Par Lab Parallel Boot Camp

Performance Tools

Karl Fuerlinger

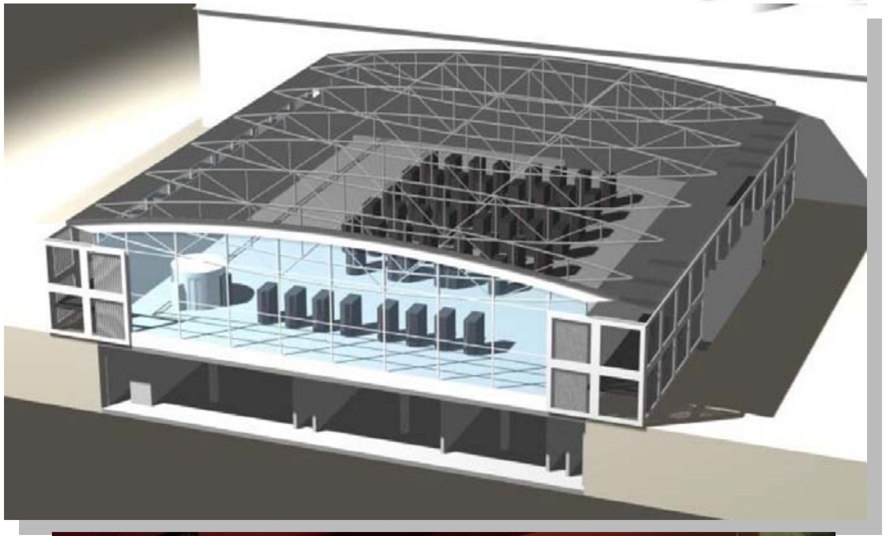
fuerling@eecs.berkeley.edu

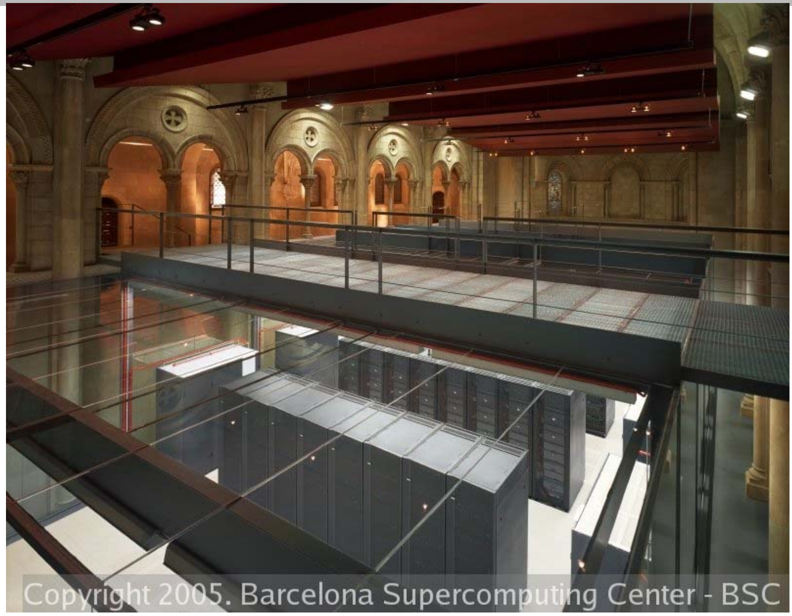
Outline

- 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

Motivation

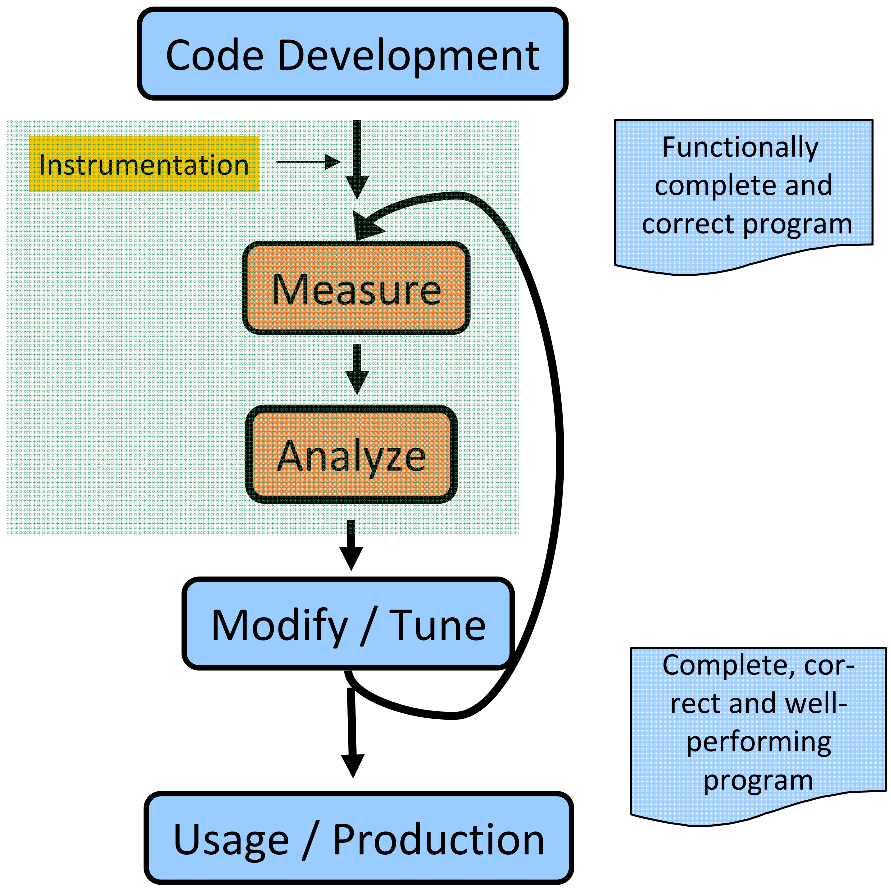
- Performance analysis is important
  - For HPC: computer systems are large investments
    - » Procurement: O($40 Mio)
    - » Operational costs: ~$5 Mio per year
    - » Power: 1 MWyear ~$1 Mio
  - Goals:
    - » Solve **larger** problems (new science)
    - » Solve problems **faster** (turn-around time)
    - » Improve **error** bounds on solutions (confidence)





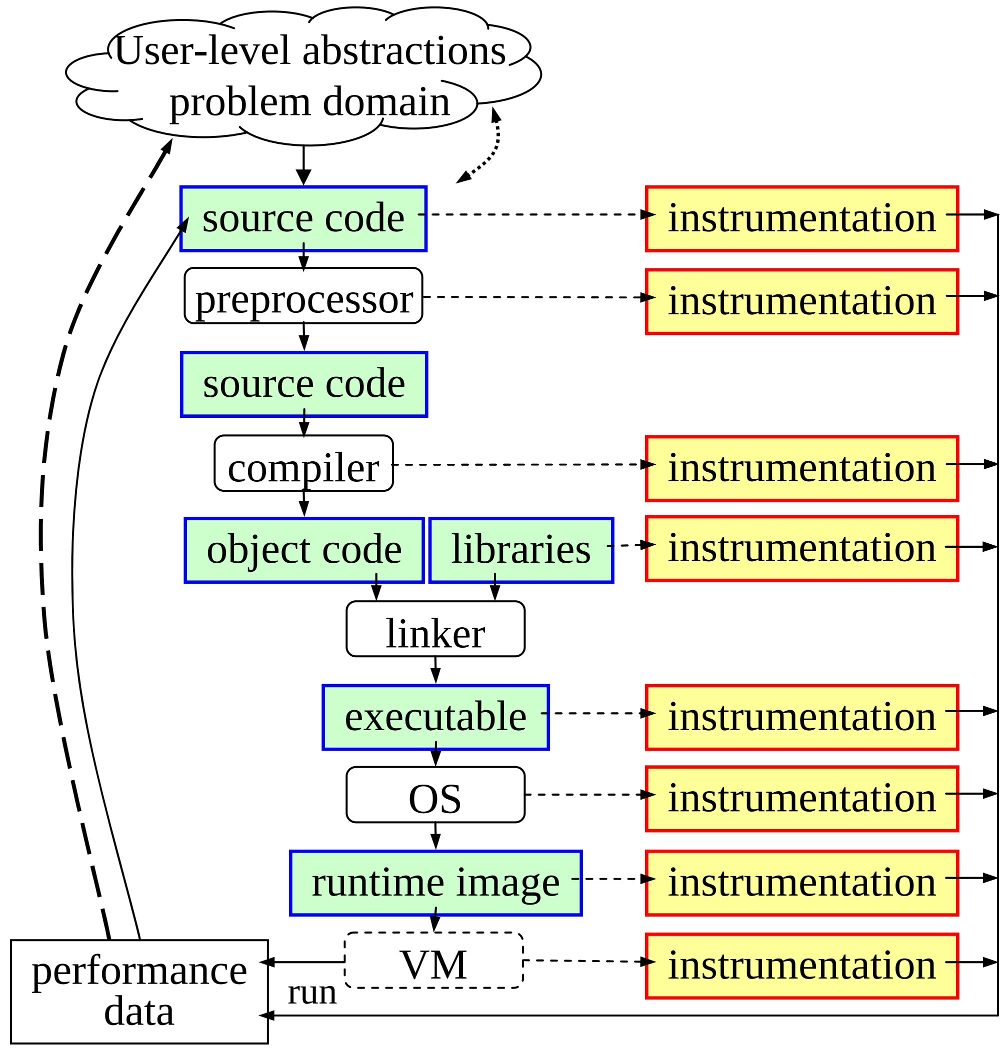
Concepts and Definitions

- The typical performance optimization cycle



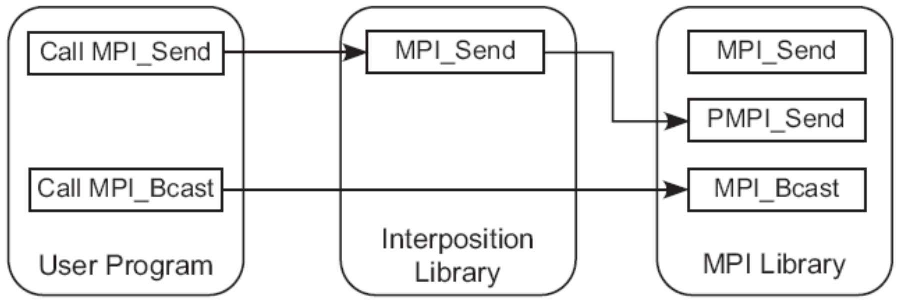
Instrumentation

- Instrumentation := adding measurement probes to the code in order to observe its execution
- Can be done on several levels and ddifferent 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 speed is an issue



Instrumentation – Examples (1)

- 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 instrumented

Instrumentation – Examples (2)

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

Orignial source code → Pre-processor → Modified (instrumented) source code

  - 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 (3)

- 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

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

  - Macro / function call based

```
ELG_USER_START("name");
// application code
```

  - `ELG_USER_END("name");`

Measurement

- Profiling vs. Tracing

- Profiling
  - Summary statistics of performance metrics
    - » Number of times a routine was invoked
    - » Exclusive, inclusive time
    - » Hardware performance counters
    - » Number of child routines invoked, etc.
    - » Structure of invocations (call-trees/call-graphs)
    - » Memory, message communication sizes

- Tracing
  - When and where events took place along a global timeline
    - » Time-stamped log of events
    - » Message communication events (sends/receives) are tracked
    - » Shows when and from/to where messages were sent
    - » Large volume of performance data generated usually leads to more perturbation in the program

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
  - **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** secs
- **Exclusive** time for **main**
  - 100-20-50-20=**10** secs
- 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

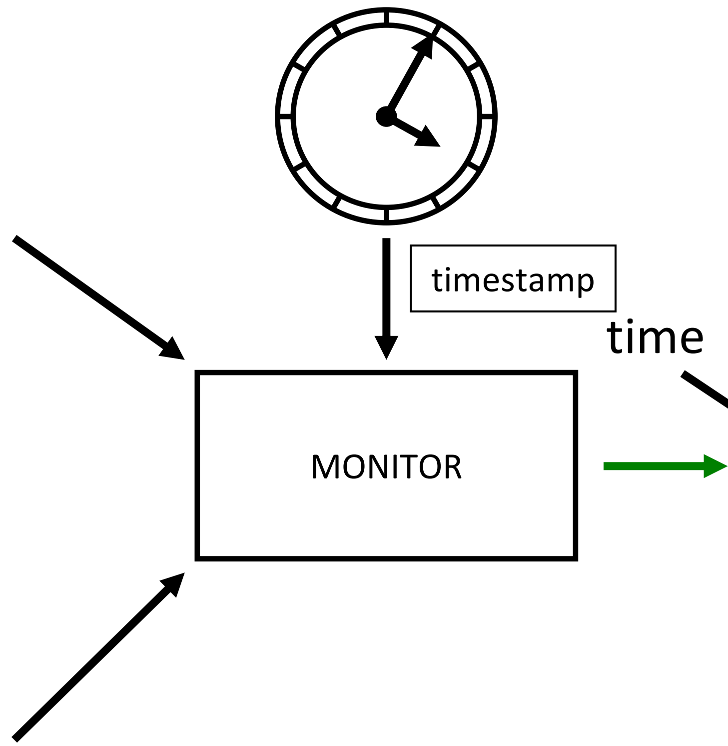
Tracing Example: Instrumentation, Monitor, Trace

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

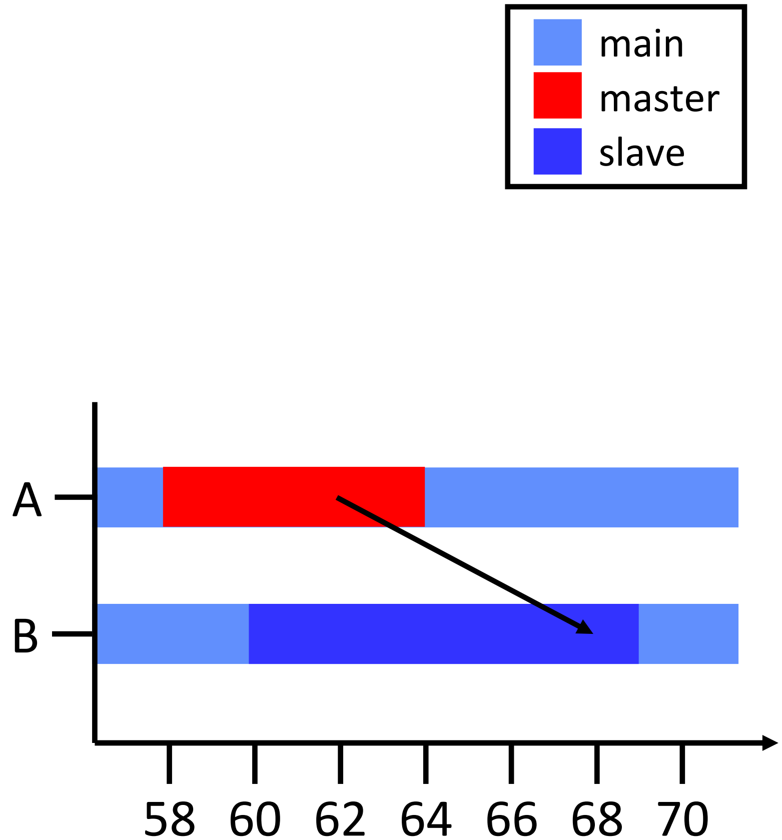
1	master
2	slave
3	...

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
...			



Measurement: 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

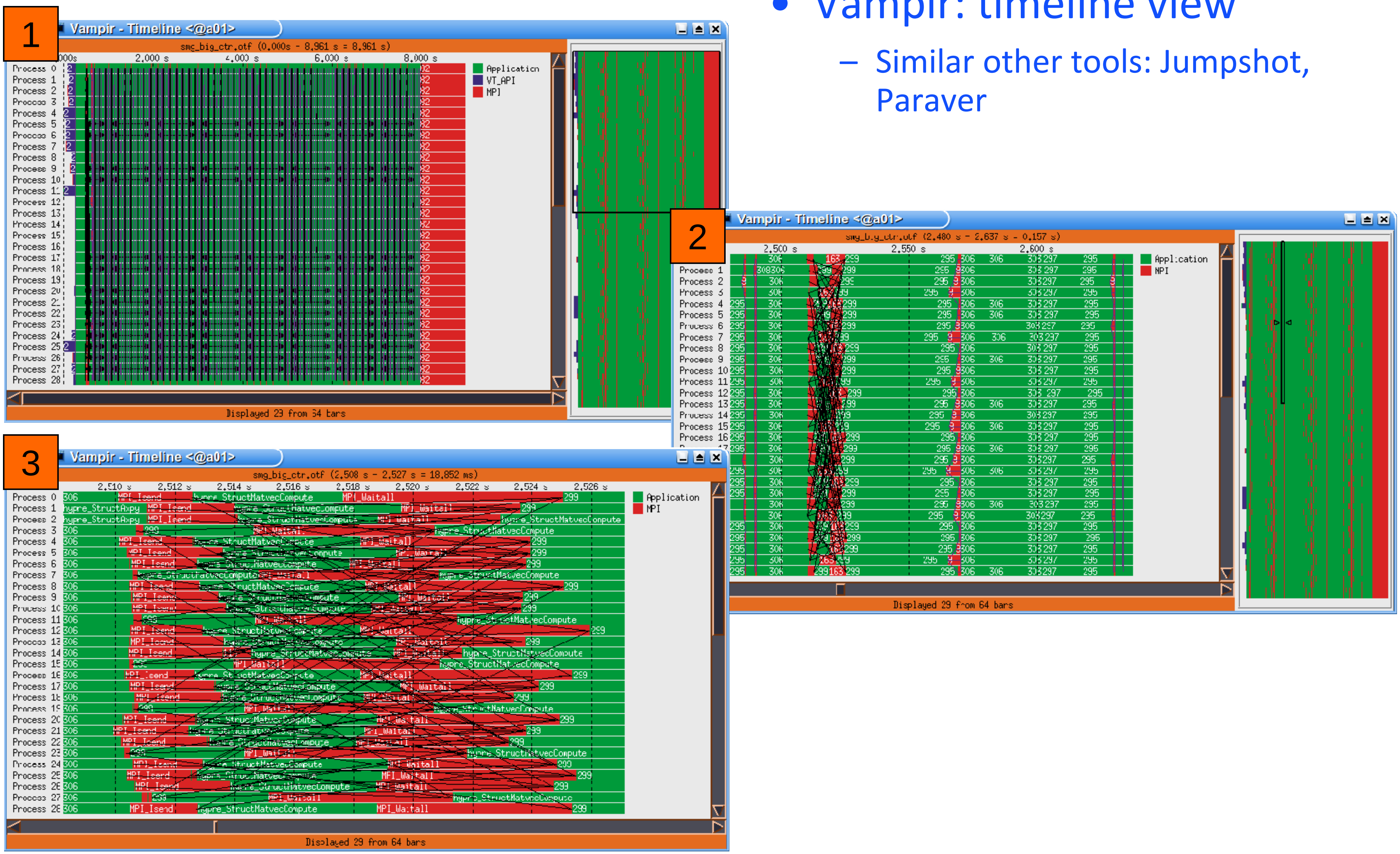
Performance Data Analysis

- Draw conclusions from measured performance data

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

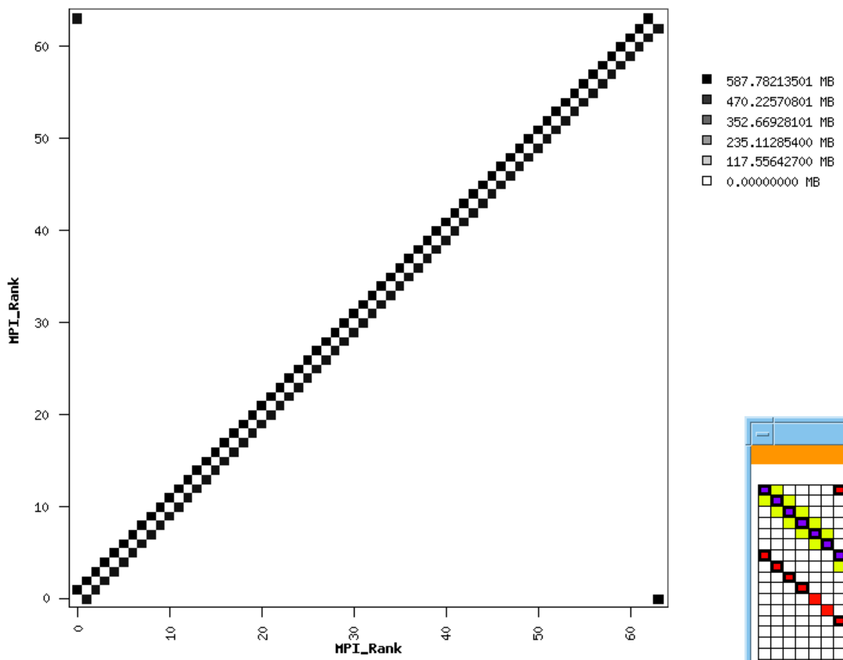
- Automated analysis
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca, Periscope

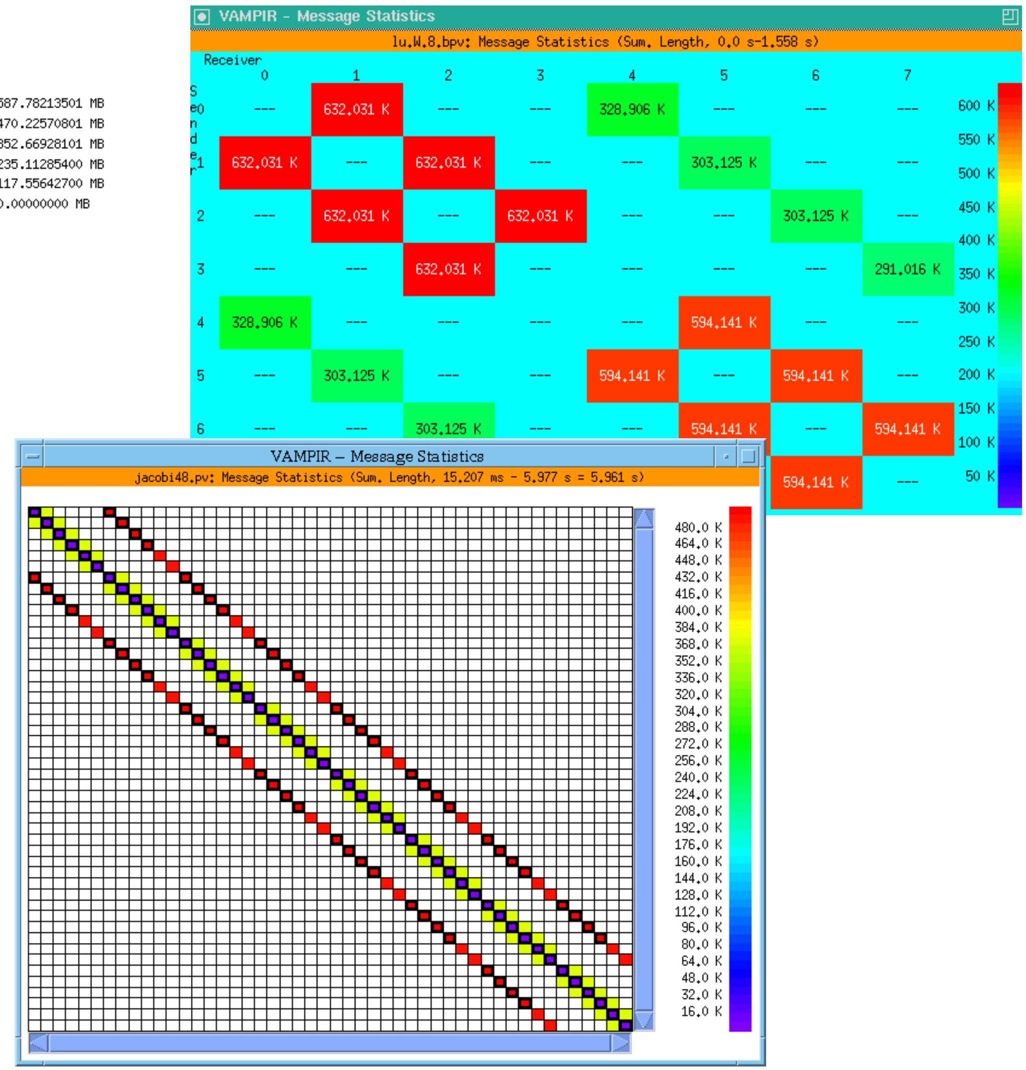
Trace File Visualization



- Vampir: timeline view
  - Similar other tools: Jumpshot, Paraver

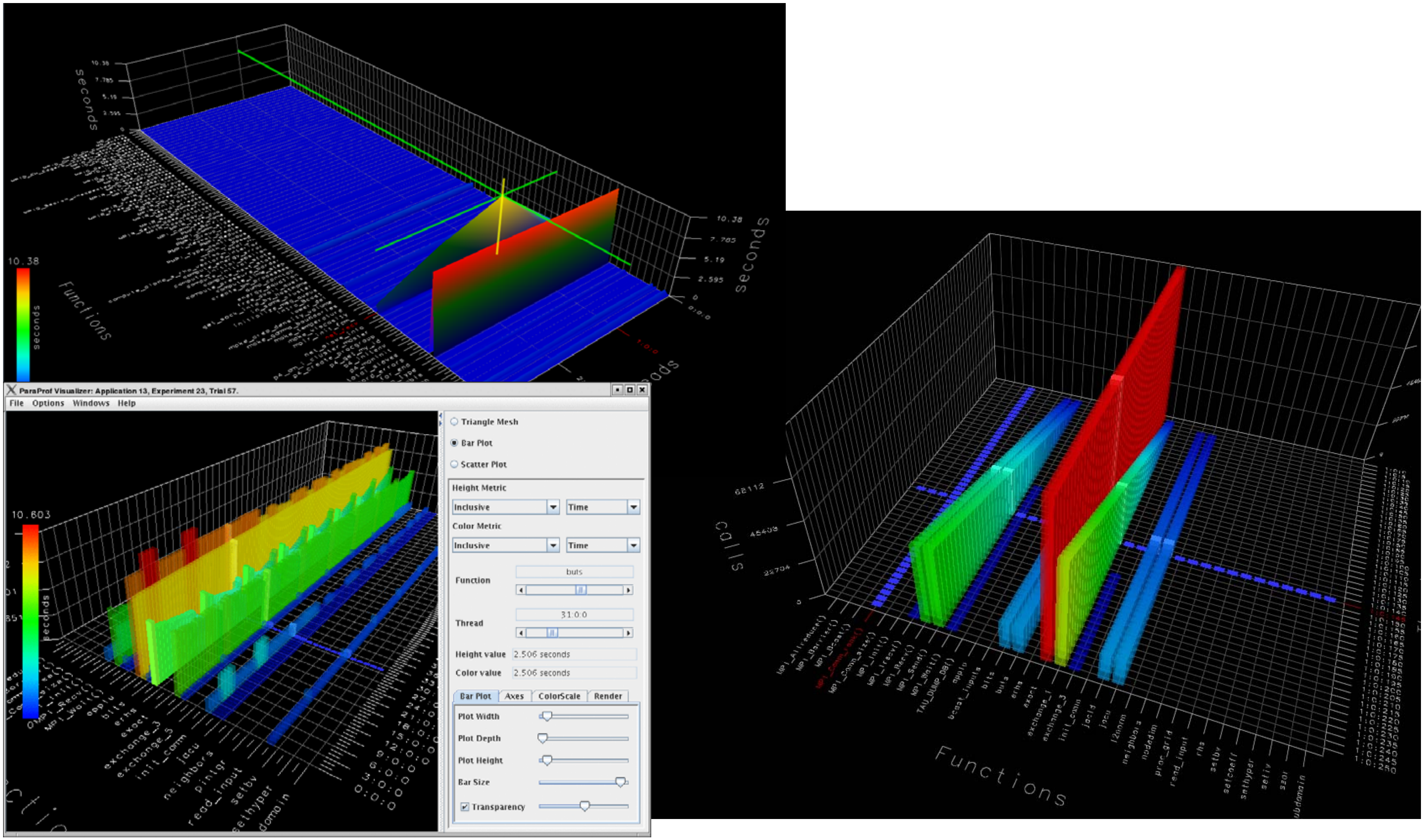
- Vampir/IPM: message communication statistics





3D performance data exploration

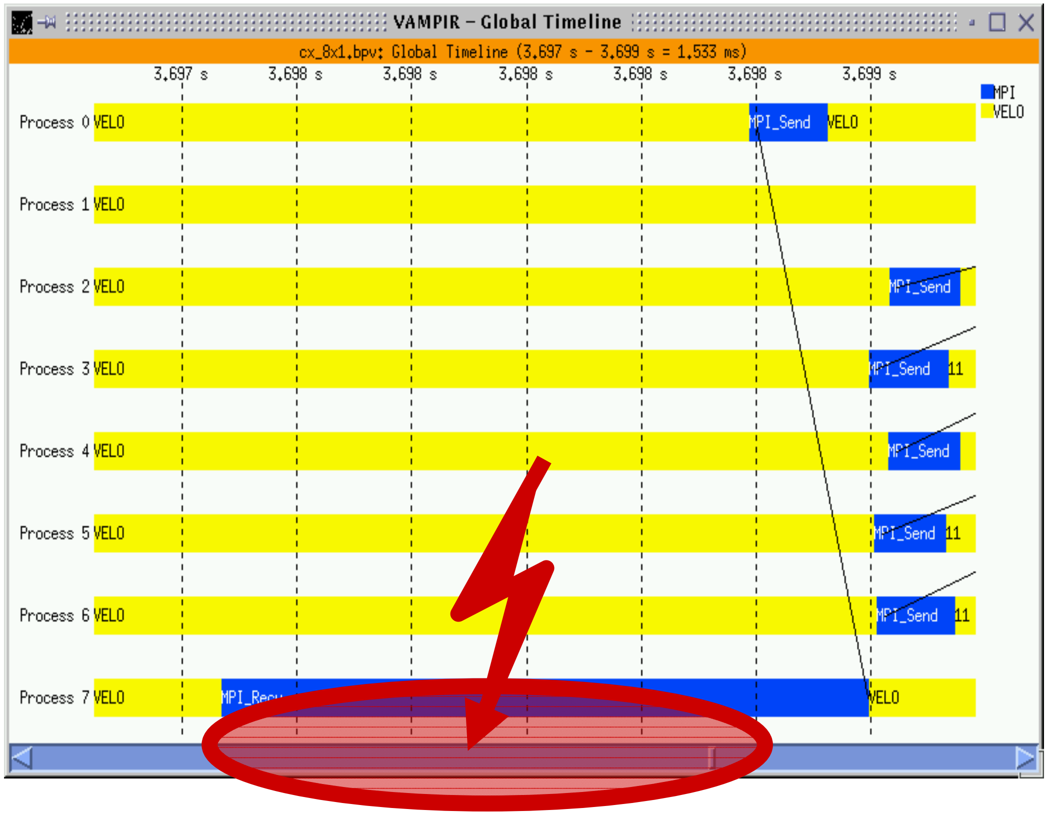
- Paraprof viewer (from the TAU toolset)



Automated Performance Analysis

- Reason for Automation
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: 1.6 million cores
  - Trend to multi-core
- 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

- **Specialized hardware registers** to measure the performance of various aspects of a microprocessor
- Originally used for hardware verification purposes
- Can provide 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)
  - 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 way
  - No guarantees of semantics portably
  - Figuring out what a counter actually counts and if it does so correctly can be hairy

PAPI Hardware Events

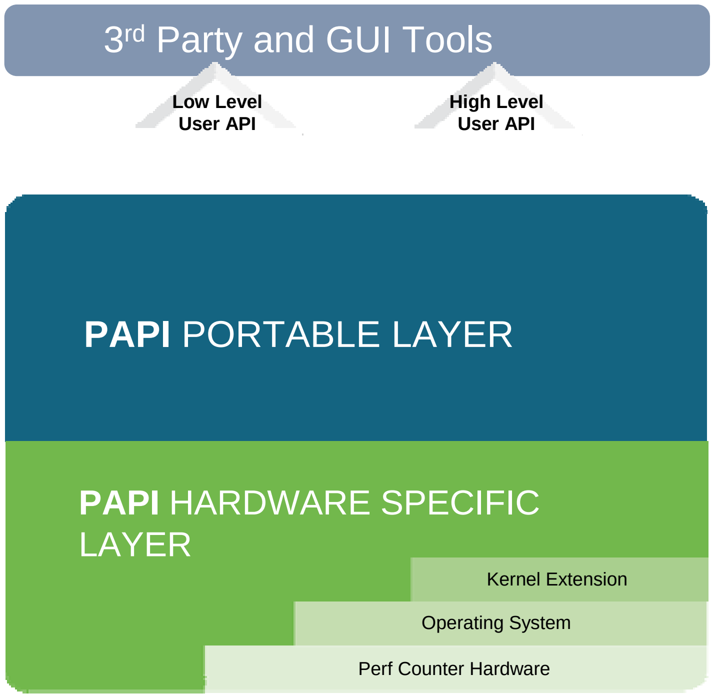
- Preset Events
  - Standard set of over 100 events for application performance tuning
  - No standardization of the exact definitions
  - Mapped to either single or linear combinations of native events on each platform
  - Use **papi_avail** to see what preset events are available on a given platform

- 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 provide facile data collection and visualization.



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

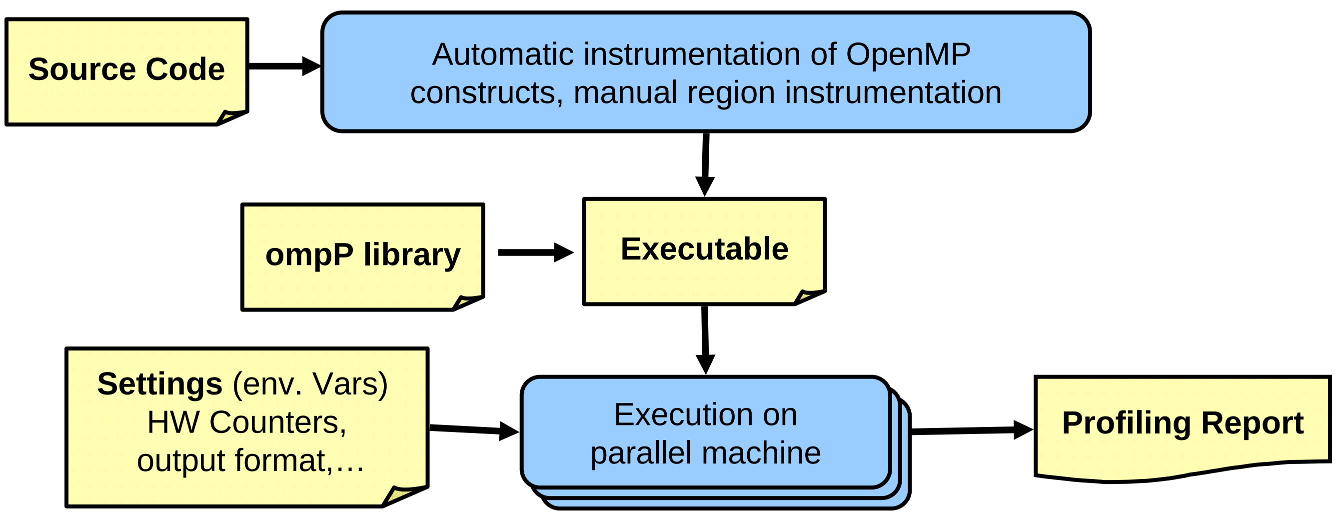
- You can use PAPI directly in your application, but most people use it through tools
- Tool might have a predfined set of counters or lets you select counters through a configuration file/environment variable, etc.
- Tools using PAPI
  - TAU (UO)
  - PerfSuite (NCSA)
  - HPCToolkit (Rice)
  - KOJAK, Scalasca (FZ Juelich, UTK)
  - Open|Speedshop (SGI)
  - ompP (UCB)
  - IPM (LBNL)

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

Callgraph

- Callgraph View
  - 'Callgraph' or 'region stack' of OpenMP constructs
  - Functions can be included by using Opari's mechanism to instrument user defined regions: #pragma pomp inst begin(…), #pragma pomp inst end(…)
- Callgraph profile
  - Similar to flat profile, but with inclusive/exclusive times
- Example:

```
main()
{
#pragma omp parallel
  {
    foo1();
    foo2();
  }
}
```

```
void foo1()
{
   bar();
}
```

```
void foo2()
{
   bar();
}
```

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

Callgraph (2)

- Callgraph display

```
  Incl. CPU time
 32.22 (100.0%)              [APP 4 threads]
 32.06 (99.50%)  PARALLEL     +-R00004 main.c (42-46)
 10.02 (31.10%)   USERREG       |-R00001 main.c (19-21) ('foo1')
 10.02 (31.10%)  CRITICAL       |   +-R00003 main.c (33-36) (unnamed)
 16.03 (49.74%)   USERREG       +-R00002 main.c (26-28) ('foo2')
 16.03 (49.74%)  CRITICAL           +-R00003 main.c (33-36) (unnamed)
```

- Callgraph profiles

```
[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
 TID     execT/I     execT/E       execC
   0        1.00        0.00           1
   1        3.00        0.00           1
   2        2.00        0.00           1
   3        4.00        0.00           1
 SUM       10.01        0.00           4

[*00] critical.ia64.ompp
[+01] R00004 main.c (42-46) PARALLEL
[+02] R00001 main.c (19-21) ('foo1') USER REGION
[=03] R00003 main.c (33-36) (unnamed) CRITICAL
 TID       execT       execC     bodyT/I     bodyT/E      enterT       exitT
   0        1.00           1        1.00        1.00        0.00        0.00
   1        3.00           1        1.00        1.00        2.00        0.00
   2        2.00           1        1.00        1.00        1.00        0.00
   3        4.00           1        1.00        1.00        3.00        0.00
 SUM       10.01           4        4.00        4.00        6.00        0.00
```

Overhead 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

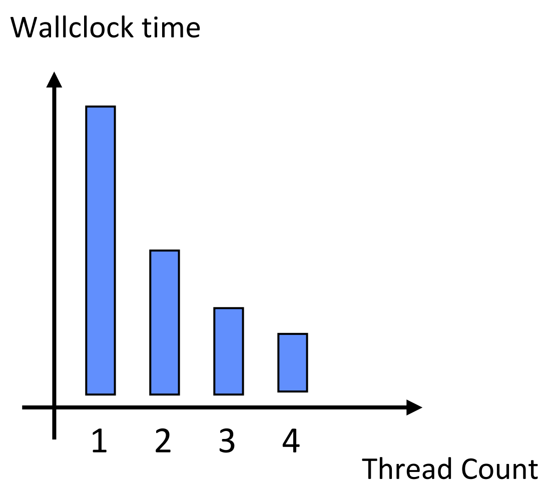
Wallclock time * number of threads

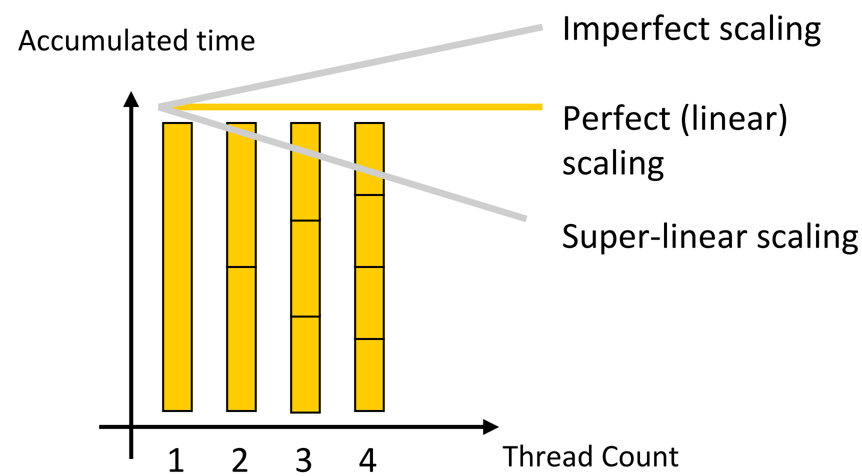
Overhead percentages wrt. this particular parallel region

Overhead percentages wrt. whole program

OpenMP Scalability Analysis

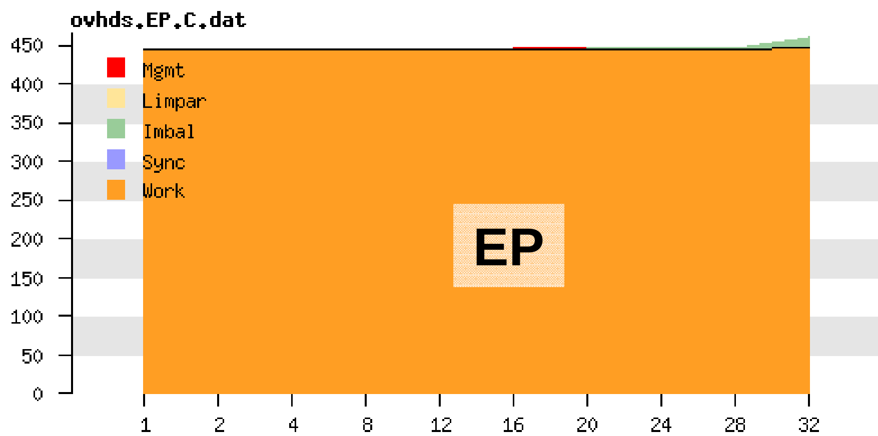
- Methodology
  - Classify execution time into “Work” and four overhead categories: “Thread Management”, “Limited Parallelism”, “Imbalance”, “Synchronization”
  - Analyze how overheads behave for increasing thread counts
  - Graphs show accumulated runtime over all threads for fixed workload (strong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

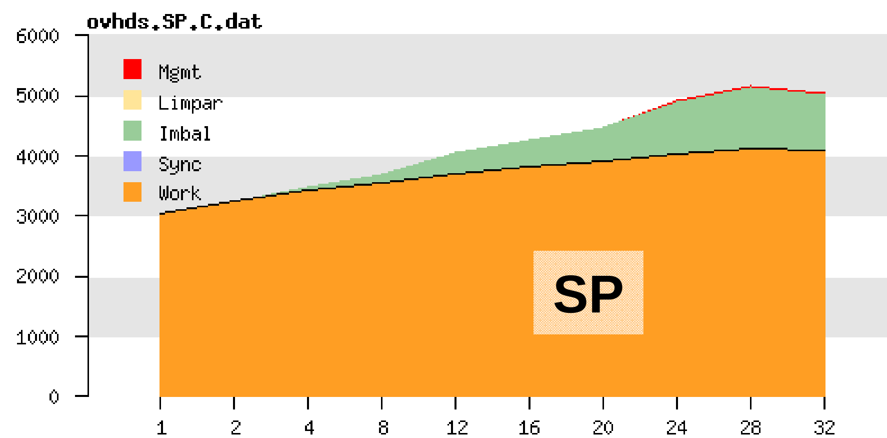




Example

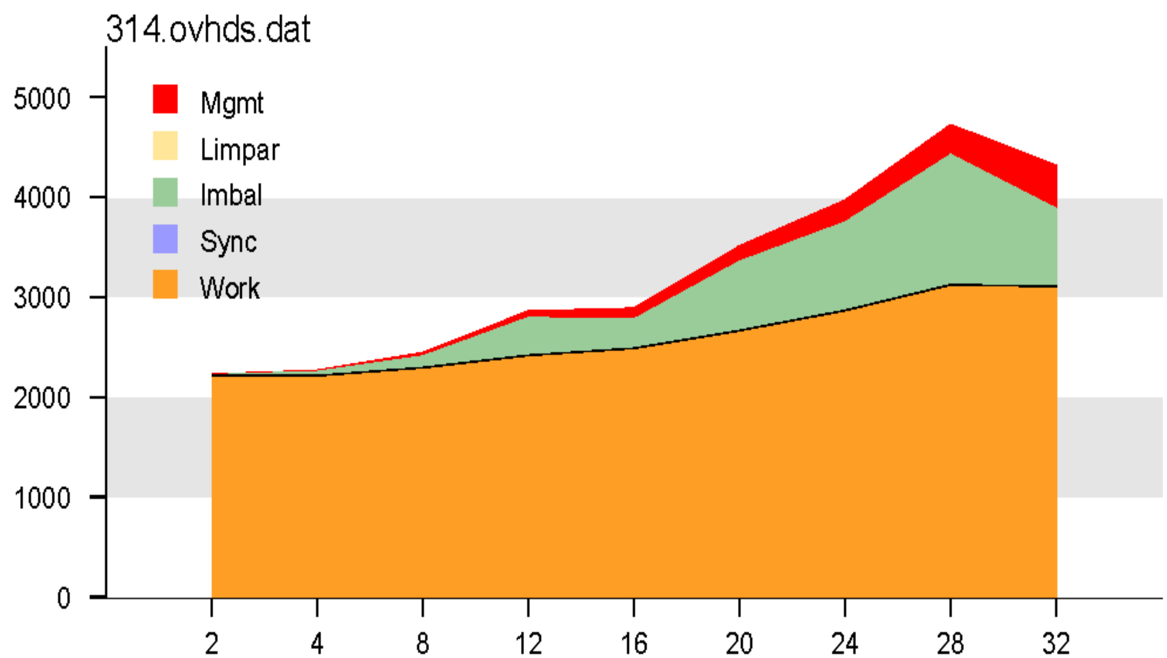
- Example
  - NAS Parallel Benchmarks
  - Class C, SGI Altix machine (Itanium 2, 1.6 GHz, 6MB L3 Cache)





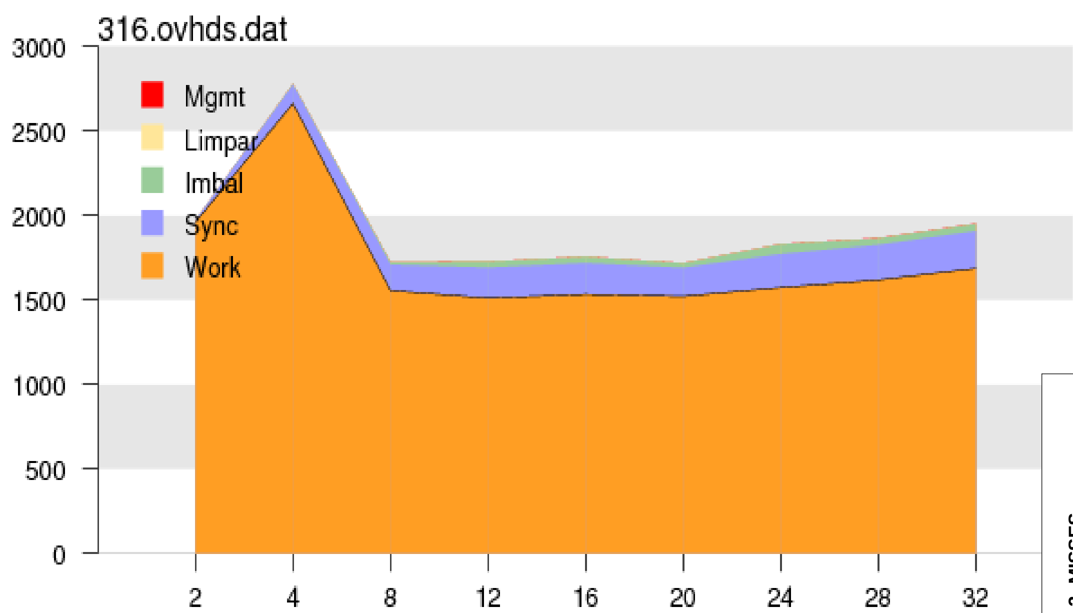
SPEC OpenMP Benchmarks (1)

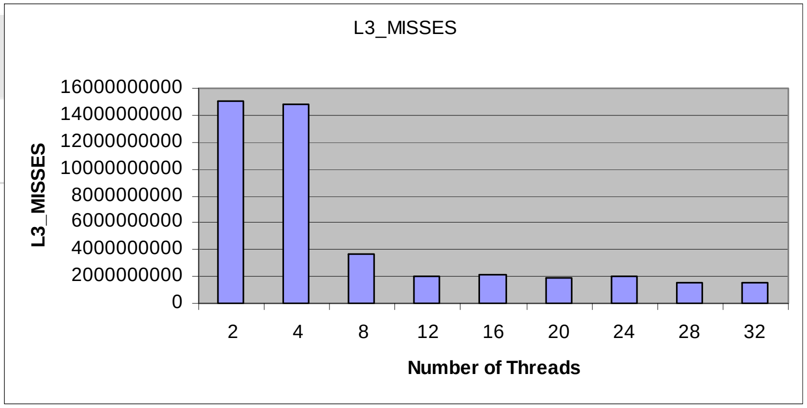
- Application 314.mgrid_m
  - Scales relatively poorly, application has 12 parallel loops, all contribute with increasingly severe load imbalance
  - Smaller load imbalance for thread counts of 32 and 16. Only three loops show this behavior
  - In all three cases, the iteration count is always a power of two (2 to 256), hence thread counts which are not a power of two exhibit more load imbalance



SPEC OpenMP Benchmarks (2)

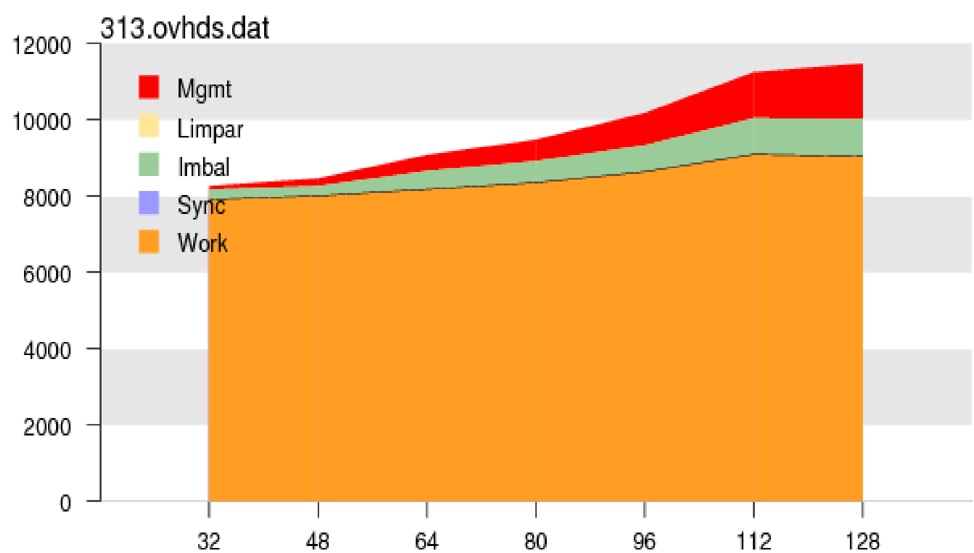
- Application 316.applu
  - Super-linear speedup
  - Only one parallel region (ssor.f 138-209) shows super-linear speedup, contributes 80% of accumulated total execution time
  - Most likely reason for super-linear speedup: increased overall cache size





SPEC OpenMP Benchmarks (3)

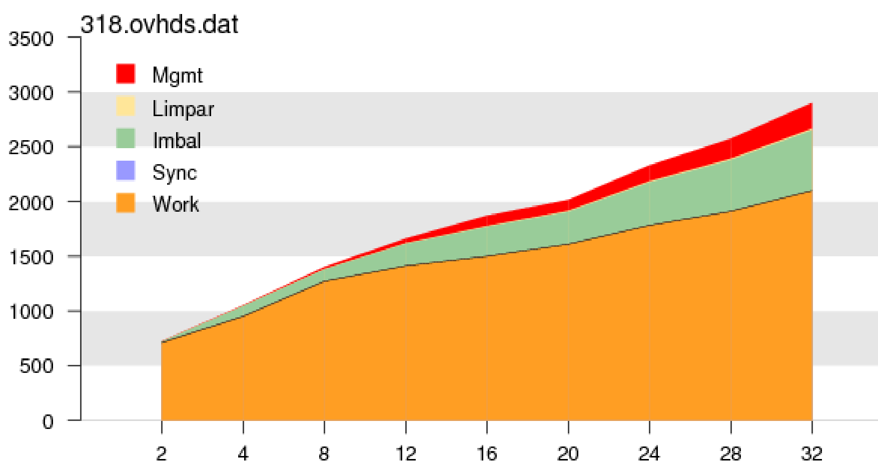
- Application 313.swim
  - Dominating source of inefficiency is thread management overhead
  - Main source: reduction of three scalar variables in a small parallel loop in swim.f 116-126.
  - At 128 threads more than 6 percent of the total accumulated runtime is spent in the reduction operation
  - Time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



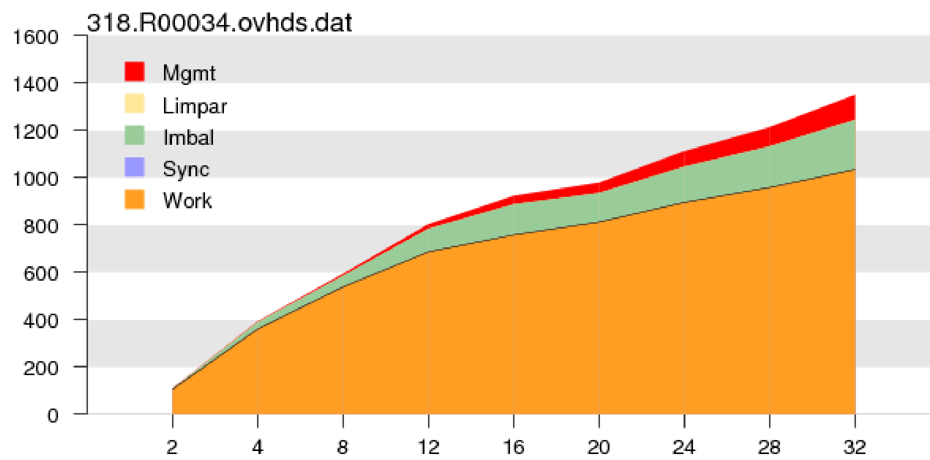
SPEC OpenMP Benchmarks (4)

- Application 318.galgel
  - Scales very poorly, large fraction of overhead not accounted for by ompP (most likely memory access latency, cache conflicts, false sharing)
  - lapack.f90 5081-5092 contributes significantly to the bad scaling
    - » accumulated CPU time increases from 107.9 (2 threads) to 1349.1 seconds (32 threads)
    - » 32 thread version is only 22% faster than 2 thread version (wall-clock time)
    - » 32 thread version parallel efficiency is only approx. 8%

Whole application



Region lapack.f90 5081-5092



Outline

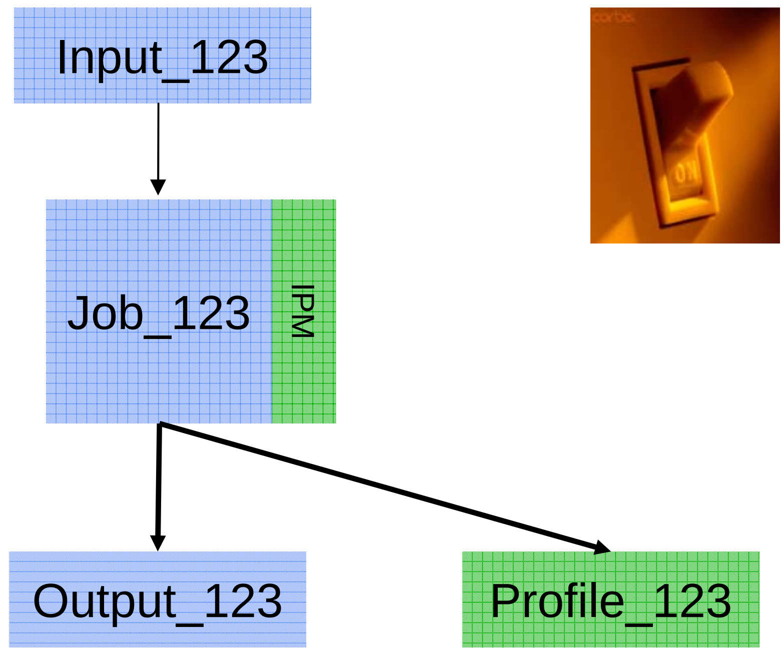
- Motivation

- 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

IPM – Integrated Performance Monitoring

- IPM provides a performance profile of a job
  - „Flip of a switch“ operation
  - http://ipm-hpc.sourceforge.net



IPM: Design Goals

- Provide high-level performance profile
  - *Event inventory*: which events happened and how much time did they take
  - How much time in communication operations
  - Less focus on drill-down into application than other tools
- Efficiency
  - Fixed memory footprint (approx. 1-2 MB per MPI rank)
  - Monitorig data is kept in a hash-table, avoid dynamic memory allocation
  - Low CPU overhead: 1-2 %
- Ease of use
  - HTML, or ASCII-based based output format
  - Flip of a switch, no recompilation, no user instrumentation
  - Portability

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 (in bytes)
    - » 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, report to stdout, write XML log

Using IPM: Basics

- Do “module load ipm”, then run normally
- Upon completion you get

```
##IPMv0.85################################################################
#
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                         mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55              wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17              %comm     : 32.43
# gbytes  : 2.57604e+00 total              gflop/sec : 2.04615e+00 total
#
###########################################################################
```

Maybe that’s enough. If so you’re done.

Have a nice day.

Want more detail? IPM_REPORT=full

```
##IPMv0.85################################################################
#
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                        mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55             wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17             %comm     : 32.43
# gbytes  : 2.57604e+00 total             gflop/sec : 2.04615e+00 total
#
#                          [total]       <avg>          min          max
# wallclock                1373.67     21.4636      21.1087      24.2784
# user                      936.95     14.6398        12.68         20.3
# system                     227.7     3.55781         1.51            5
# mpi                      503.853      7.8727       4.2293      9.13725
# %comm                                32.4268        17.42       41.407
# gflop/sec                2.04614   0.0319709      0.02724      0.04041
# gbytes                   2.57604   0.0402507    0.0399284    0.0408173
# gbytes_tx               0.665125   0.0103926  1.09673e-05    0.0368981
# gbyte_rx                0.659763   0.0103088  9.83477e-07    0.0417372
#
```

Want more detail? IPM_REPORT=full

```
# PM_CYC              3.00519e+11   4.69561e+09   4.50223e+09   5.83342e+09
# PM_FPU0_CMPL        2.45263e+10   3.83223e+08    3.3396e+08   5.12702e+08
# PM_FPU1_CMPL        1.48426e+10   2.31916e+08   1.90704e+08    2.8053e+08
# PM_FPU_FMA          1.03083e+10   1.61067e+08   1.36815e+08   1.96841e+08
# PM_INST_CMPL        3.33597e+11   5.21245e+09   4.33725e+09   6.44214e+09
# PM_LD_CMPL          1.03239e+11   1.61311e+09   1.29033e+09   1.84128e+09
# PM_ST_CMPL          7.19365e+10   1.12401e+09   8.77684e+08   1.29017e+09
# PM_TLB_MISS         1.67892e+08   2.62332e+06   1.16104e+06   2.36664e+07
#
#                         [time]       [calls]        <%mpi>      <%wall>
# MPI_Bcast              352.365          2816         69.93        22.68
# MPI_Waitany            81.0002        185729         16.08         5.21
# MPI_Allreduce          38.6718          5184          7.68         2.49
# MPI_Allgatherv         14.7468           448          2.93         0.95
# MPI_Isend              12.9071        185729          2.56         0.83
# MPI_Gatherv            2.06443           128          0.41         0.13
# MPI_Irecv                1.349        185729          0.27         0.09
# MPI_Waitall           0.606749          8064          0.12         0.04
# MPI_Gather           0.0942596           192          0.02         0.01
###############################################################################
```

IPM: XML log files

- There's a lot more information in the logfile than you get to stdout. A logfile is written that has the hash table, switch traffic, memory usage, executable information, ...

- Parallelism in writing of the log (when possible)

- The IPM logs are durable performance profiles serving

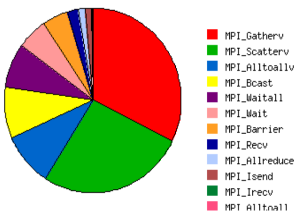
  – HPC center production needs:
    https://www.nersc.gov/nusers/status/llsum/
    http://www.sdsc.edu/user_services/top/ipm/

  – HPC research: ipm_parse renders txt and html
    http://www.nersc.gov/projects/ipm/ex3/

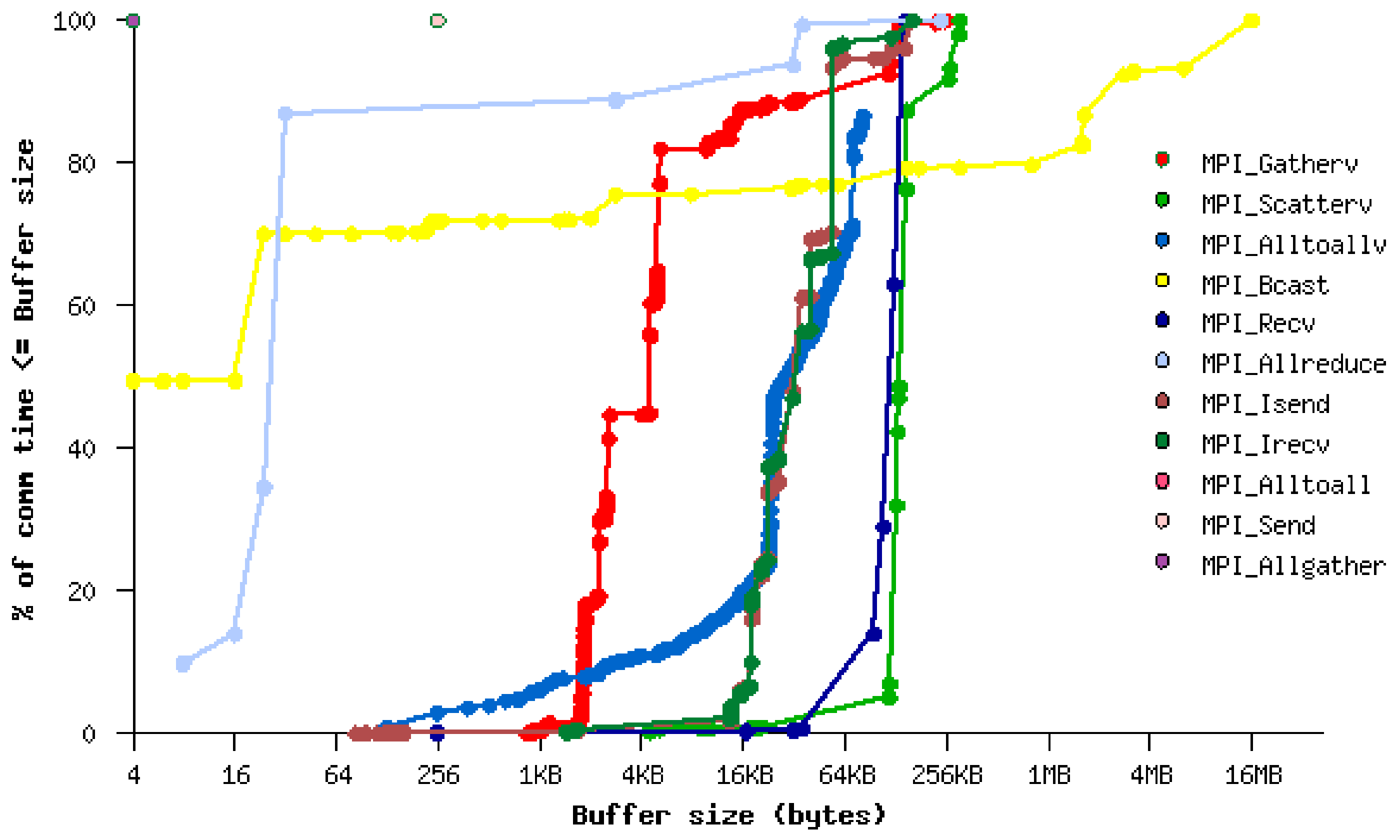
  – your own XML consuming entity, feed, or process

Message Sizes : CAM 336 way

per MPI call

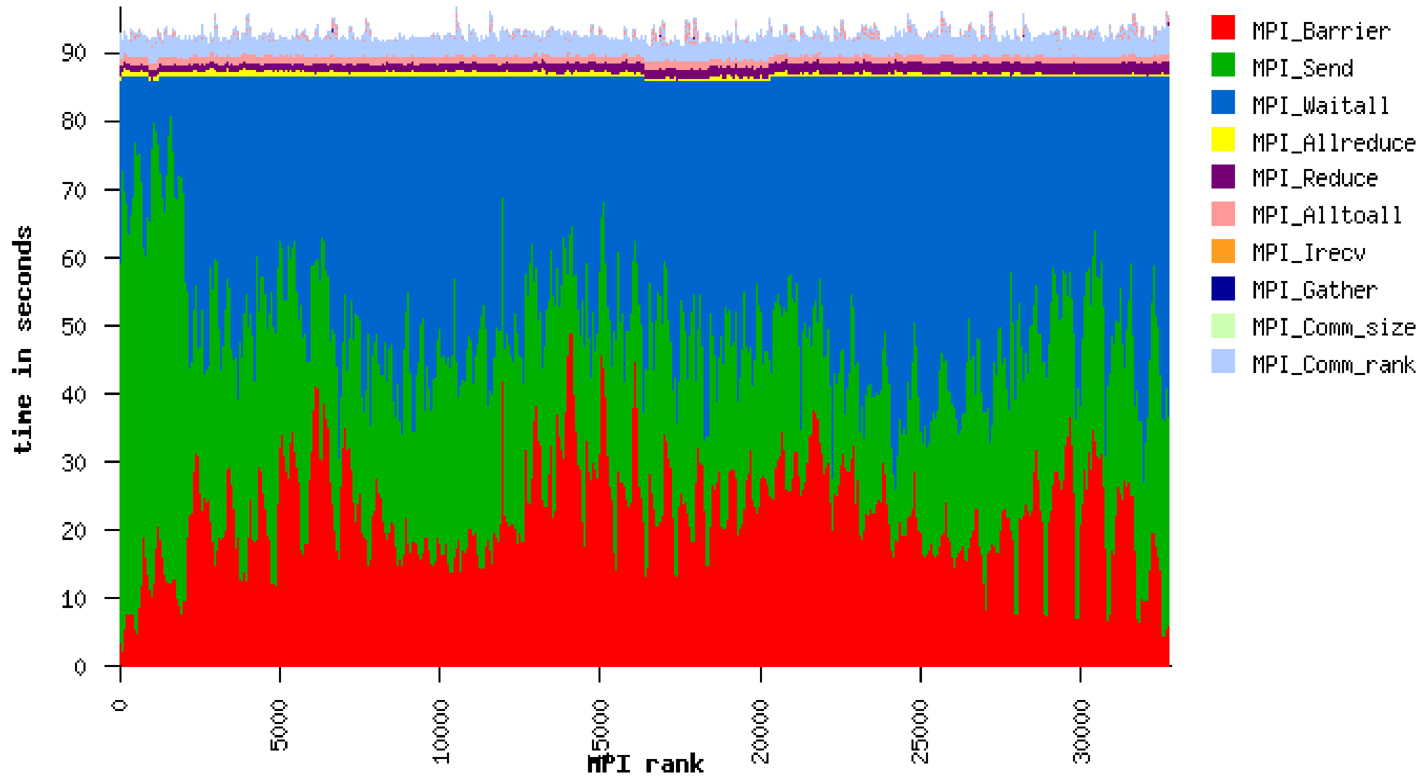


per MPI call & buffer size

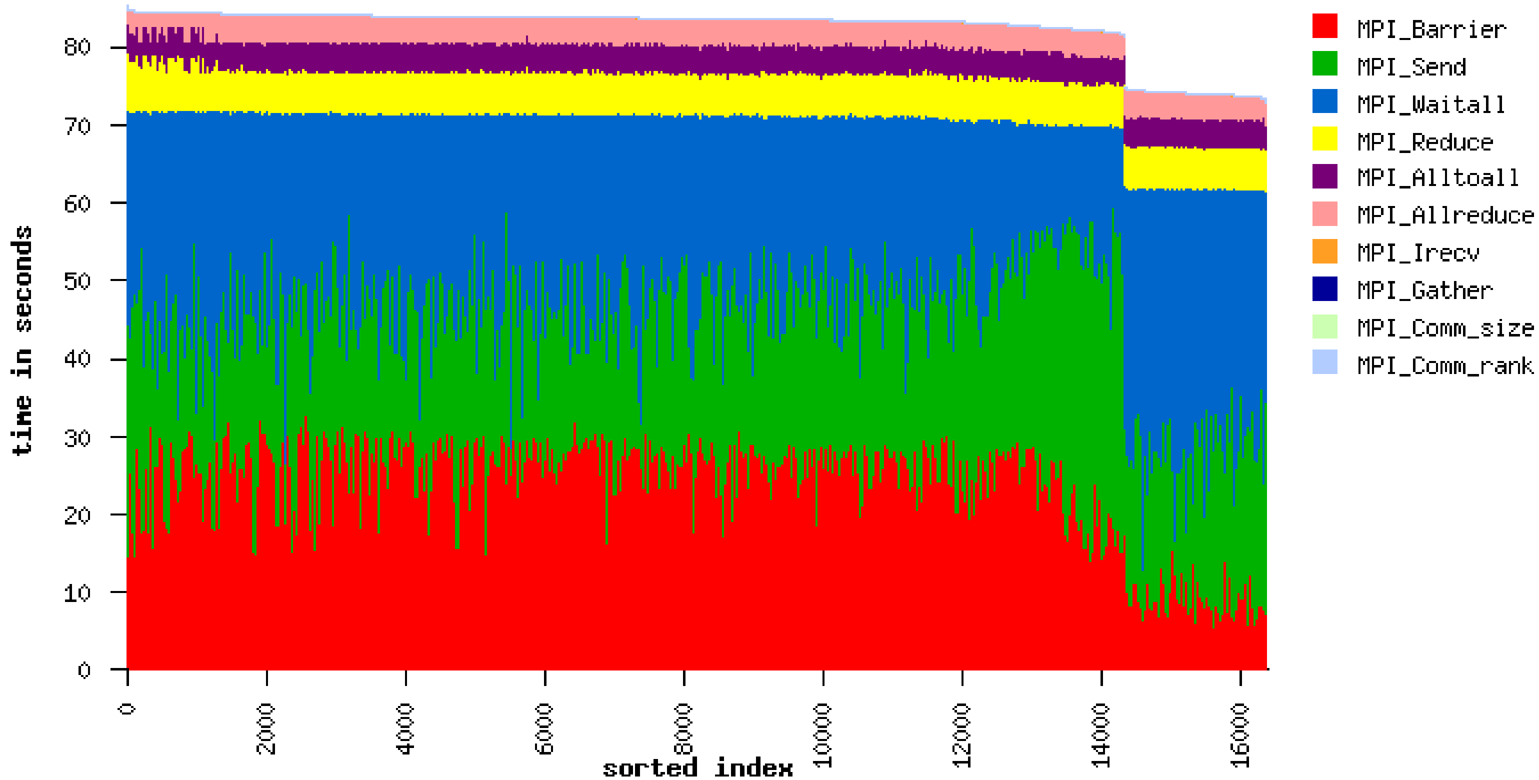


Scalability: Required

32K tasks AMR code



More than a pretty picture



Which problems should be tackled with IPM?

- Performance Bottleneck Identification
  - Does the profile show what I expect it to?
  - Why is my code not scaling?
  - Why is my code running 20% slower than I expected?
- Understanding Scaling
  - Why does my code scale as it does ?
- Optimizing MPI Performance
  - Combining Messages

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)

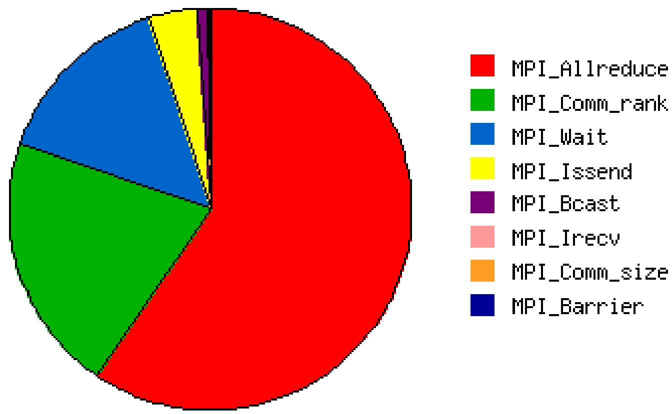
When to reach for another tool

- Full application tracing
- Looking for hotspots on the statement level in code
- Want to step through the code
- Data structure level detail
- Automated performance feedback

What's wrong here?

Communication

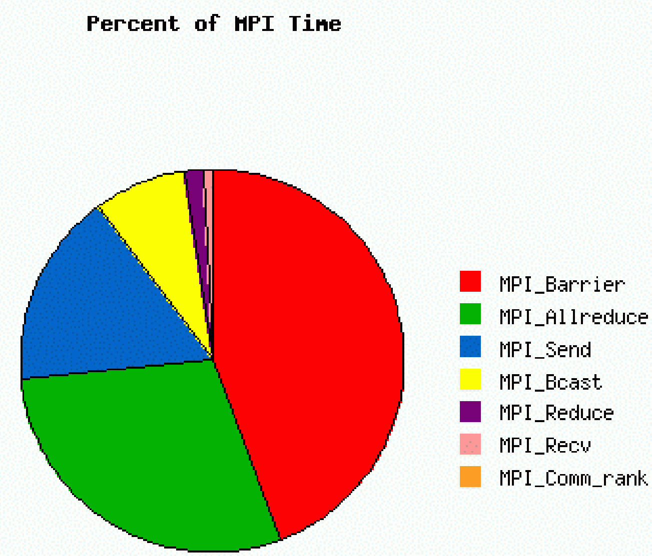
% of MPI Time



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

Function	Total calls	Total time (sec)		Total buffer size (MB)	Avg. Buffer Size/call (Bytes)
MPI_Barrier	6.02e+05	3.48e+05	44.23%	0	0
MPI_Allreduce	3.18e+07	2.31e+05	29.33%	3.61e+05	11,936
MPI_Send	1.29e+08	1.29e+05	16.36%	5.24e+04	426
MPI_Bcast	5.73e+07	6.08e+04	7.73%	5.39e+04	987
MPI_Reduce	1.08e+08	1.24e+04	1.58%	1.66e+05	1,620
MPI_Recv	1.29e+08	6.11e+03	0.78%	5.24e+04	426
MPI_Comm_rank	1.14e+03	5.92e-01	7.52e-05%	0	0
MPI_Comm_size	6.66e+02	0	0%	0	0



Is MPI_Barrier time bad? Probably. Is it avoidable?

- The stray / unknown / debug barrier
- Barriers used for I/O ordering

Summary

- Performance montioring concepts
  - Instrument, measure, analyze
  - Profiling/tracing, sampling, direct measurment

- Tools
  - PAPI, ompP, IPM as examples

- Lots of other tools
  - Vendor tools: Cray PAT, Sun Studio, Intel Thread Profiler, Vtune, PTU,…
  - Portable tools: TAU, Perfsuite, Paradyn, HPCToolkit, Kojak, Scalasca, Vampir, oprofile, gprof, …

Thank you for your attention!