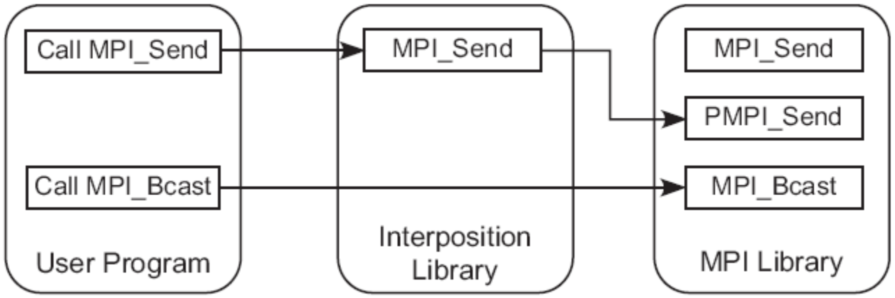
- LD_PRELOAD=timewarp.so ./myapp



- Used in practice for MPI, File-I/O, GPU (CUDA) monitoring

Instrumentation – Examples (2)

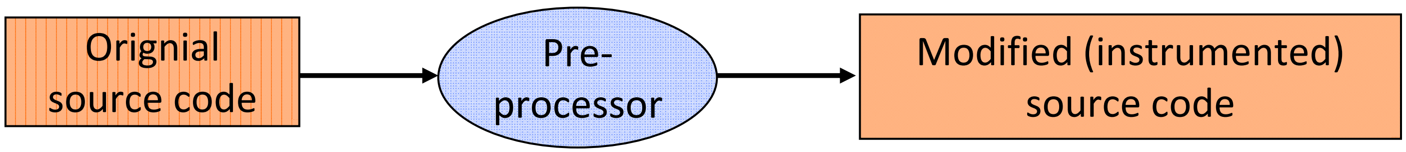
- MPI Library Instrumentation:



- MPI library interposition
  - All functions are available under two names: **MPI_Xxx** and **PMPI_Xxx**,
  - **MPI_Xxx** symbols are **weak**, can be over-written by interposition library
  - Measurement code in the interposition library measures begin, end, transmitted data, etc… and calls corresponding PMPI routine.
  - Not all MPI functions need to be implemented in the interpostion library
  - Works for statically linked applications too

Instrumentation – Examples (3)

- Preprocessor Instrumentation
  - Example: Instrumenting OpenMP constructs with Opari
  - Preprocessor operation



- Example: Instrumentation of a parallel region

```
POMP_Parallel_fork [master]
#pragma omp parallel {
  POMP_Parallel_begin [team]
   /* user code in parallel region */
   /* user code in parallel region */
}
    POMP_Barrier_enter [team]
    #pragma omp barrier
    POMP_Barrier_exit [team]
  POMP_Parallel_end [team]
}
POMP_Parallel_join [master]
```

This approach is used for OpenMP instrumentation by most vendor-independent tools. Examples: TAU/Kojak/Scalasca/ompP

Instrumentation added by Opari

Instrumentation – Examples (4)

- Source code instrumentation
  - **User-added** time measurement, etc. (e.g., **`printf`**(), **`gettimeofday`**())
  - **Think twice** before you roll your own solution, many **tools** expose mechanisms for source code instrumentation in addition to automatic instrumentation facilities they offer
  - Instrument program phases:
    - » Initialization
    - » main loop iteration 1,2,3,4,...
    - » data post-processing
  - Pragma and pre-processor based, e.g., Opari

```
#pragma pomp inst begin(foo)
// application code
#pragma pomp inst end(foo)
```

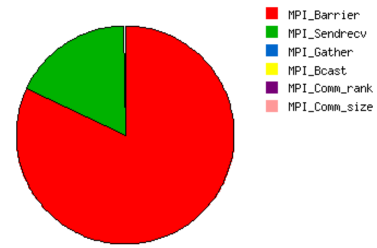
  - MPI_Pcontrol based, e.g., IPM

```
MPI_Pcontrol(1,"name");
// application code
MPI_Pcontrol(-1,"name");
```

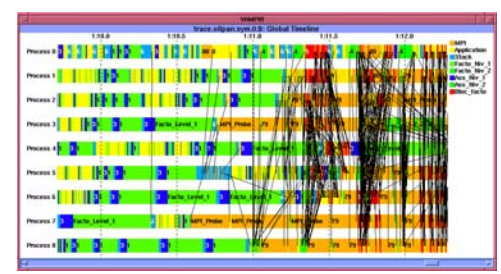
Measurement

- Profiling vs. Tracing

- Profiling
  - Summary statistics of performance metrics
    - » Number of times a routine was invoked
    - » Exclusive, inclusive time
    - » Hardware performance statistics
    - » Number of child routines invoked, etc.
    - » Call tree, call graph



- Tracing
  - Record when and where events took place along a global timeline
    - » Time-stamped log of events
    - » Large volume of performance data
    - » Individual sends, receives are tracked



Measurement: Profiling

- Profiling
  - Helps to expose performance bottlenecks and hotspots
  - 80/20 –rule or *Pareto principle:* often 80% of the execution time in 20% of your application
  - Optimize what matters, don't waste time optimizing things that have negligible overall influence on performance
- Implementation
  - **Sampling**: periodic OS interrupts or hardware counter traps
    - » Build a histogram of sampled program counter (PC) values
    - » Hotspots will show up as regions with many hits
    - » Examples gprof, HPCtoolkit
  - **Measurement**: direct insertion of measurement code
    - » Measure at start and end of regions of interests, compute difference

Profiling: Inclusive vs. Exclusive Time

```
int main( )             /* takes 100 secs */
{
   f1();                /* takes 20 secs */
   /* other work */
   f2();                /* takes 50 secs */
   f1();                /* takes 20 secs */
   /* other work */
}
```

- **Inclusive** time for **main**
  - **100** sec.
- **Exclusive** time for **main**
  - 100-20-50-20=**10** sec.
- Exclusive time sometimes called “self” time
- Similar definitions for inclusive/exclusive time for f1() and f2()
- Similar for other metrics, such as hardware performance counters, etc

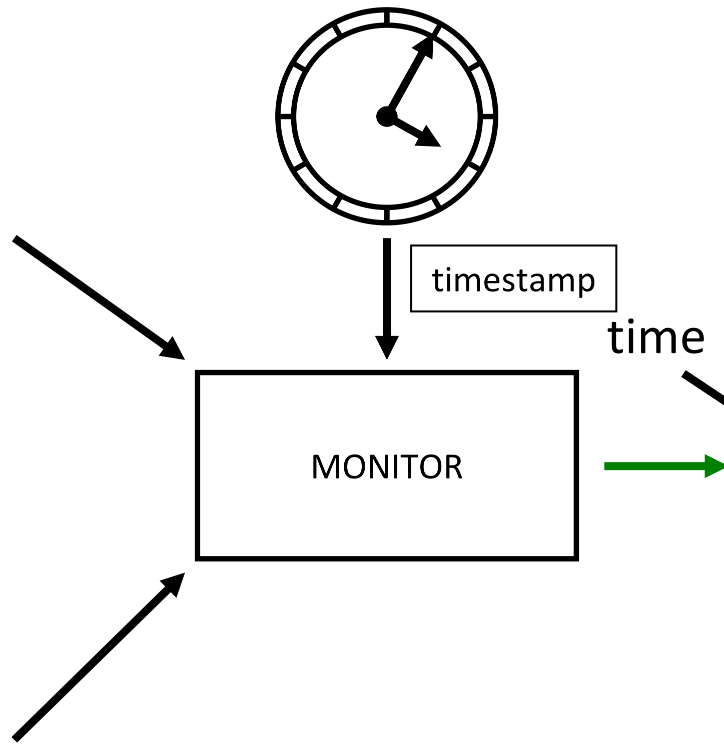
Tracing Example: Instrumentation, Monitor, Trace

Process A:

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

Process B:

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



Event definitions

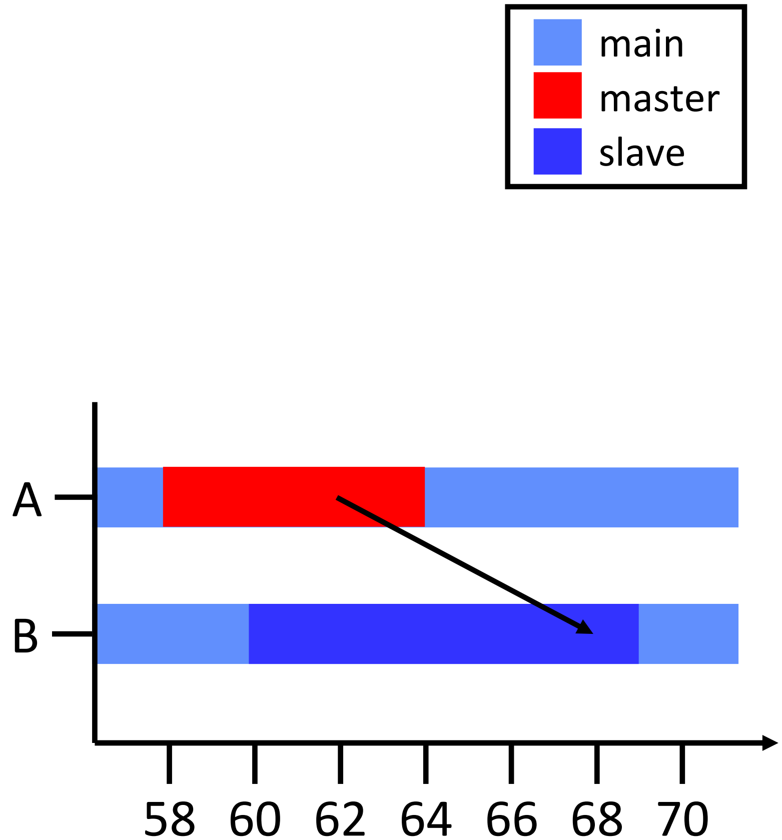
1	master
2	slave
3	...

time	location	event	context
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Tracing: Timeline Visualization

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
...			



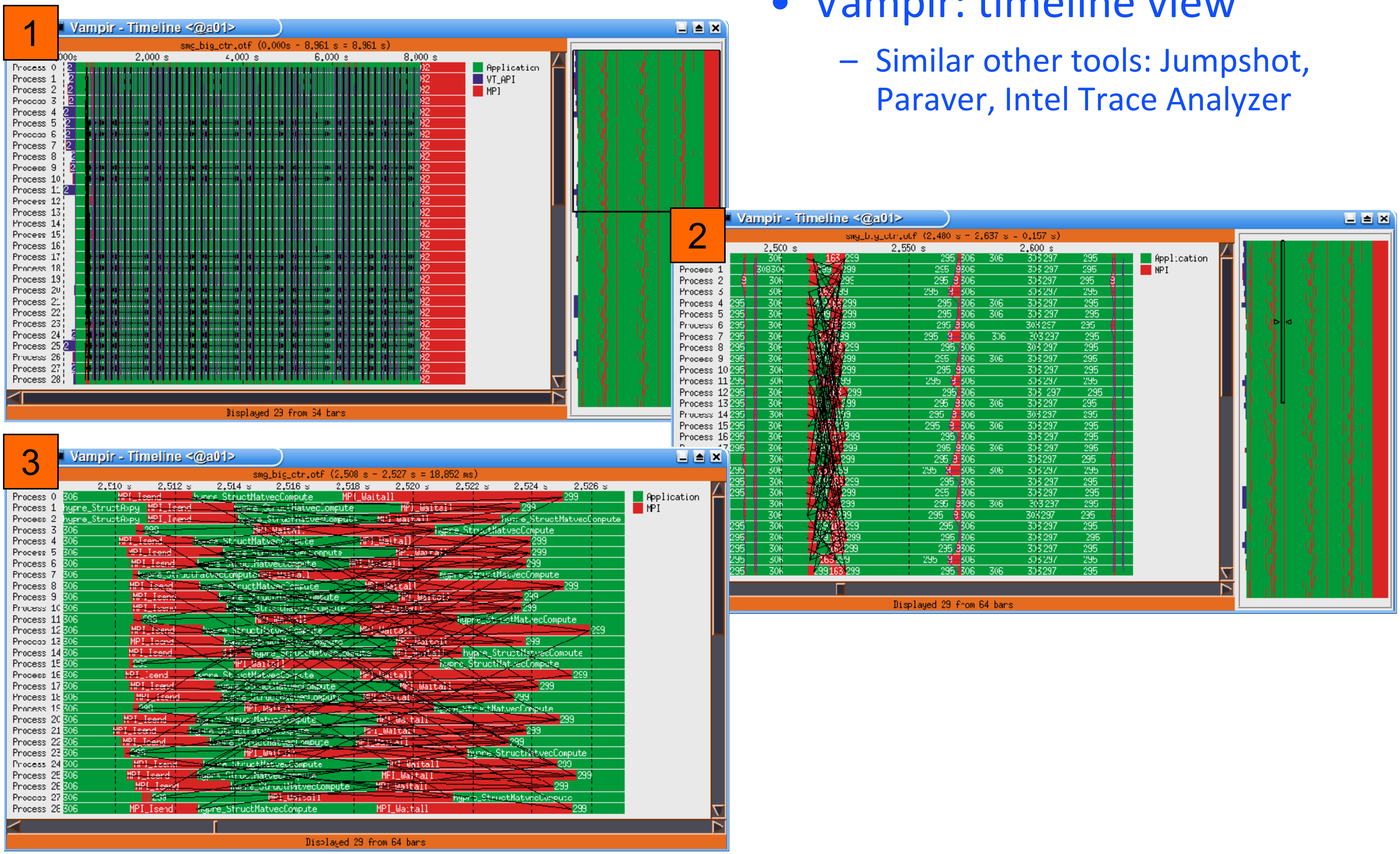
Performance Data Analysis

- Draw conclusions from measured performance data

- Manual analysis
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling

- Automated analysis
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca, Periscope

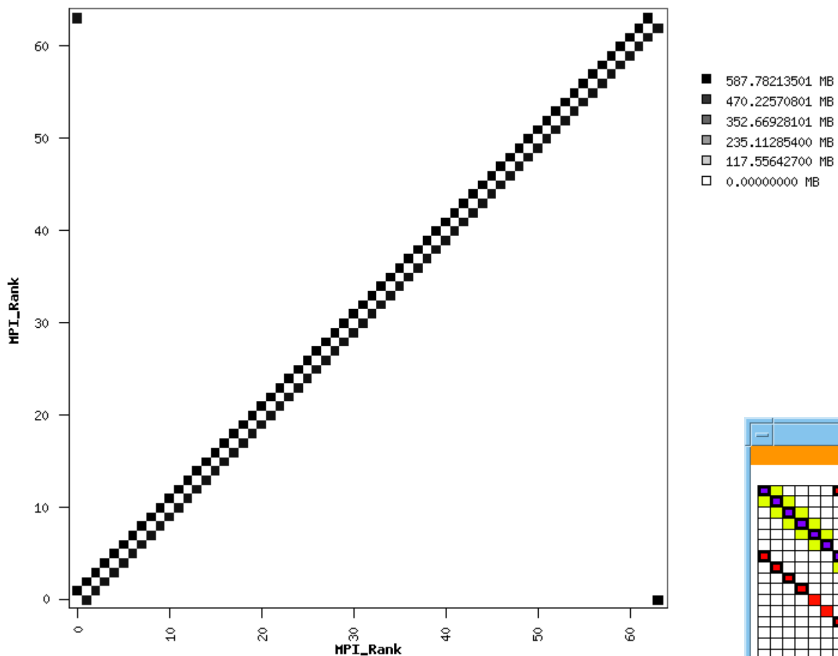
Trace File Visualization

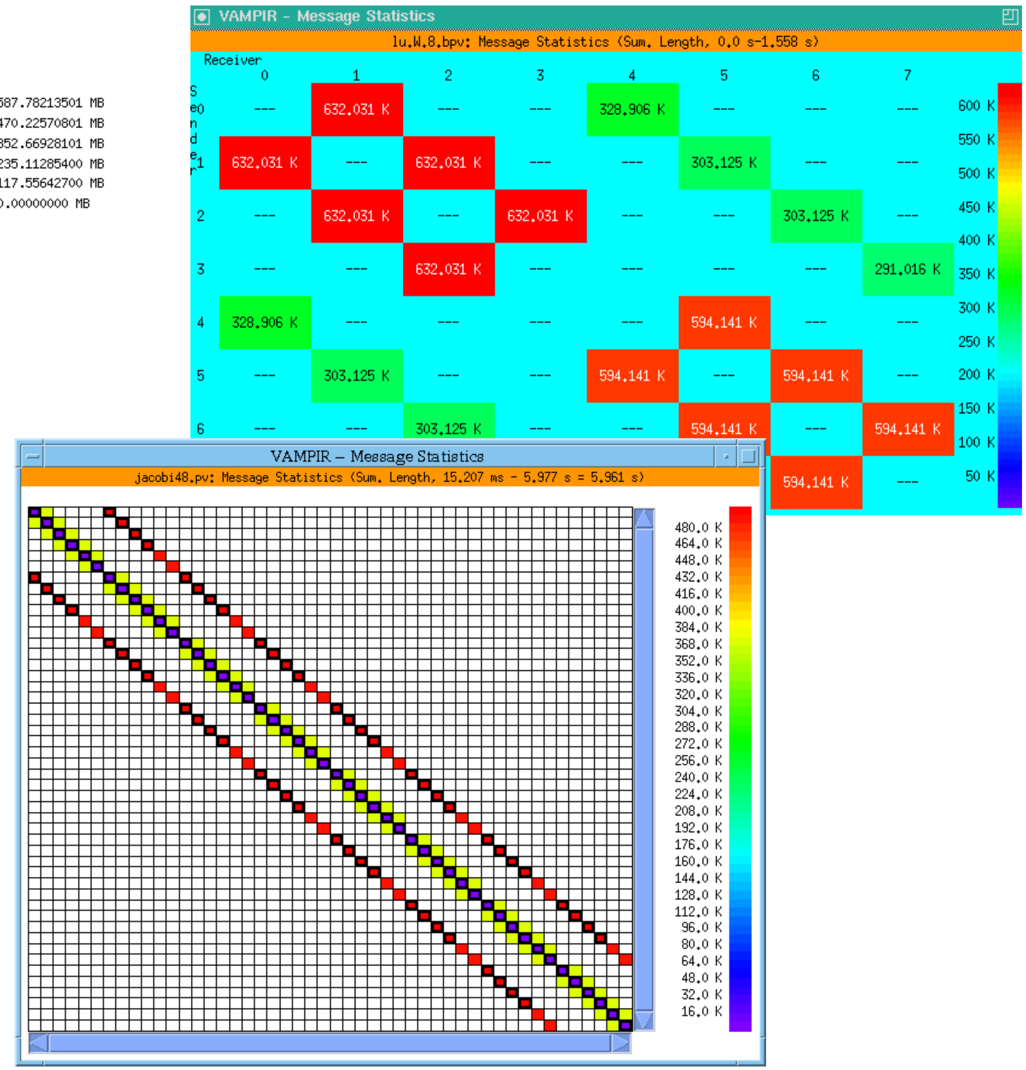


- Vampir: timeline view
  - Similar other tools: Jumpshot, Paraver, Intel Trace Analyzer

Trace File Visualization

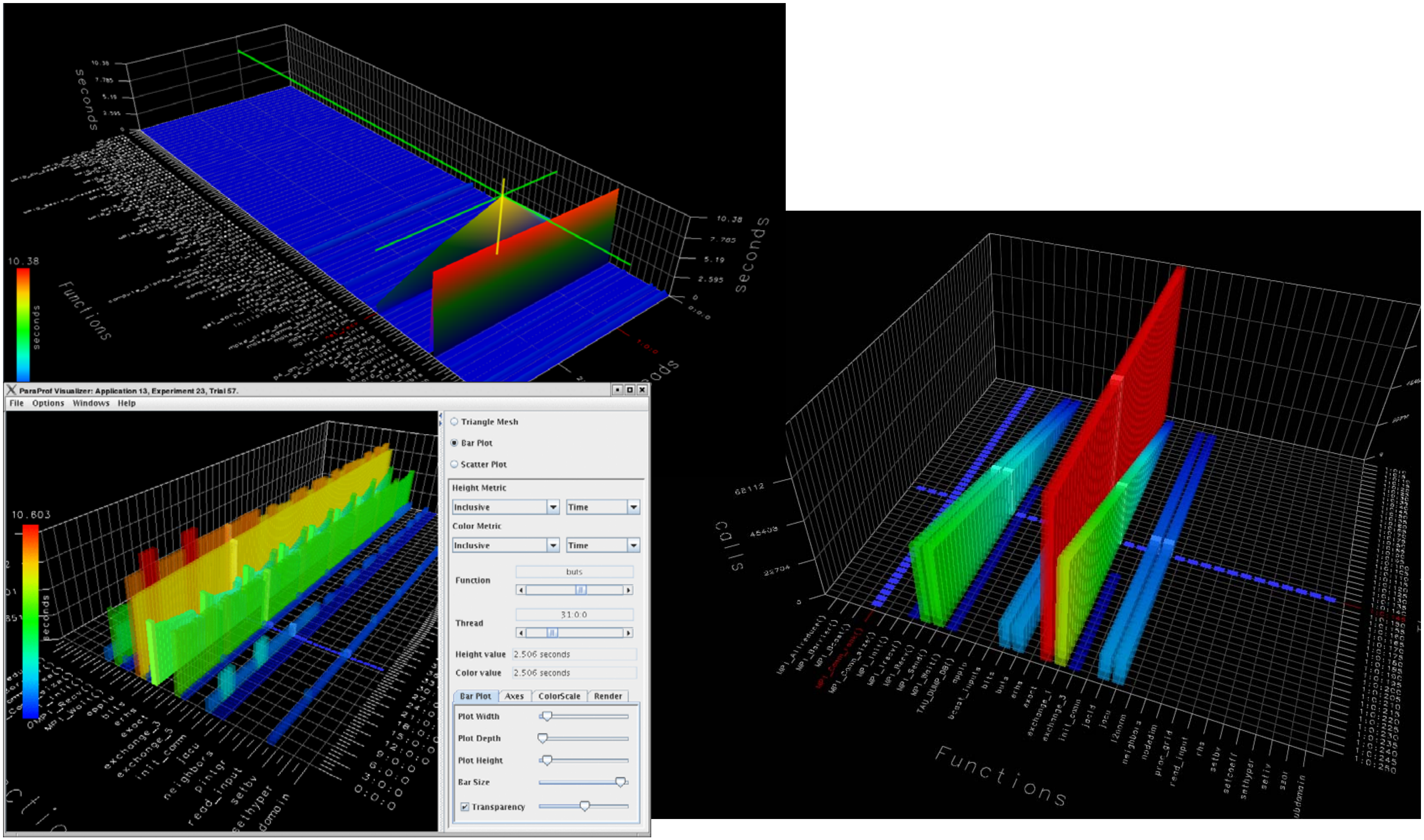
- Vampir/IPM: message communication statistics





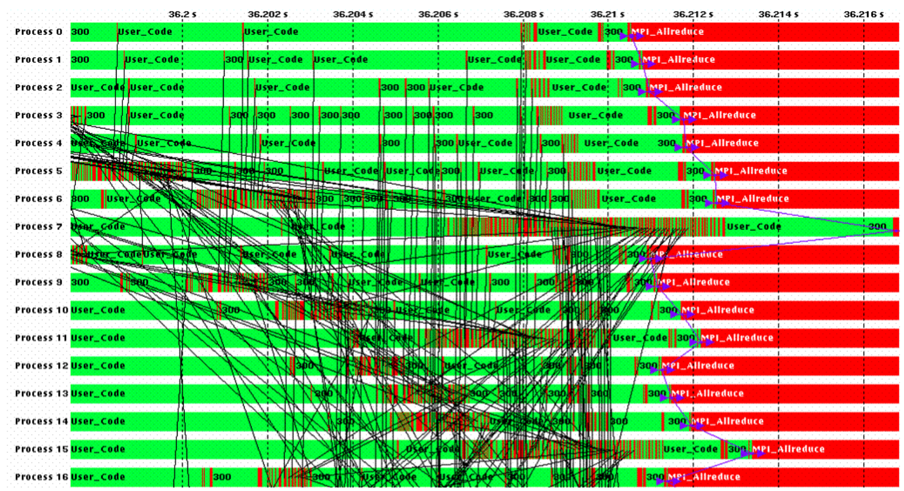
3D performance data exploration

- Paraprof viewer (from the TAU toolset)

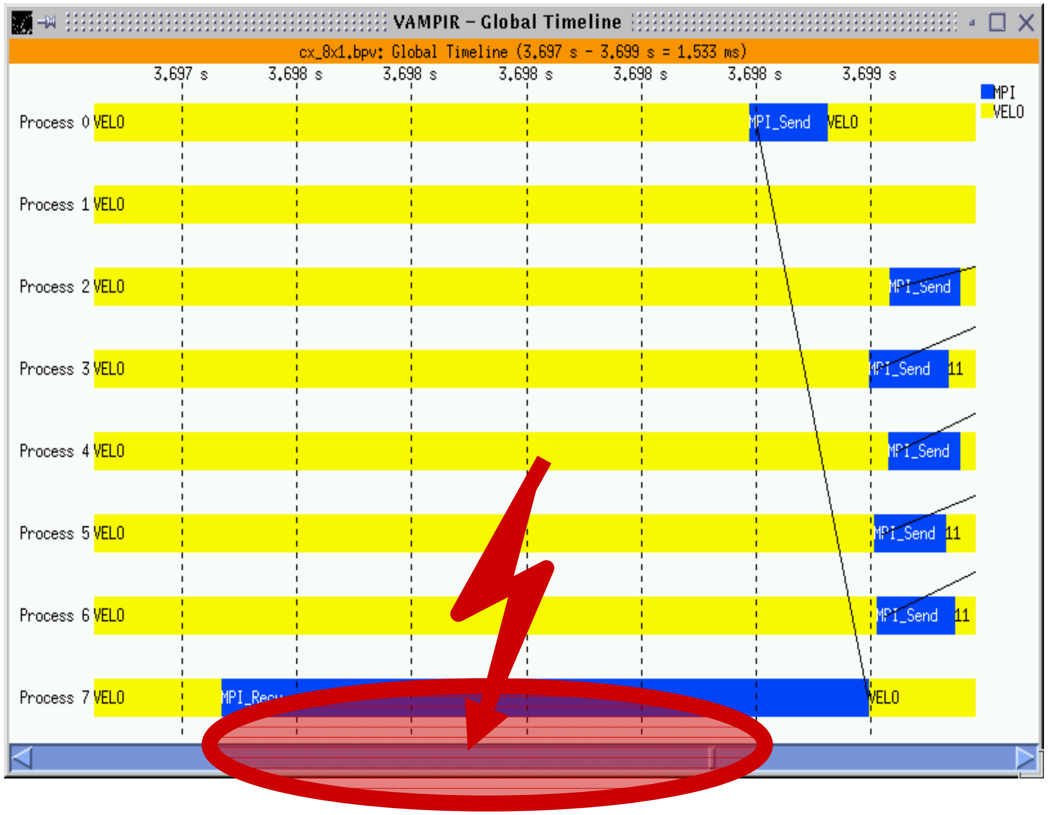


Automated Performance Analysis

- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: 1.6 million cores
  - Trend towards multicore, manycore, accelerators
- Large amounts of performance data when tracing
  - Several gigabytes or even terabytes
- Not all programmers are performance experts
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- Automation can solve some of these issues



Automation - Example



- „Late sender“ pattern

- This pattern can be detected automatically by analyzing the trace

Outline

- Motivation
- Concepts and Definitions
  - Instrumentation, monitoring, analysis
- Some tools and their functionality
  - PAPI – access to hardware performance counters
  - ompP – profiling OpenMP code
  - IPM – monitoring message passing applications

Hardware Performance Counters

- HW counters are **specialized hardware registers** to measure the performance of various aspects of a microprocessor
- Originally and still used for chip verification purposes
- Can provide very detailed insight into:
  - Cache behavior
  - Branching behavior
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
- Counters vs. events
  - Usually a large number of countable events (several hundred)
  - On a small number of counters (4-18)
  - Restrictions on what can be counted simultaneously
  - PAPI handles multiplexing if required

What is PAPI

- **Middleware** that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.

- Countable events are defined in two ways:
  - Platform-neutral **Preset Events** (e.g., PAPI_TOT_INS)
  - Platform-dependent **Native Events** (e.g., L3_CACHE_MISS)

- Preset Events can be **derived** from multiple Native Events (e.g. PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform)

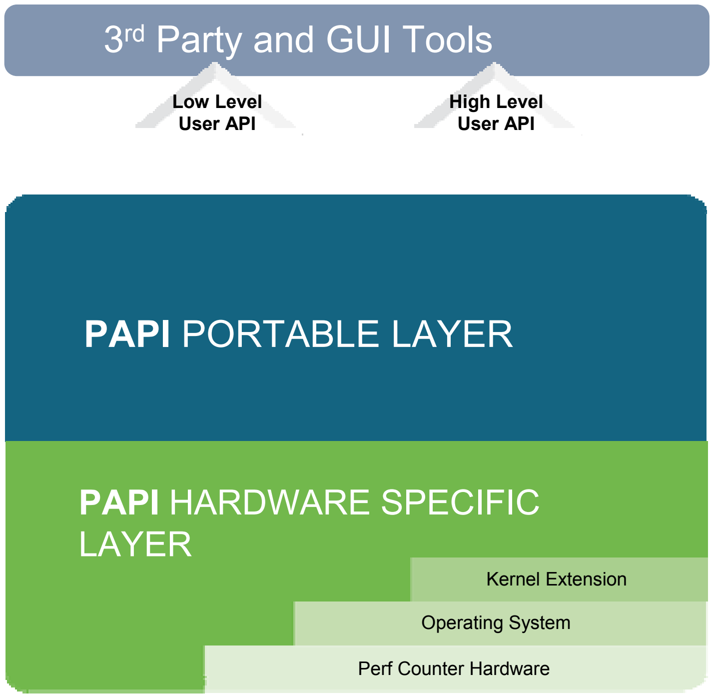
- Preset events are defined in a best effort manner
  - No guarantees of semantics portably
  - Figuring out what a counter actually counts and if it does so correctly can be difficult

PAPI Hardware Events

- Preset Events
  - Standard set of over 100 events for application performance tuning
  - Mapped to either single or linear combinations of native events on each platform
  - Use **papi_avail** to see what preset events are available on a given platform
- Native Events
  - Any event countable by the CPU
  - Same interface as for preset events
  - Use **papi_native_avail** utility to see all available native events
- Use **papi_event_chooser** utility to select a compatible set of events

PAPI Counter Interfaces

- PAPI provides 3 interfaces to the underlying counter hardware:
  - A **low level API** manages hardware events (preset and native) in user-defined groups called EventSets.
    Meant for experienced application programmers wanting fine-grained measurements.
  - A **high level API** provides the ability to start, stop and read the counters for a specified list of events (preset only).
    Meant for programmers wanting simple event measurements.
  - **Graphical** and end-user tools



PAPI High Level Calls

- PAPI_num_counters()
  - get the number of hardware counters available on the system
- PAPI_flips (**float** *rtime**,** **float** *ptime, **long long** *flpins, **float** *mflips)
  - simplified call to get Mflips/s (floating point instruction rate), real and processor time
- PAPI_flops (**float** *rtime, **float** *ptime, **long long** *flpops, **float** *mflops)
  - simplified call to get Mflops/s (floating point operation rate), real and processor time
- PAPI_ipc (**float** *rtime, **float** *ptime, **long long** *ins, **float** *ipc)
  - gets instructions per cycle, real and processor time
- PAPI_accum_counters (**long long** *values, **int** array_len)
  - add current counts to array and reset counters
- PAPI_read_counters (**long long** *values, **int** array_len)
  - copy current counts to array and reset counters
- PAPI_start_counters (**int** *events, **int** array_len)
  - start counting hardware events
- PAPI_stop_counters (**long long** *values, **int** array_len)
  - stop counters and return current counts

PAPI Example Low Level API Usage

```
#include "papi.h”
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];

/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);

/* Start the counters */
retval = PAPI_start (EventSet);

do_work();  /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);
```

Using PAPI through tools

- You can use PAPI directly in your application, but most people use it through tools

- Tool might have a predfined set of counters or lets you select counters through a configuration file/environment variable, etc.
  - E.g., export IPM_HPM=PAPI_FP_OPS

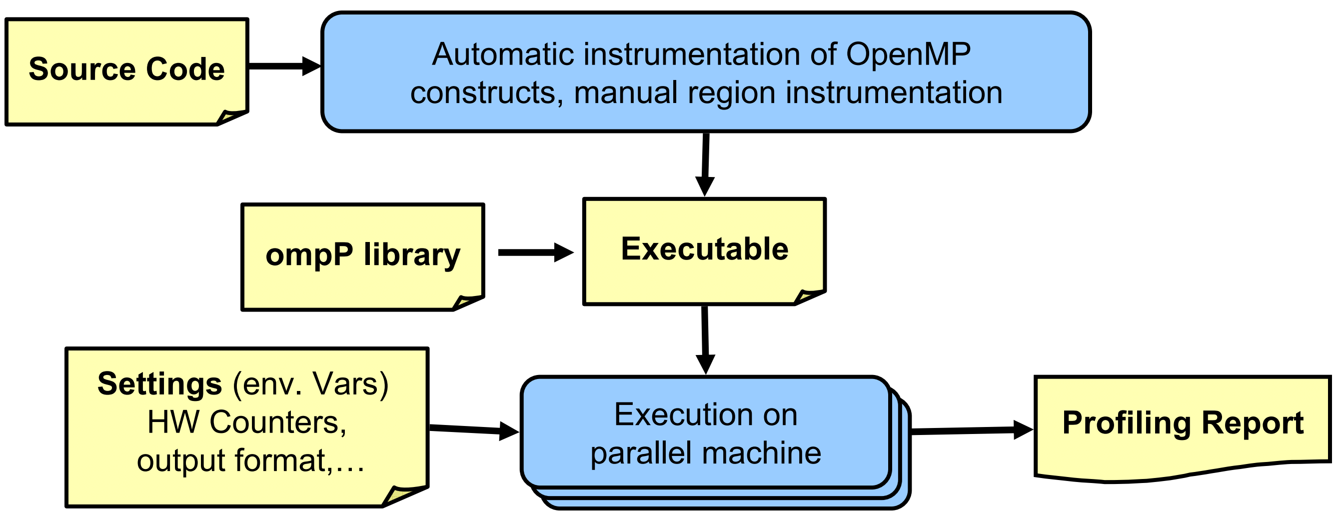
- Tools using PAPI
  - TAU (UO)
  - PerfSuite (NCSA)
  - HPCToolkit (Rice)
  - KOJAK, Scalasca (FZ Juelich, UTK)
  - Open|Speedshop (SGI)
  - ompP (LBL, UCB)
  - IPM (LBL)

Outline

- Motivation
- Concepts and Definitions
  - Instrumentation, monitoring, analysis
- Some tools and their functionality
  - PAPI – access to hardware performance counters
  - ompP – profiling OpenMP code
  - IPM – monitoring message passing applications

OpenMP Performance Analysis with ompP

- ompP: Profiling tool for OpenMP
  - Based on source code instrumentation
  - Independent of the compiler and runtime used
  - Tested and supported: Linux, Solaris, AIX and Intel, Pathscale, PGI, IBM, gcc, SUN studio compilers
  - Supports HW counters through PAPI
  - Uses source code instrumenter ***Opari*** from the KOJAK/SCALASCA toolset
  - Available for download (GPL): http://www.ompp-tool.com



ompP's Profiling Report

- Header
  - Date, time, duration of the run, number of threads, used hardware counters,…
- Region Overview
  - Number of OpenMP regions (constructs) and their source-code locations
- Flat Region Profile
  - Inclusive times, counts, hardware counter data
- Callgraph
- Callgraph Profiles
  - With Inclusive and exclusive times
- Overhead Analysis Report
  - Four overhead categories
  - Per-parallel region breakdown
  - Absolute times and percentages

Profiling Data

- Example profiling data

Code:

```
#pragma omp parallel
{
 #pragma omp critical
 {
   sleep(1.0);
 }
}
```

Profile:

```
R00002 main.c (34-37) (default) CRITICAL
 TID     execT      execC     bodyT    enterT     exitT   PAPI_TOT_INS
   0      3.00          1      1.00      2.00      0.00           1595
   1      1.00          1      1.00      0.00      0.00           6347
   2      2.00          1      1.00      1.00      0.00           1595
   3      4.00          1      1.00      3.00      0.00           1595
 SUM     10.01          4      4.00      6.00      0.00          11132
```

- Components:
  - Source code location and type of region
  - Timing data and execution counts, **depending on the particular construct**
  - One line per thread, last line sums over all threads
  - Hardware counter data (if PAPI is available and HW counters are selected)
  - Data is “exact” (measured, not based on sampling)

Flat Region Profile (2)

- Times and counts reported by ompP for various OpenMP constructs

	main		enter		body					barr	exit	
construct	execT	execC	enterT	startupT	bodyT	sectionT	sectionC	singleT	singleC	exitBarT	exitT	shutdwnT
MASTER	•	•										
ATOMIC	•	•										
BARRIER	•	•										
FLUSH	•	•										
USER REGION	•	•										
CRITICAL	•	•	•		•						•	
LOCK	•	•	•		•						•	
LOOP	•	•			•					•		
WORKSHARE	•	•			•					•		
SECTIONS	•	•				•	•			•		
SINGLE	•	•						•	•	•		
PARALLEL	•	•		•	•					•		•
PARALLEL LOOP	•	•		•	•					•		•
PARALLEL SECTIONS	•	•		•		•	•			•		•
PARALLEL WORKSHARE	•	•		•	•					•		•

Ends with **T: time**

Ends with **C: count**

Main =
enter +
body +
barr +
exit

Overheads Analysis (1)

- Certain timing categories reported by ompP can be classified as overheads:
  - Example: **`exitBarT`**: time wasted by threads idling at the exit barrier of work-sharing constructs. Reason is most likely an **imbalanced** amount of work

- Four overhead categories are defined in ompP:
  - **Imbalance**: waiting time incurred due to an imbalanced amount of work in a worksharing or parallel region
  - **Synchronization**: overhead that arises due to threads having to synchronize their activity, e.g. **`barrier`** call
  - **Limited Parallelism**: idle threads due not enough parallelism being exposed by the program
  - **Thread management**: overhead for the creation and destruction of threads, and for signaling critical sections, locks as available

Overhead Analysis (2)

	main		*enter*		*body*					*barr*	*exit*	
construct	execT	execC	enterT	startupT	bodyT	sectionT	sectionC	singleT	singleC	exitBarT	exitT	shutdwnT
MASTER	•	•										
ATOMIC	•(S)	•										
BARRIER	•(S)	•										
FLUSH	•(S)	•										
USER REGION	•	•										
CRITICAL	•	•	•(S)		•						•(M)	
LOCK	•	•	•(S)		•						•(M)	
LOOP	•	•			•					•(I)		
WORKSHARE	•	•			•					•(I)		
SECTIONS	•	•				•	•			•(I/L)		
SINGLE	•	•						•	•	•(L)		
PARALLEL	•	•		•(M)	•					•(I)		•(M)
PARALLEL LOOP	•	•		•(M)	•					•(I)		•(M)
PARALLEL SECTIONS	•	•		•(M)		•	•			•(I/L)		•(M)
PARALLEL WORKSHARE	•	•		•(M)	•					•(I)		•(M)

S: Synchronization overhead

I: Imbalance overhead

M: Thread management overhead

L: Limited Parallelism overhead

ompP's Overhead Analysis Report

```
----------------------------------------------------------------------
----     ompP Overhead Analysis Report    ----------------------------
----------------------------------------------------------------------
Total runtime (wallclock)    : 172.64 sec [32 threads]
Number of parallel regions   : 12
Parallel coverage            : 134.83 sec (78.10%)


Parallel regions sorted by wallclock time:
          Type                            Location       Wallclock (%)
R00011  PARALL                  mgrid.F (360-384)       55.75 (32.29)
R00019  PARALL                  mgrid.F (403-427)       23.02 (13.34)
R00009  PARALL                  mgrid.F (204-217)       11.94 ( 6.92)
...
                                              SUM      134.83 (78.10)


Overheads wrt. each individual parallel region:
          Total        Ovhds (%)  =    Synch   (%)  +  Imbal    (%)  +   Limpar (%)   +     Mgmt (%)
R00011  1783.95   337.26 (18.91)    0.00 ( 0.00)  305.75 (17.14)    0.00 ( 0.00)   31.51 ( 1.77)
R00019   736.80   129.95 (17.64)    0.00 ( 0.00)  104.28 (14.15)    0.00 ( 0.00)   25.66 ( 3.48)
R00009   382.15   183.14 (47.92)    0.00 ( 0.00)   96.47 (25.24)    0.00 ( 0.00)   86.67 (22.68)
R00015   276.11    68.85 (24.94)    0.00 ( 0.00)   51.15 (18.52)    0.00 ( 0.00)   17.70 ( 6.41)
...


Overheads wrt. whole program:
          Total        Ovhds (%)  =    Synch   (%)  +  Imbal    (%)  +   Limpar (%)   +     Mgmt (%)
R00011  1783.95   337.26 ( 6.10)    0.00 ( 0.00)  305.75 ( 5.53)    0.00 ( 0.00)   31.51 ( 0.57)
R00009   382.15   183.14 ( 3.32)    0.00 ( 0.00)   96.47 ( 1.75)    0.00 ( 0.00)   86.67 ( 1.57)
R00005   264.16   164.90 ( 2.98)    0.00 ( 0.00)   63.92 ( 1.16)    0.00 ( 0.00)  100.98 ( 1.83)
R00007   230.63   151.91 ( 2.75)    0.00 ( 0.00)   68.58 ( 1.24)    0.00 ( 0.00)   83.33 ( 1.51)
...
   SUM  4314.62  1277.89 (23.13)    0.00 ( 0.00)  872.92 (15.80)    0.00 ( 0.00)  404.97 ( 7.33)
```

Number of threads, parallel regions, parallel coverage

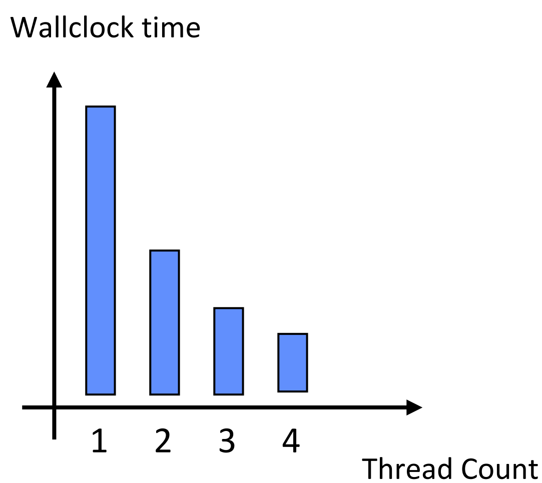
Wallclock time * number of threads

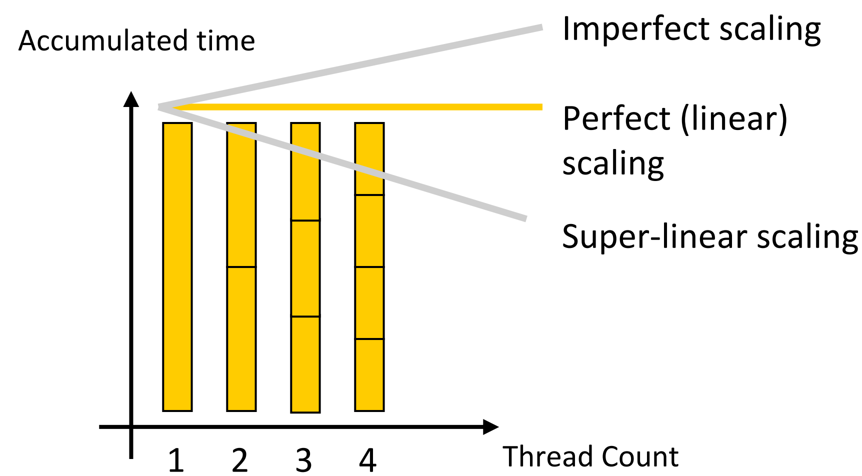
Overhead percentages wrt. this particular parallel region

Overhead percentages wrt. whole program

OpenMP Scalability Analysis

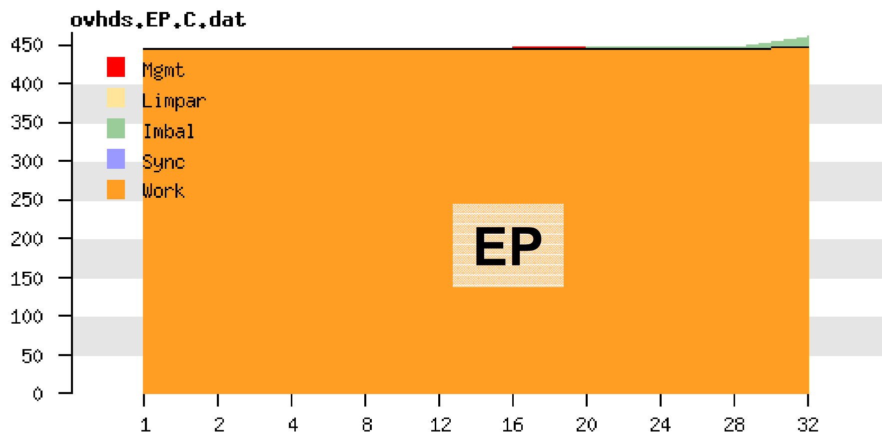
- Methodology
  - Classify execution time into “Work” and four overhead categories: “Thread Management”, “Limited Parallelism”, “Imbalance”, “Synchronization”
  - Analyze how overheads behave for increasing thread counts
  - Graphs show accumulated runtime over all threads for fixed workload (strong scaling)
  - Horizontal line = perfect (linear) scalability

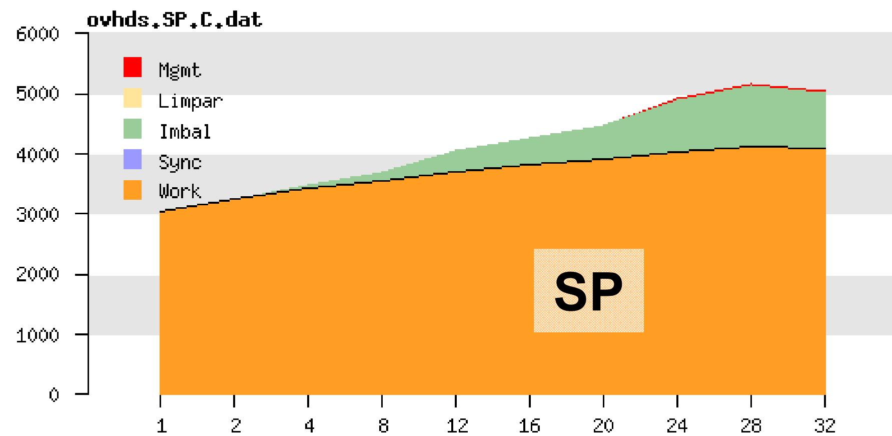




Example

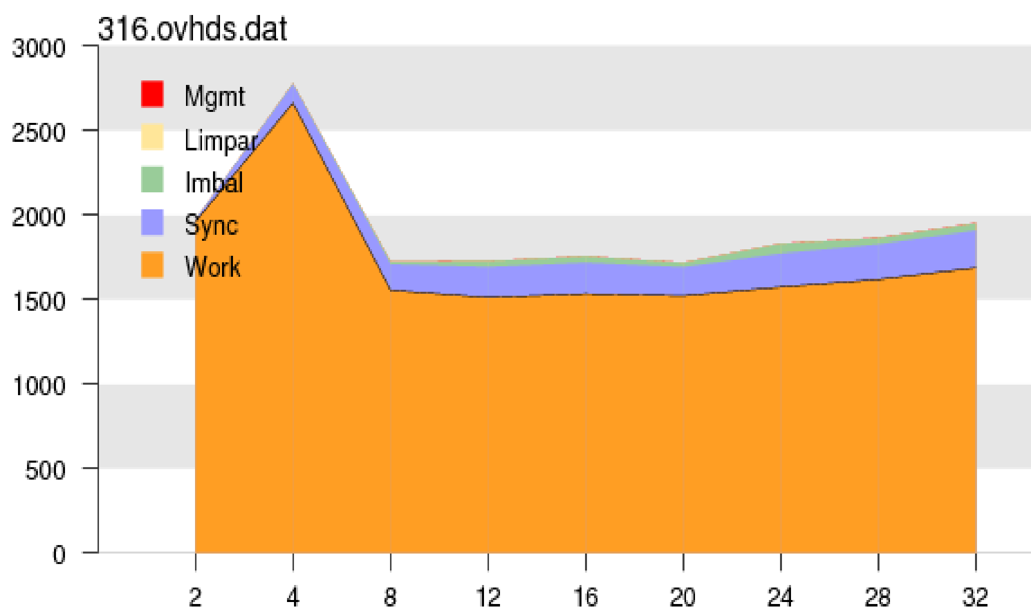
- Example
  - NAS Parallel Benchmarks
  - Class C, SGI Altix machine (Itanium 2, 1.6 GHz, 6MB L3 Cache)

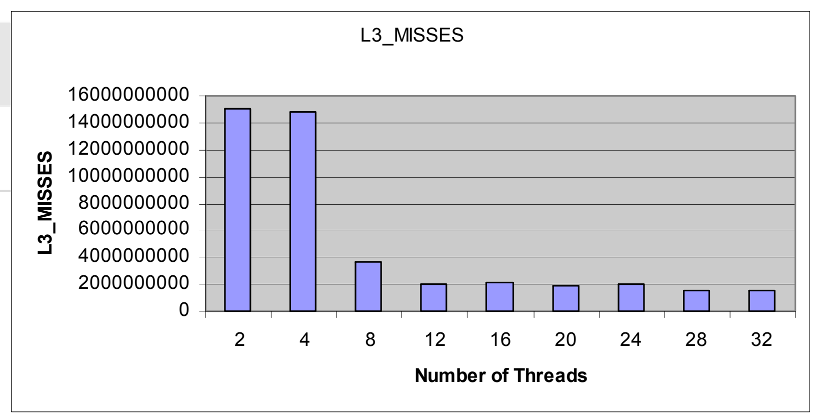




SPEC OpenMP Benchmarks (2)

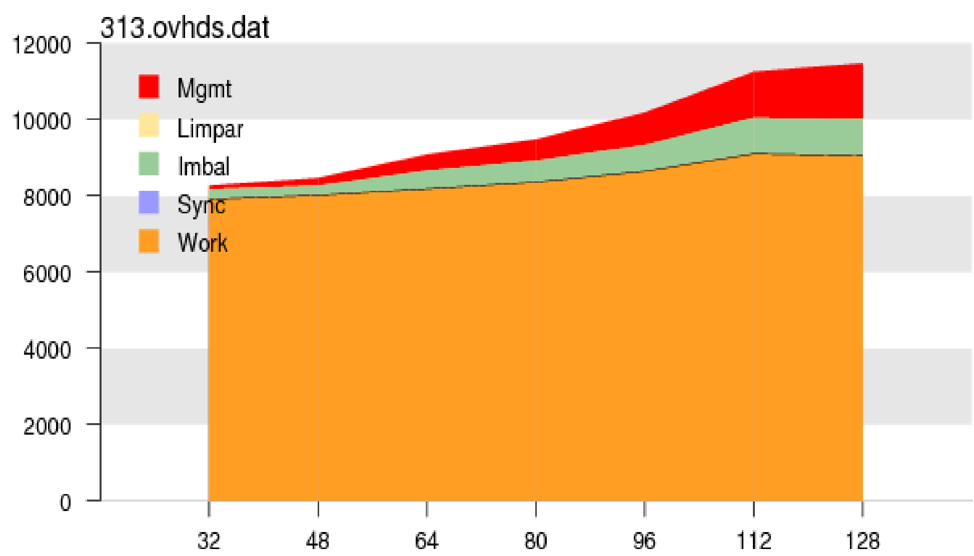
- Application 316.applu
  - Super-linear speedup
  - Only one parallel region (ssor.f 138-209) shows super-linear speedup, contributes 80% of accumulated total execution time
  - Most likely reason for super-linear speedup: increased overall cache size





SPEC OpenMP Benchmarks (3)

- Application 313.swim
  - Dominating source of inefficiency is thread management overhead
  - Main source: reduction of three scalar variables in a small parallel loop in swim.f 116-126.
  - At 128 threads more than 6 percent of the total accumulated runtime is spent in the reduction operation
  - Time for the reduction operation is larger than time spent in the body of the parallel region



Outline

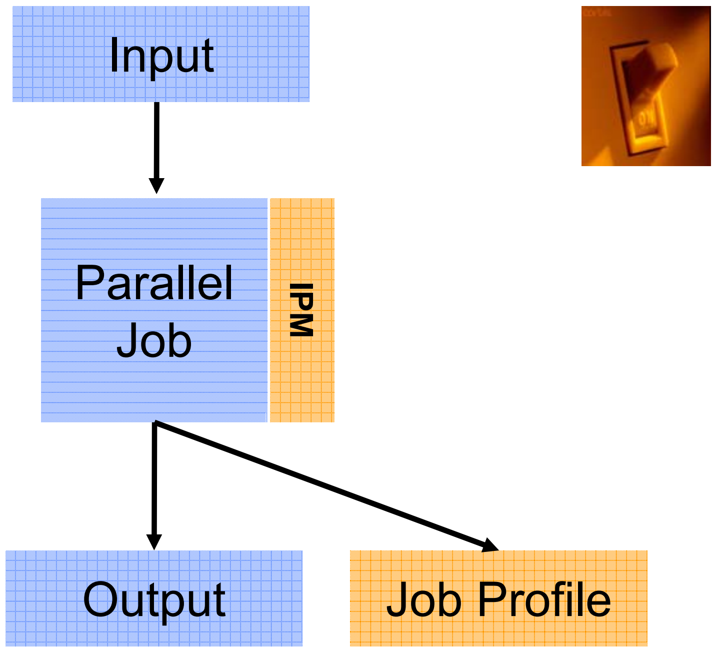
- Motivation
- Concepts and Definitions
  - Instrumentation, monitoring, analysis
- Some tools and their functionality
  - PAPI – access to hardware performance counters
  - ompP – profiling OpenMP code
  - IPM – monitoring message passing applications

What is IPM

- IPM implements a thin measurement layer
  - Sitting between the application and the runtime/OS

- Goals
  - **Efficient** gathering of **high-level** performance metrics
  - Event inventorization
  - Determination of resource requirements and first order identification of performance problems
  - Less focus on drill-down into application
    - » Currently no automatic function-level instrumentation
    - » Manual region instrumentation supported

IPM Philosophy



- Banner on stderr
- Detailed profiling log file (XML format)
- Profiling report (HTML format)

- "Flip of a switch" monitoring
  - Resource consumption (used virtual memory, hw counter data)
  - Application execution event statistics
- Using /proc, other OS services, and PAPI for the measuring resource consumption
- Efficient collection of event statistics in a hash table

IPM: Methodology

- MPI_Init()
  - Initialize monitoring environment, allocate memory

- For each MPI call
  - Compute hash **key** from
    - » Type of call (send/recv/bcast/...)
    - » Buffer size
    - » Communication partner rank
    - » Call-site, region or phase identifier, ...
  - Store / update **value** in hash table with timing data
    - » Number of invocations
    - » Minimum duration, maximum duration, summed time

- MPI_Finalize()
  - Aggregate, banner report to stdout, write XML log file

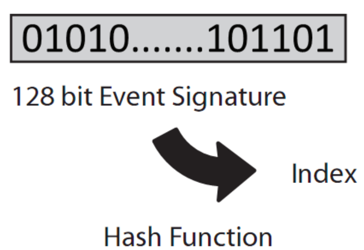
IPM Event Hash Keys

- IPM uses 128 bit hash keys
  - 64 bit context key (where, what)
  - 64 bit resource key (buffer sizes, comm. partners, …)

0 0 0 0
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Event ID	Region ID	Thread ID
Callsite ID	Res. Select	Resvd
Buffer/Message Size		
Partner ID		

- Table holds event statistics
  - Event count
  - Minimum duration
  - Maximum duration
  - Average duration

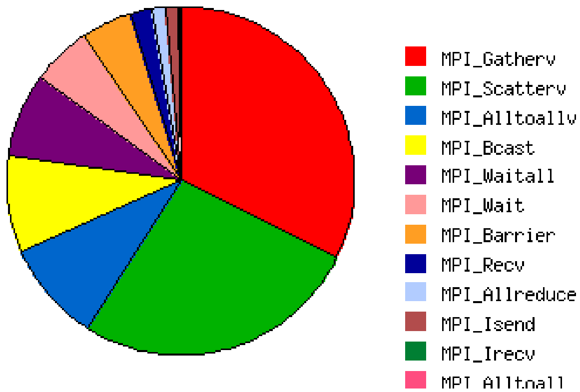


Signature	#events, tmin, tmax, tavg
…	
010...101	728, 3.20, 5.61, 4.41
…	

Performance Data Hash Table

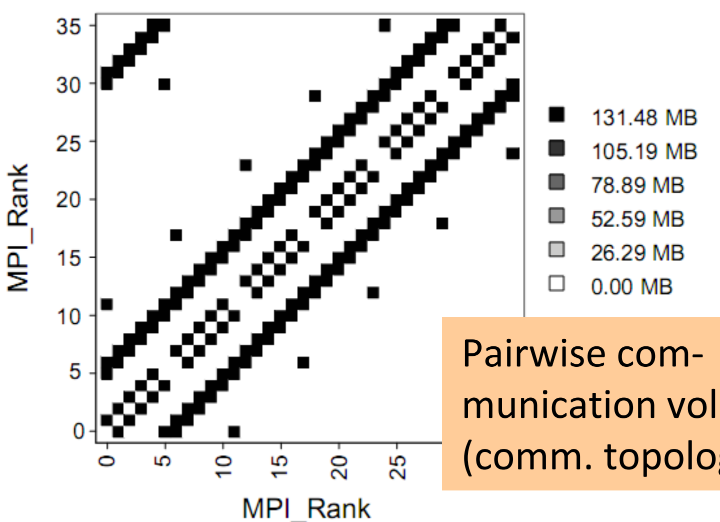
Analyzing the Event Signatures

- The hash table of event signatures contains a lot of interesting data

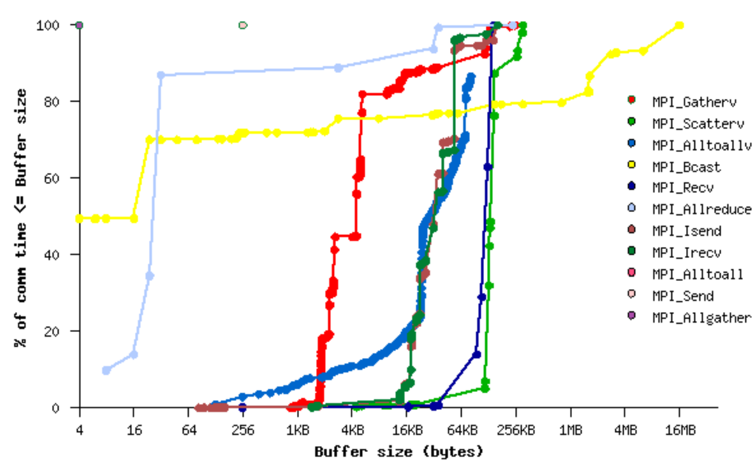


Communication time per type of MPI call

CDF of time per MPI call over message sizes



Pairwise com-munication volume (comm. topology)



Using IPM

- Do "**module load ipm**", then run normally (e.g., on franklin)
  - Uses LD_PRELOAD
  - Re-linking required for static binaries (franklin: include $IPM on link line)
- Upon completion you get :

```
##IPM2#################################################################
# command    : ./a.out
# start      : Sun Mar 14 16:55:39 2010   host      : nid01829
# stop       : Sun Mar 14 17:04:33 2010   wallclock : 533.12
# mpi_tasks  : 2048 on 1024 nodes         %comm     : 29.41
# omp_thrds  : 6                          %omp      : 50.63
# files      : 12                         %i/o      : 12.09
# mem [GB]   : 2774.44                    gflop/sec : 418.58
#######################################################################
```

- Environment variables
  - IPM_HPM for PAPI counters
  - IPM_REPORT = **full** | **terse** | **none**
  - IPM_LOG = **full** | **terse** | **none**

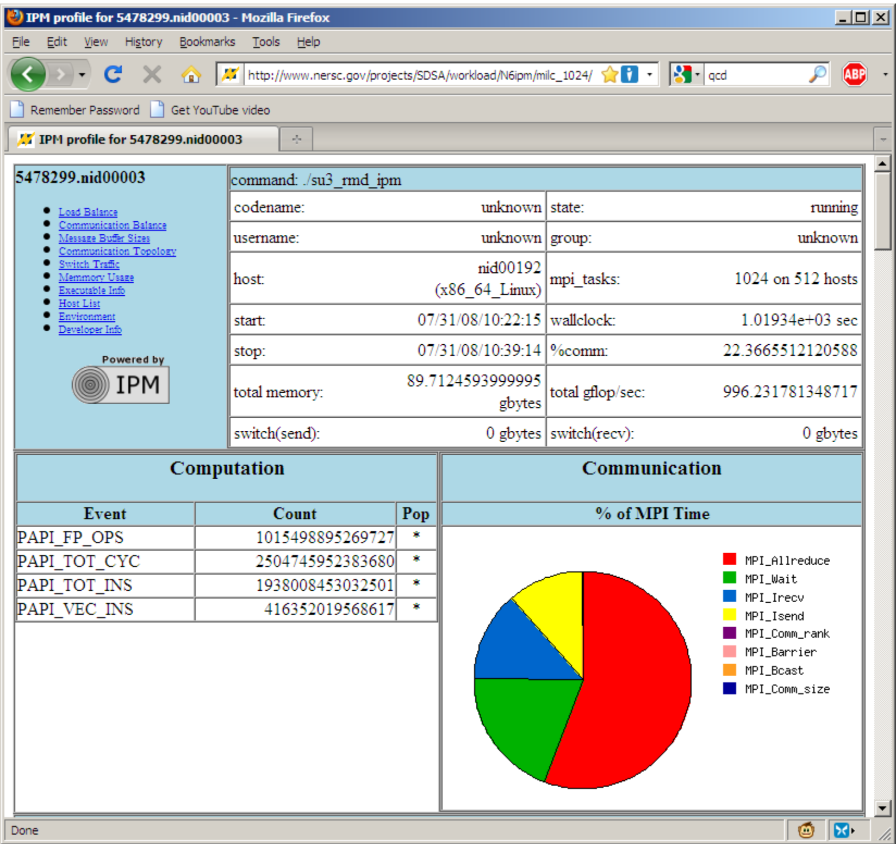
More details with IPM_REPORT=full

```
##IPM2#################################################################
# command    : ./a.out
# start      : Sun Mar 14 16:55:39 2010   host      : nid01829
# stop       : Sun Mar 14 17:04:33 2010   wallclock : 533.12
# mpi_tasks : 2048 on 1024 nodes          %comm     : 29.41
# omp_thrds : 6                           %omp      : 50.63
# files      : 12                         %i/o      : 12.09
# mem [GB]   : 2774.44                    gflop/sec : 418.58
#
#            :      [total]         <avg>         min          max
# wallclock :    1091671.57        533.04      532.99       533.12
# MPI        :    321034.43        156.76      109.03       239.23
# I/O        :    131947.08         64.43       11.83       113.87
# OMP        :    552665.28        269.86      205.07       305.36
# OMP idle   :     48262.98         23.57       21.30        27.40
# %wall      :
#   MPI      :                      29.41       20.45        44.88
#   OMP      :                      50.63       38.47        57.28
#   I/O      :                      12.09        2.22        21.36
# #calls     :
#   MPI      :     76235998         37224       37223        37320
# mem [GB]   :      2774.44          1.35        1.35         1.36
#
#                            [time]        [count]          <%wall>
# OMP_PARALLEL             552665.28      131439989           50.63
# MPI_Allreduce            247648.04       14438400           22.69
# fread                     69813.27        5488640            6.40
# ...
#######################################################################
```

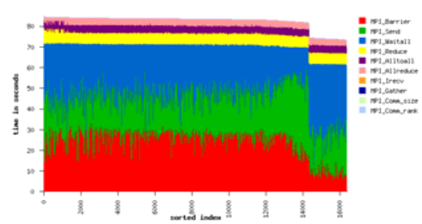
- Statistics of high level metrics across tasks
- Details of the contribution of individual events

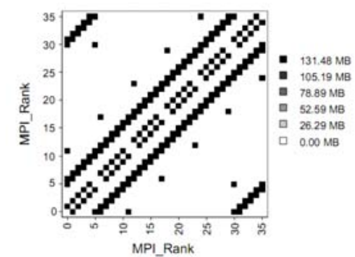
IPM HTML Profiling Report

- ipm_parse generates HTML profiling report

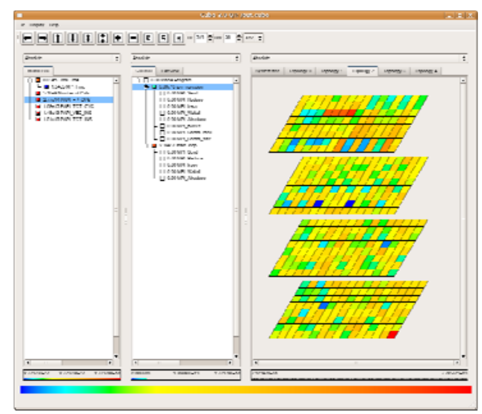


- Contents of the webpage:
  - Banner
  - Communication time breakdown
  - Load balance by task graph
  - Communication balance by task graph
  - Communication topology graph





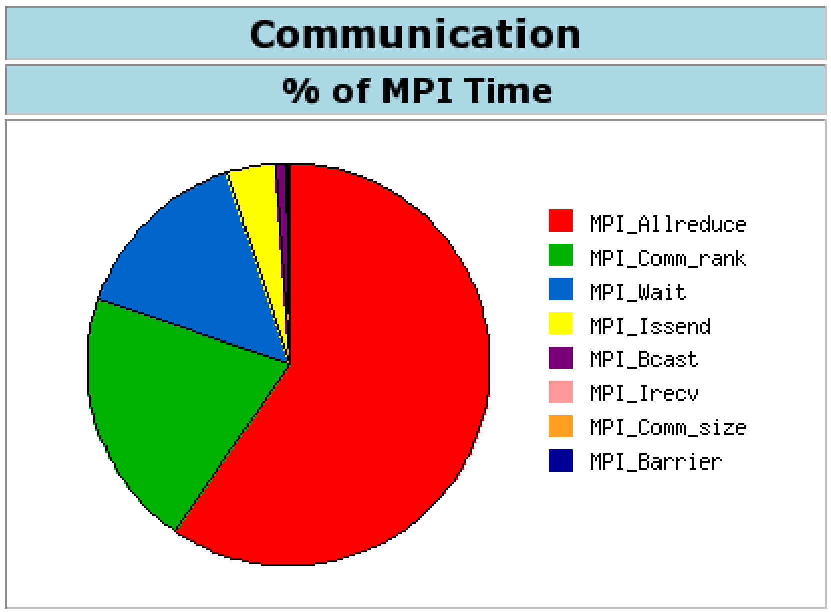
- IPM to CUBE converter



Application Assessment with IPM

- Provide high level performance numbers with small overhead
  - To get an initial read on application runtimes
  - For allocation/reporting
  - To check the performance weather on systems with high variability
- What's going on overall in my code?
  - How much comp, comm, I/O?
  - Where to start with optimization?
- How is my load balance?
  - Domain decomposition vs. concurrency (M work on N tasks)

What's wrong here?



Communication Event Statistics (100.00% detail)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Allreduce	8	3278848	124132.547	0.000	114.920	59.35	16.88
MPI_Comm_rank	0	35173439489	43439.102	0.000	41.961	20.77	5.91
MPI_Wait	98304	13221888	15710.953	0.000	3.586	7.51	2.14
MPI_Wait	196608	13221888	5331.236	0.000	5.716	2.55	0.72
MPI_Wait	589824	206848	5166.272	0.000	7.265	2.47	0.70

Summary

- Performance montioring concepts
  - Instrument, measure, analyze
  - Profiling and tracing,
  - Sampling and direct (instrumentation based) measurment
- Tools
  - PAPI, ompP, IPM as examples
- Lots of other tools
  - Vendor tools: Cray PAT, Oracle (nee Sun) Studio, Intel Thread Profiler, Intel Vtune, Intel PTU,…
  - Independent, portable tools: TAU, Perfsuite, Paradyn, HPCToolkit, Kojak, Scalasca, Vampir, oprofile, gprof, …

Thank you for your attention!