Hardware Performance Analysis on the Opteron with PAPI

Philip J. Mucci, mucci@cs.utk.edu

ClusterWorld 2004, San Jose, CA

INNOVATIVE COMPUTING LABORATORY
UNIVERSITY OF TENNESSEE
DEPARTMENT OF COMPUTER SCIENCE

Outline

- Hardware Performance Analysis
- Opteron Performance Counter Hardware
- Description of PerfCtr and PAPI
- Building and Installing PAPI and PerfCtr
- Some sample PAPI instrumentation
- Building and Installing 2 performance tools, papiex and TAU.
- How to use those tools on your code
- Links to other tools.

Performance Evaluation

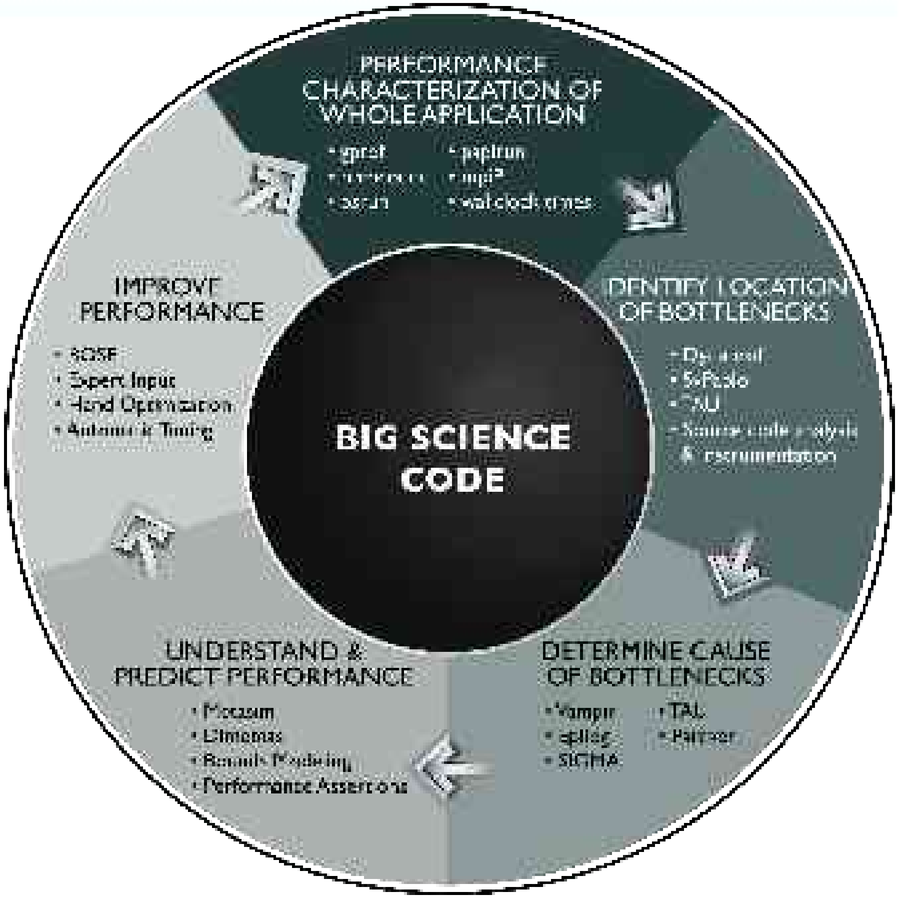
- Traditionally, performance evaluation has been somewhat of an art form:
  - Limited set of tools (time & -p/-pg)
  - Major differences between systems
  - Lots of guesswork as to what was 'behind the numbers'
- Today, the situation is different.
  - Hardware support for performance analysis
  - A wide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

Performance

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

- William Gropp
Argonne National Lab

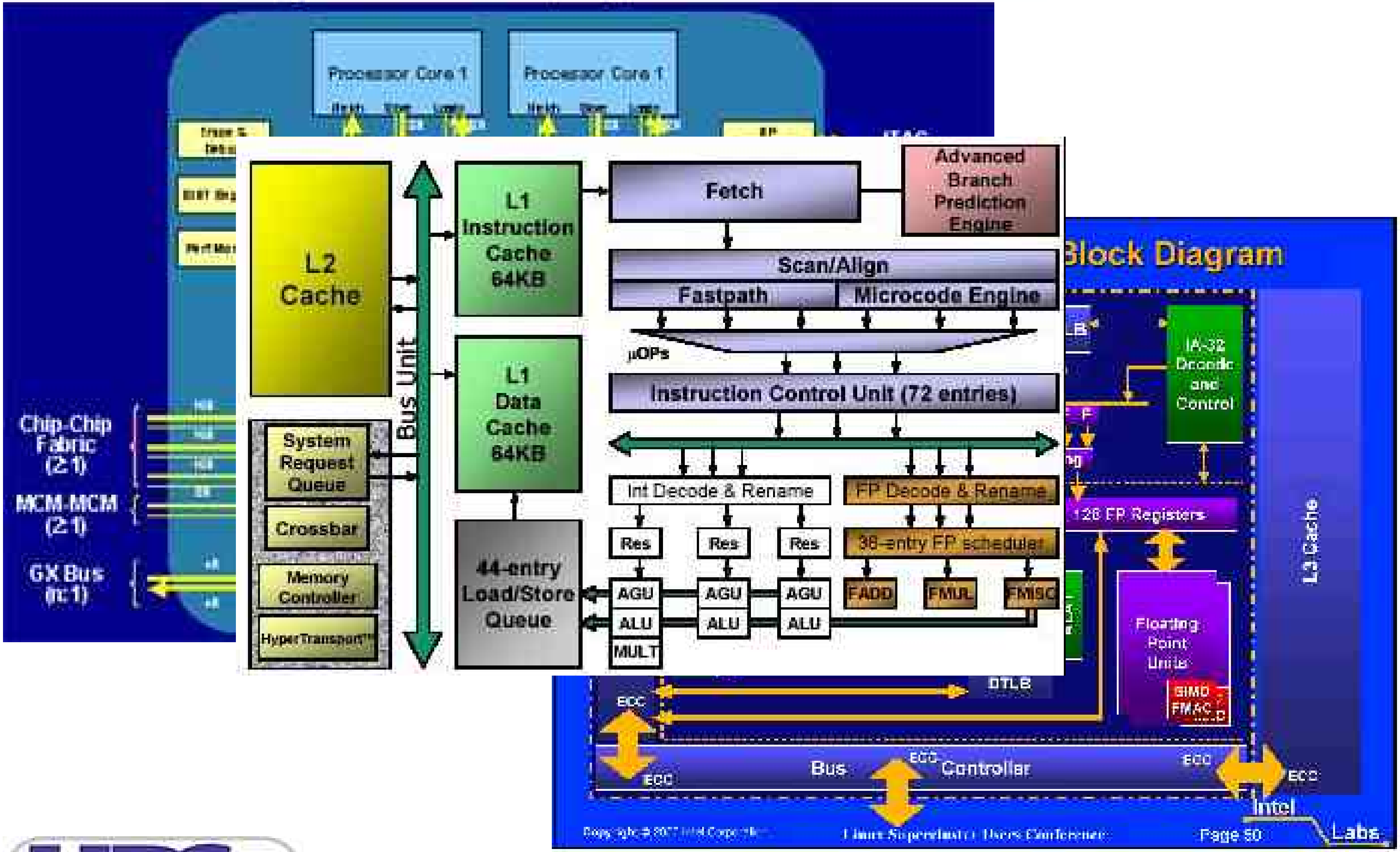
A New Performance Science



Performance Evaluation Research Center at LBL

http://perc.nersc.gov

Rising Complexity



Rising Processor Complexity

- No longer can we easily trace the execution of a segment of code.
  - Static/Dynamic Branch Prediction
  - 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.

Performance Counters

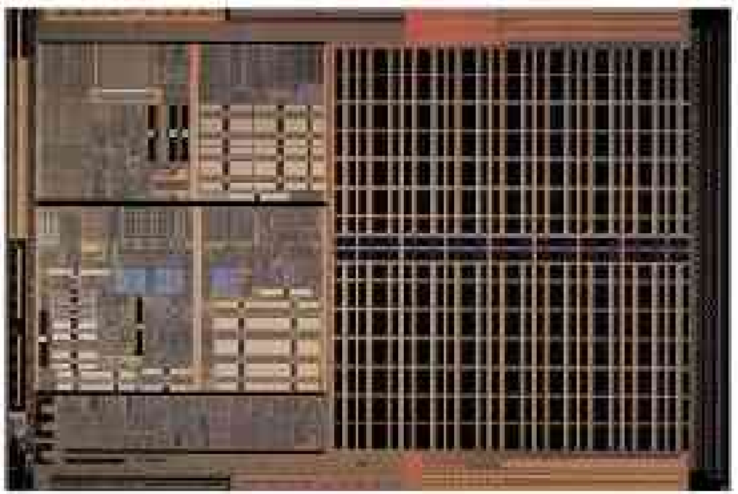
- Performance Counters are hardware registers dedicated to counting certain types of events within the processor or system.
  - Usually a small number of these registers (2,4,8)
  - Sometimes they can count a lot of events or just a few
  - Symmetric or asymmetric
- Each register has an associated control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

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, Pentium
  - SGI MIPS R1xK Series
  - Sun UltraSparc II+
  - And many others…

Opteron Performance Counters

- The Opteron has 4 symmetric performance counter registers.
- Each can count 58+ events, each of which has a multitude of different flags that can be set.
- Supports hardware interrupt on counter overflow.



Opteron Performance Counters

- Example of hardware events:
  - DISPATCHED_FPU_OPS
  - NO_FPU_OPS (cycles with no-FPU ops retired)
  - LS_BUFFER_FULL (load-store buffer full)
  - CYCLES
- For a full description of all the events, see AMD's BIOS and Kernel Developer's Guide, Document #26094

Access to Performance Counters

- On some platforms there are APIs, on others nothing is available.
- When the APIs do exist, they are usually:
  - Not appropriate for the application engineer.
  - Not very well documented.
- The same can be said for the counter hardware itself.
  - Often not well documented.
  - Rarely are the events verified by the engineers.

PAPI

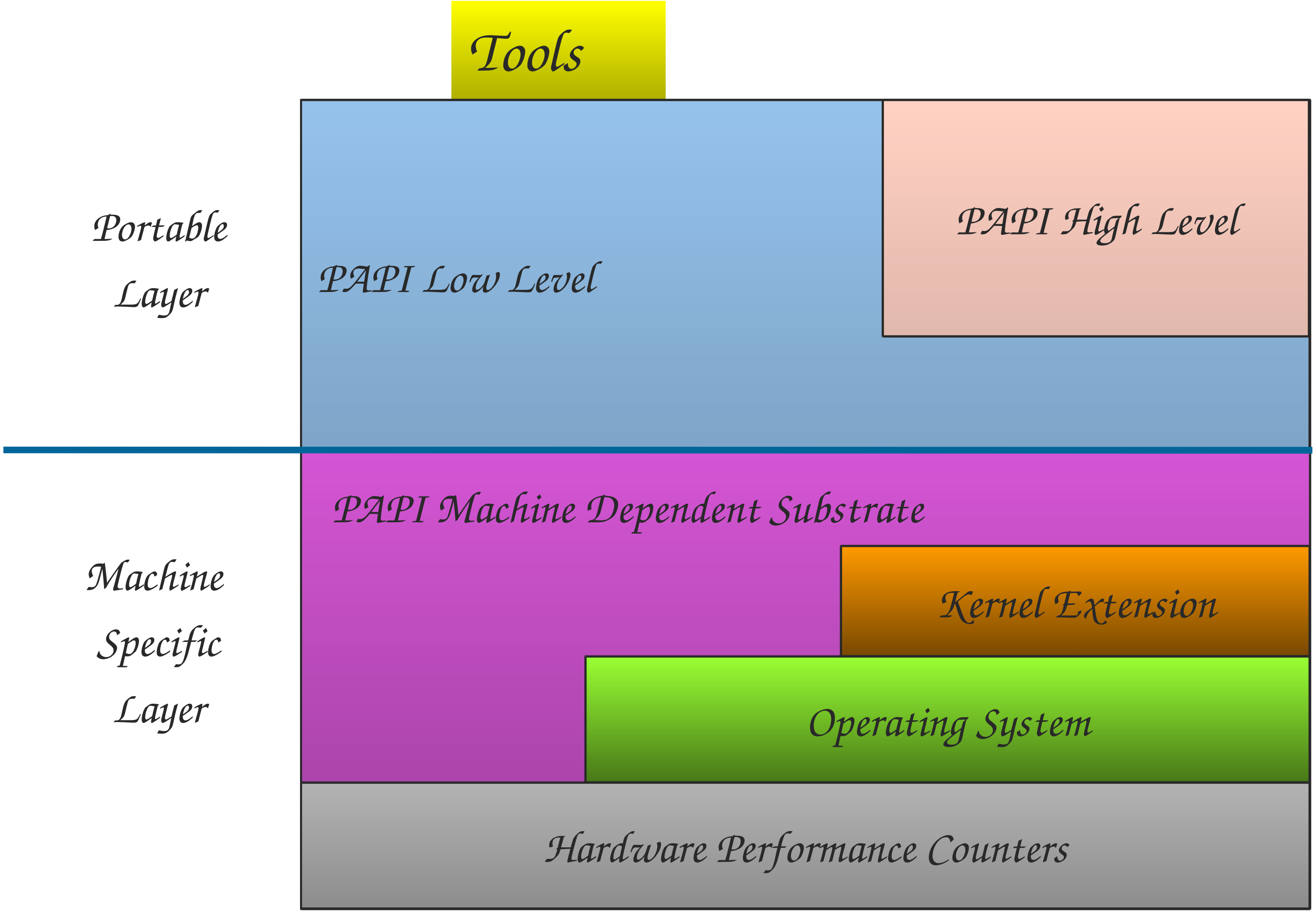
- **P**erformance **A**pplication **P**rogramming **I**nterface
- The purpose of PAPI is to implement a standardized portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- The goal of PAPI is to facilitate the optimization of parallel and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

http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

PAPI Features

- Standardized Access to Performance Counters
- Preset Performance Metrics
- Easy Access to Platform-Specific Metrics
- Multiplexed Event Measurement
- Dispatch on Overflow
- Full SVR4 Profiling
- Bindings for C, Fortran, Matlab, and Java

PAPI Implementation



Tools

Portable Layer

PAPI Low Level

PAPI High Level

Machine Specific Layer

PAPI Machine Dependent Substrate

Kernel Extension

Operating System

Hardware Performance Counters

Preset Events

- PAPI supports 103 preset events, defined in papiStdEventDefs.h
- 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
  - ctests/avail -a will list the PAPI preset events available
- PAPI also supports presets that may be derived from the underlying hardware metrics
  - Floating Point Operations is PAPI_FP_OPS

Native Events

- Any event countable by the CPU can be counted even if there is no matching preset PAPI event.
- Same interface as when setting up a preset event, but we use one additional call to translate a CPU-specific moniker into a PAPI event definition.
- See ctests and ftests directory in the distribution.

PAPI Counter Interfaces

- PAPI provides 2 interfaces to the underlying counter hardware:

1. The high level interface provides the ability to start, stop and read the counters for a specified list of events.
2. The low level interface manages hardware events in user defined groups called *EventSets*, and provides access to advanced features.

High-level Interface

- Meant for application programmers wanting coarse-grained measurements
- Not thread safe
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level

High-level API

- C interface
  PAPI_start_counters
  PAPI_read_counters
  PAPI_stop_counters
  PAPI_accum_counters
  PAPI_num_counters
  PAPI_flops

- Fortran interface
  PAPIF_start_counters
  PAPIF_read_counters
  PAPIF_stop_counters
  PAPIF_accum_counters
  PAPIF_num_counters
  PAPIF_flops

PAPI_flops

- `int PAPI_flops(float *real_time, float *proc_time, long_long *flpins, float *mflops)`
  - Only two calls needed, PAPI_flops before and after the code you want to monitor
  - real_time is the wall-clocktime between the two calls
  - proc_time is the “virtual” time or time the process was actually executing between the two calls (not as fine grained as real_time but better for longer measurements)
  - flpins is the total floating point instructions executed between the two calls
  - mflops is the Mflop/s rating between the two calls
  - If *flpins == -1 the counters are reset

High-level Interface Setup

- `int PAPI_num_counters(void)`
  - Initializes PAPI (if needed)
  - Returns number of hardware counters
- `int PAPI_start_counters(int *events, int len)`
  - Initializes PAPI (if needed)
  - Sets up an event set with the given counters
  - Starts counting in the event set
- `int PAPI_library_init(int version)`
  - Low-level routine implicitly called by above

Controlling the Counters

- `PAPI_stop_counters(long_long *vals, int  alen)`
  - Stop counters and put counter values in array
- `PAPI_accum_counters(long_long *vals, int  alen)`
  - Accumulate counters into array and reset
- `PAPI_read_counters(long_long *vals, int  alen)`
  - Copy counter values into array and reset counters
- `PAPI_flops(float *rtime, float *ptime, long_long *flpins, float *mflops)`
  - Wallclock time, process time, FP ins since start,
  - Mflop/s since last call

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware & the memory
- Allows native events
- Can perform counter multiplexing
- Callbacks on counter overflow
- SVR4 compatible profil() interface
- Approximately 54 functions

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

Example PAPI Data: Parallel Ocean Program Performance
x1 Data Set, 2x2 Procs, 10 Steps

Raw Data	Debug	Optimized
PAPI_LD_INS	1.21E+011	2.104E+10
PAPI_SR_INS	2.02E+010	7.783E+09
PAPI_BR_INS	8.64E+009	5.043E+09
PAPI_FP_INS	2.21E+010	2.251E+10
PAPI_FMA_INS	1.04E+010	1.007E+10
PAPI_FPU_FDIV		2.551E+08
PAPI_FPU_FSQRT		1.317E+08
PAPI_TOT_INS	3.28E+011	6.257E+10
PAPI_TOT_CYC	3.63E+011	6.226E+10
PAPI_L1_LDM	1.03E+009	1.011E+09
PAPI_L1_STM	3.54E+008	3.475E+08
PAPI_L2_LDM	6.94E+008	6.894E+08
PAPI_FPU_IDL	1.66E+011	1.411E+10
PAPI_LSU_IDL	4.06E+010	1.483E+10
PAPI_MEM_RCY	1.03E+011	1.368E+10
PAPI_MEM_SCY	1.26E+011	2.413E+10
PAPI_STL_CCY	2.01E+011	3.367E+10

Metric	Debug	Optimized
% Ld Ins	36.86	33.63
% Sr Ins	6.17	12.44
% Br Ins	2.63	8.06
% FP Ins	6.75	35.98
% FMA Ins	3.16	16.09
% FP Divide		0.41
% FP SQRT		0.21
MFLIPS	12.19	72.31
% MFLIPS Peak	3.05	18.08
IPC	0.90	1.00
Mem Opts/FLIP	6.38	1.28
% L1 Ld HR	99.15	95.19
% L1 Sr HR	98.25	95.54
% L2 Ld HR	99.43	96.72
% FPU Idle Cyc	45.77	22.66
% LSU Idle Cyc	11.17	23.82
% Ld Stall Cyc	28.28	21.97
% Sr Stall Cyc	34.59	38.76
% No Ins. Cyc	55.25	54.08

Supported Architectures

- AMD Athlon and Opteron
- Cray T3E and X1
- HP Alpha (caveats)
- IBM POWER3, POWER4
- Intel Pentium Pro,II,III + 4, Itanium 1 + 2
- MIPS R10K, R12K, R14K
- Sun UltraSparc I, II, III

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- About 56 functions (http://icl.cs.utk.edu/projects/papi/files/html_man/papi.html#4)
- Obtain information about the executable and the hardware
- Thread-safe
- Fully programmable (native events)
- Multiplexing
- Callbacks on counter overflow
- Profiling

Low-level Functionality

- Library initialization

  PAPI_library_init, PAPI_thread_init, PAPI_multiplex_init, PAPI_shutdown

- Timing functions: highest resolution and accuracy

  PAPI_get_real_usec, PAPI_get_virt_usec PAPI_get_real_cyc, PAPI_get_virt_cyc

- Inquiry functions
- EventSet management
- Thread specific data pointers
- Simple fast lock/unlock operators

  PAPI_lock/PAPI_unlock

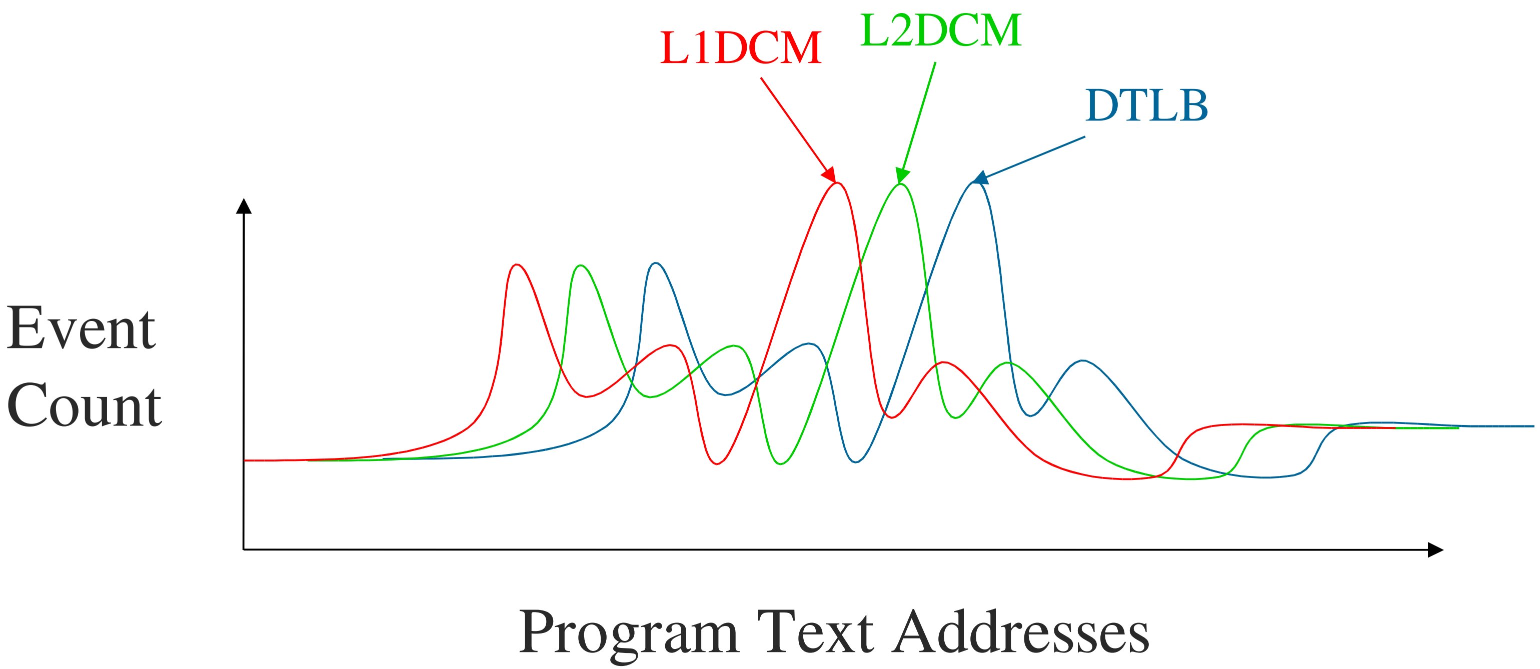
Callbacks on Counter Overflow

- PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.
- For systems that do not support counter overflow at the OS level, PAPI sets up a high resolution interval timer and installs a timer interrupt handler.

Statistical Profiling

- Callbacks on counter overflow allow us to do interesting things.
- PAPI provides support for SVR4-compatible execution profiling based on any counter event.
- PAPI_profil() creates a histogram of overflow counts for a specified region of the application code.

Counter Profiling using Overflow



PAPI 3.0

- Lower Measurement Overheads
- Overflow and Profiling on Multiple Simultaneous Events
- Complete and Easy Access to All Platform-Specific Metrics
- High level API is now thread safe
- Internal timer/signal/thread abstractions

PAPI Performance on Opteron

- Wallclock cycle cost of a PAPI_start(),PAPI_stop() sequence is approximately 4500 cycles.
- Cost of PAPI_read() is 116 cycles.
- Cost of Wallclock Cycle timer is 25 cycles, resolution 1 cycle.
- Cost of Virtual Cycle timer is 28 cycles, resolution 1 cycle.

The PerfCtr Kernel Patch

- To use the hardware counters, it is necessary to patch the Linux kernel.
  - Allows measurement only the process or thread of interest with no interference.
  - Allows more than one user to use the counter hardware.
  - Ensures 64-bit values.
  - Saves/restores counter information at context switch.
    - If in use by a process, only a few hundred cycles difference per context switch.
  - Allows dispatch of signal on counter overflow.

The PerfCtr Kernel Patch

- Originally, PAPI used a system-call based kernel patch.
- A better patch, the PerfCtr kernel patch, was developed by Mikael Pettersson of Uppsala.
  - Biggest difference was that it provided memory mapped access of the counters for ultra-fast access.
- The patch consists of a patch to the kernel and a small shared library, libperfctr.so.

Downloading PAPI and PerfCtr

- The release version of PAPI comes with a recent version of PerfCtr in papi/src/perfctr-2.6.x.
- Download the latest version of PAPI 3 from the PAPI website.
  - Latest and greatest from the CVS tree. (recommended)

```
Cvs -d :pserver:anonymous@icl.cs.utk.edu:/cvs/homes/papi login
<no password>
Cvs -d :pserver:anonymous@icl.cs.utk.edu:/cvs/homes/papi co papi/src
```

  - Last stable release: 3.0 Beta 2

```
Wget http://icl.cs.utk.edu/projects/papi/downloads/papi-3.0-beta2.tar.gz
```

Difference between oprofile and PAPI/PerfCtr

- Oprofile
  - PC Sampling only, no aggregate measurement.
  - Handles samples in kernel space as well as application and library space.
  - Configuration is set by root, users cannot change what counters are used.
  - Ideal for tuning of system throughput, not suited to a production environment where users need to tune their codes with different tools.

Patching the Linux Kernel

- Let PDIR be the directory containing the PerfCtr distribution.
- Cd to your Linux kernel directory
  - Cd /usr/src/linux-2.4
- Patch the kernel
  - PDIR/update-kernel
  - If there is an error: PDIR/update-kernel –test –patch=version where version is found in PDIR/patches
- Reconfigure the Kernel

Enable PerfCtr Support

- Reconfigure the Kernel
  - Make oldconfig
- Answer Y to the questions about PerfCtr.
  - CONFIG_PERFCTR=y
  - CONFIG_PERFCTR_GLOBAL=y
  - CONFIG_PERFCTR_VIRTUAL=y
- For single CPU systems, make sure you enable uniprocessor APIC support.
- Recompile and install the new kernel.

Enable APIC Support

- For single CPU systems, make sure you enable uniprocessor APIC support. Either
  - Vi .config -or-
  - Make config
  - CONFIG_X86_GOOD_APIC=y
  - CONFIG_X86_UP_APIC=y
  - CONFIG_X86_UP_IOAPIC=y
  - CONFIG_X86_LOCAL_APIC=y
  - CONFIG_X86_IO_APIC=y
- Recompile and install the new kernel.

Reboot with the PerfCtr Kernel

- Reboot
  - Look for PerfCtr messages in your boot log like:
  - perfctr: driver 2.6.2, cpu type AMD K8C at 1794933 kHz
- Make the device file
  - Mknod /dev/perfctr c 10 182
  - Chmod 644 /dev/perfctr
- Build the PerfCtr library
  - Cd PDIR
  - make

Test the PerfCtr Patch

- Build the PerfCtr library
  - Cd PDIR; make
- Test the example program: (make sure pcint is there for APIC)
  - `examples/self/self`
  - `examples/perfex/perfex -i`

```
PerfCtr Info:
abi_version               0x05000500
driver_version            2.6.2
cpu_type                  15 (AMD K8 Revision C)
cpu_features              0x7 (rdpmc,rdtsc,pcint)
cpu_khz                   1794933
tsc_to_cpu_mult           0 (unspecified, assume 1)
cpu_nrctrs                4
cpus                      [0], total: 1
cpus_forbidden            [], total: 0
```

PerfCtr Perfex Command

- Perfex is a program that directly manipulates the control registers and reads the values of a forked process.
  - No thread/fork support
  - All events specified in hexadecimal
  - Useful for testing, not for performance analysis
- Now build PAPI
  - Cd papi/src
  - Make -f Makefile.linux-opteron

Build and Test PAPI

- Now build PAPI:
  - Cd papi/src
  - Make -f Makefile.linux-opteron
- Do a quick test that it works:
  - ctests/zero

```
Test case 0: start, stop.
PAPI_FP_INS  :       40000194
PAPI_TOT_CYC :      227791408
Real usec    :         127205
Real cycles  :      228205736
zero.c                                   PASSED
```

- Optionally run all the test cases and wait.
  - Sh run_tests.sh

Install PAPI and PerfCtr

- Install PAPI and PerfCtr:
  - Make -f Makefile.linux-opteron install-all PREFIX=/usr
- Replace /usr with your local prefix of installation.
  - /usr/local
  - $HOME/usr
- Verify library paths:
  - Ldd <PREFIX>/lib/libpapi.so should show all libraries as found.
  - If PAPI and PerfCtr are not installed in the standard place, you'll need to set your LD_LIBRARY_PATH to the location of the libraries. (<PREFIX>/lib)

Preset Listing from tests/avail -a

```
Test case 8: Available events and hardware information.
-------------------------------------------------------------------------
Vendor string and code   : AuthenticAMD (2)
Model string and code    : AMD K8 Revision C (15)
CPU Revision             : 8.000000
CPU Megahertz            : 1794.932983
CPU's in this Node       : 1
Nodes in this System     : 1
Total CPU's              : 1
Number Hardware Counters : 4
Max Multiplex Counters   : 32
-------------------------------------------------------------------------
Name            Derived Description (Mgr. Note)
PAPI_L1_DCM     No      Level 1 data cache misses (DC_MISS)
PAPI_L1_ICM     No      Level 1 instruction cache misses (IC_MISS)
PAPI_L2_DCM     No      Level 2 data cache misses (BU_L2_FILL_MISS_DC)
PAPI_L2_ICM     No      Level 2 instruction cache misses (BU_L2_FILL_MISS_IC)
PAPI_L1_TCM     Yes     Level 1 cache misses (DC_MISS, IC_MISS)
PAPI_L2_TCM     Yes     Level 2 cache misses (BU_L2_FILL_MISS_IC, BU_L2_FILL_MISS_DC)
PAPI_FPU_IDL    No      Cycles floating point units are idle (FP_NONE_RET)
PAPI_TLB_DM     No      Data translation lookaside buffer misses
    (DC_L1_DTLB_MISS_AND_L2_DTLB_MISS)
PAPI_TLB_IM     No      Instruction translation lookaside buffer misses
    (IC_L1ITLB_MISS_AND_L2ITLB_MISS)
```

Simple Low Level Example in C

```
#include "papi.h”
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int retval, EventSet = PAPI_NULL;
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();
/*Stop counters and store results in values */
retval = PAPI_read(EventSet,values);
do_more_work();
/*Stop counters and store results in values */
retval = PAPI_stop(EventSet,values);
```

Simple Low Level Example in Fortran

```
#include “fpapi.h”
integer evset, status, retval
integer*8 values(2)
retval = PAPI_VER_CURRENT
evset = PAPI_NULL
call papif_library_init(retval)
call papif_create_eventset(evset, status)
call papif_add_event(evset, PAPI_TOT_CYC, status)
call papif_add_event(evset, PAPI_FP_INS, status)
call papif_start(evset, status)
C
call do_work()
C
call papif_read(evset, values, status)
C
call do_more_work()
C
call papif_stop(evset, values, status)
```

PAPI High-level Example

```
long long values[NUM_EVENTS];
unsigned int Events[NUM_EVENTS]=
  {PAPI_TOT_INS,PAPI_TOT_CYC};
 /* Start the counters */
 PAPI_start_counters((int*)Events,NUM_EVENTS);
 /* What we are monitoring… */
 do_work();
 /* Stop the counters and store the results in
  values */
 retval = PAPI_stop_counters(values,NUM_EVENTS);
```

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

- Instrumentation Mechanism
  - Source
  - Binary (DPCL/DynInst)
  - Library interposition
- Data Management
  - Performance Database
  - User (Flat file)
- Data Visualization
  - Run Time
  - Post Mortem
  - Serial/Parallel Display
  - ASCII

papiex: PAPI Execute

- A simple tool that generates performance measurements for the entire run of a code.
- Requires no recompilation.
  - Uses library preloading and function replacement.
- Monitors all subprocesses.
- Output goes to stderr.
- Currently only handles the main thread.

Build the papiex Tool

- Cd papi/tools/trapper
- Make install PREFIX=<dir> (default /usr)
- Cd ../papiex
- Make install PREFIX=<dir> PAPI_PREFIX=<dir>

Papiex Usage

```
> papiex -h
Usage: ./papiex [-lihvtm] [-e event]... -- <cmd> <cmd options>
 -l             Print the available preset events.
 -i             Print information about the host machine.
 -h             Print this message.
 -v             Print version information.
 -t             Enable monitoring of multiple threads.
 -m             Enable multiplexing of hardware counters.
 -e event       Monitor this hardware event.

** All counts reflect user mode code only. **
```

Using papiex

```
torc13(109)> ./papiex -i
Vendor string and code    : AuthenticAMD (2)
Model string and code     : AMD K8 Revision C (15)
CPU Megahertz             : 1794.932983
Total # of CPU's          : 1
Number Hardware Counters : 4
Max Multiplex Counters    : 32

torc13(110)> ./papiex -l
Preset events.

Preset   Derived Description     Native Name
PAPI_L1_DCM     No      Level 1 data cache misses (DC_MISS)
PAPI_L1_ICM     No      Level 1 instruction cache misses (IC_MISS)
PAPI_L2_DCM     No      Level 2 data cache misses
(BU_L2_FILL_MISS_DC)
PAPI_L2_ICM     No      Level 2 instruction cache misses
(BU_L2_FILL_MISS_IC)
PAPI_L1_TCM     Yes     Level 1 cache misses (DC_MISS, IC_MISS)
PAPI_L2_TCM     Yes     Level 2 cache misses (BU_L2_FILL_MISS_IC,
BU_L2_FILL_MISS_DC)
PAPI_FPU_IDL    No      Cycles floating point units are idle
(FP_NONE_RET)
```

Using papiex on SWIM

```
> f77 swim.F

> f77 -O3 swim.F

> papiex -e PAPI_L1_DCA -e PAPI_L1_DCM -e PAPI_FP_INS -e
   PAPI_TOT_CYC ./a.out

[swim output deleted]

Executable:     /a/nala/flannel/homes/mucci/dynaprof/tests/a.out
Process ID:     16217
Hostname:       torc13
Start:          Fri Apr  2 21:45:54 2004
Finish:         Fri Apr  2 21:46:02 2004
PAPI_L1_DCA:    7767266125
PAPI_L1_DCM:    56169610
PAPI_FP_INS:    3354924950
PAPI_TOT_CYC:   14117355986
Real usecs:     7967477
Proc usecs:     7927087
Real cycles:    14293650686
Proc cycles:    14221192099
```

Example papiex Performance Data

- From that data: (parens are -O3 version)
  - Delivered MFLOP/S: 423 (795)
    - PAPI_FP_INS / Process Virtual Usec
  - L1 Hit Rate: 99.3% (96.4%)
    - (1.0 - (PAPI_L1_DCM / PAPI_L1_DCA)) * 100.0
  - Computation Intensity: 2.3 Load-Stores/Flop (.56)
    - PAPI_L1_DCA / PAPI_FP_INS
  - Note that PAPI_FP_INS does not include SSE, SSE2 or 3dNow! Instructions.

TAU Performance System

Tuning and Analysis Utilities (11+ year project effort)

Performance system framework for scalable parallel and distributed high-performance computing

Targets a general complex system computation model

- nodes / contexts / threads
- Multi-level: system / software / parallelism
- Measurement and analysis abstraction

Integrated toolkit for performance instrumentation, measurement, analysis, and visualization

Portable performance profiling and tracing facility

- Open software approach with technology integration

University of Oregon , Forschungszentrum Jülich, LANL

TAU Performance Measurement

- TAU supports profiling and tracing measurement
- Robust timing and hardware performance support using PAPI
- Support for online performance monitoring
  - Profile and trace performance data export to file system
  - Selective exporting
- Extension of TAU measurement for multiple counters
  - Creation of user-defined TAU counters
  - Access to system-level metrics
- Support for callpath measurement
- Integration with system-level performance data
  - Linux MAGNET/MUSE (Wu Feng, LANL)

TAU Measurement

Performance information

- Performance events
- High-resolution timer library (real-time / virtual clocks)
- General software counter library (user-defined events)

Hardware performance counters

PAPI (Performance API) (UTK, Ptools Consortium)

- consistent, portable API

Organization

- Node, context, thread levels

Profile groups for collective events (runtime selective)

- Performance data mapping between software levels

TAU Measurement Options

Parallel profiling

- Function-level, block-level, statement-level
- Supports user-defined events
- TAU parallel profile data stored during execution
- Hardware counts values
- Support for multiple counters
- Support for callgraph and callpath profiling

Tracing

- All profile-level events
- Inter-process communication events
- Trace merging and format conversion

TAU Instrumentation Approach

- *Support for standard program events*
  - *Routines*
  - *Classes and templates*
  - *Statement-level blocks*
- *Support for user-defined events*
  - *Begin/End events ("user-defined timers")*
  - *Atomic events (e.g., size of memory allocated/freed)*
  - *Selection of event statistics*
- *Support definition of "semantic" entities for mapping*
- *Support for event groups*
- *Instrumentation optimization*

TAU Instrumentation

Flexible instrumentation mechanisms at multiple levels

Source code

- *manual*
- *automatic*
  - *C, C++, F77/90/95 (Program Database Toolkit (PDT))*
  - *OpenMP (directive rewriting (Opari), POMP spec)*

Object code

- *pre-instrumented libraries (e.g., MPI using PMPI)*
- *statically-linked and dynamically-linked*

Executable code

- *dynamic instrumentation (pre-execution) (DynInstAPI)*
- *virtual machine instrumentation (e.g., Java using JVMPI)*

Multi-Level Instrumentation

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

Program Database Toolkit (PDT)

Program code analysis framework

- develop source-based tools

High-level interface to source code information

Integrated toolkit for source code parsing, database creation, and database query

- Commercial grade front-end parsers
- Portable IL analyzer, database format, and access API
- Open software approach for tool development

Multiple source languages

Implement automatic performance instrumentation tools

tau_instrumentor

TAU's MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
  - Provides name shifted interface
    - MPI_Send = PMPI_Send
    - Weak bindings
- Interpose TAU's MPI wrapper library between MPI and TAU
  - -lmpi replaced by –lTauMpi –lpmpi –lmpi
- No change to the source code! Just re-link the application to generate performance data

Steps in using TAU

- Install TAU
- Instrument application
  - TAU Profiling API
- Typically modify application makefile
  - include TAU's stub makefile, modify variables
- Set environment variables
  - directory where profiles/traces are to be stored
- Execute application

```
% mpirun –np <procs> a.out;
```

- Analyze performance data
  - paraprof, vampir, pprof, paraver …

TAU Analysis

Parallel profile analysis

Pprof

- parallel profiler with text-based display

ParaProf

- Graphical, scalable, parallel profile analysis and display

Trace analysis and visualization

- Trace merging and clock adjustment (if necessary)
- Trace format conversion (ALOG, SDDF, VTF, Paraver)
- Trace visualization using *Vampir* (Pallas/Intel)

TAU Performance System Status

Computing platforms (selected)

- IBM SP / pSeries, SGI Origin 2K/3K, Cray T3E / SV-1 / X1, HP (Compaq) SC (Tru64), Sun, Hitachi SR8000, NEC SX-5/6, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS X), Windows

Programming languages

- C, C++, Fortran 77/90/95, HPF, Java, OpenMP, Python

Thread libraries

- pthreads, SGI sproc, Java,Windows, OpenMP

Compilers (selected)

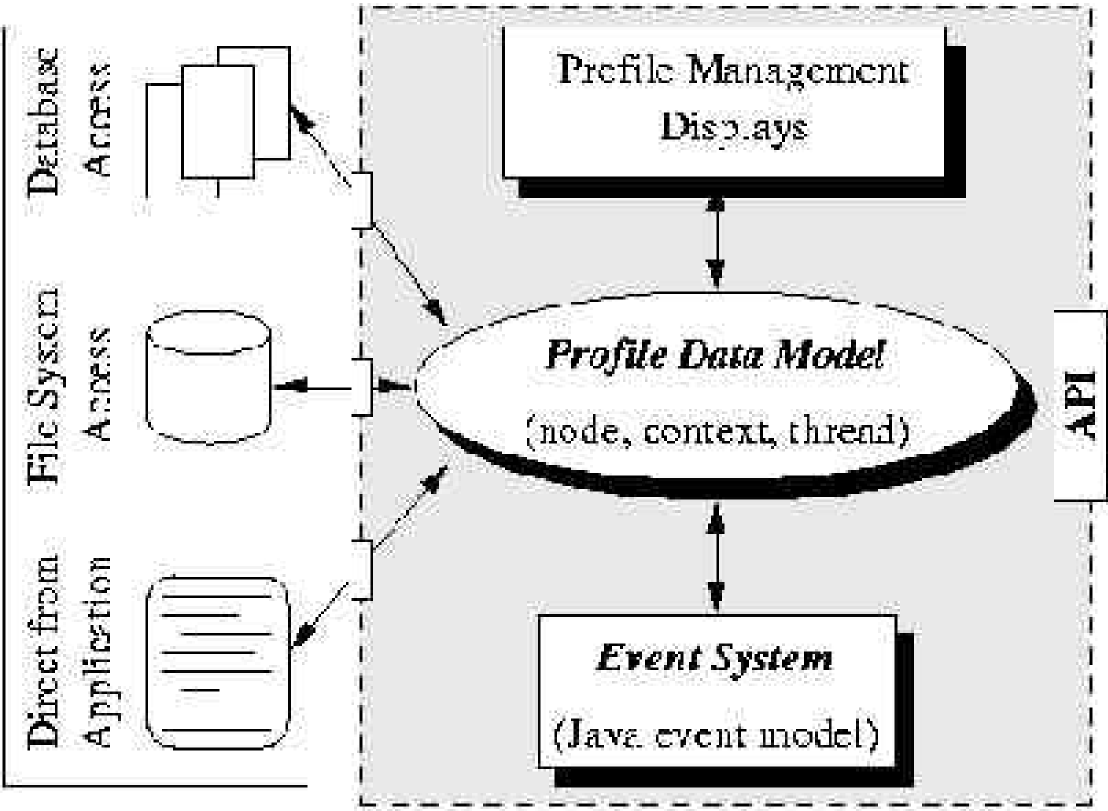
- Intel KAI (KCC, KAP/Pro), PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM (xlc, xlf), Compaq, NEC, Intel

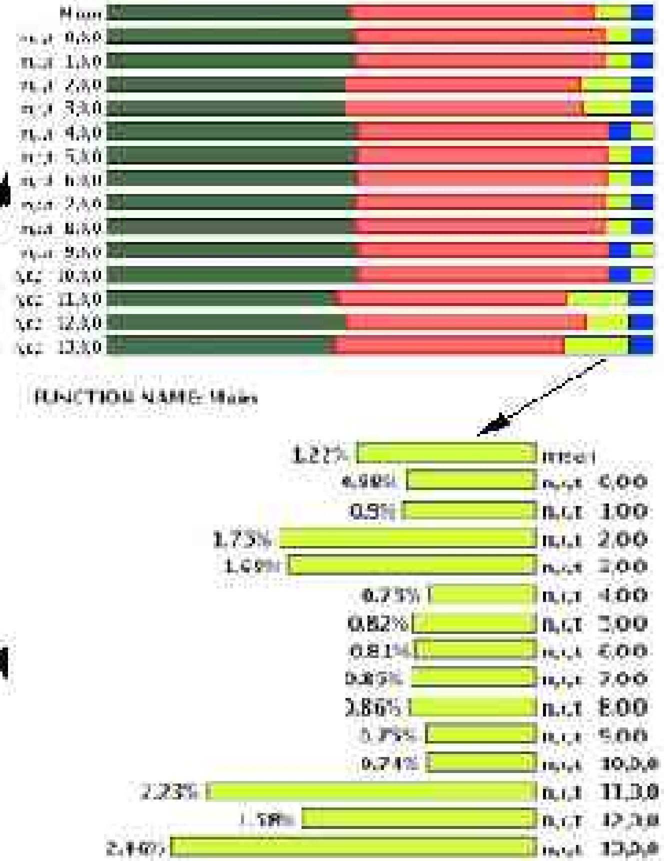
ParaProf Framework Architecture

Portable, extensible, and scalable tool for profile analysis

Try to offer "best of breed" capabilities to analysts

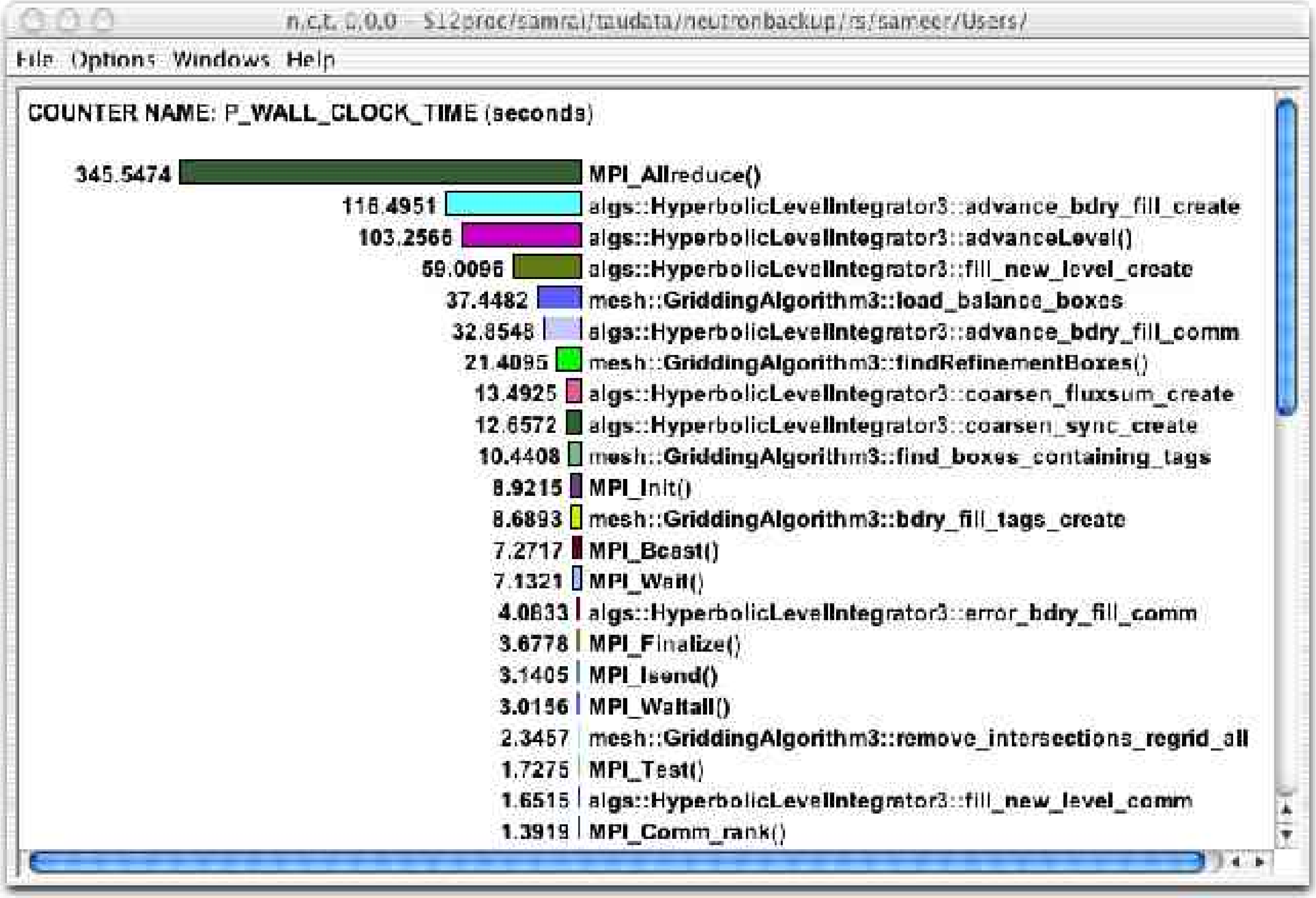
Build as profile analysis framework for extensibility





http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

Node / Context / Thread Profile Window



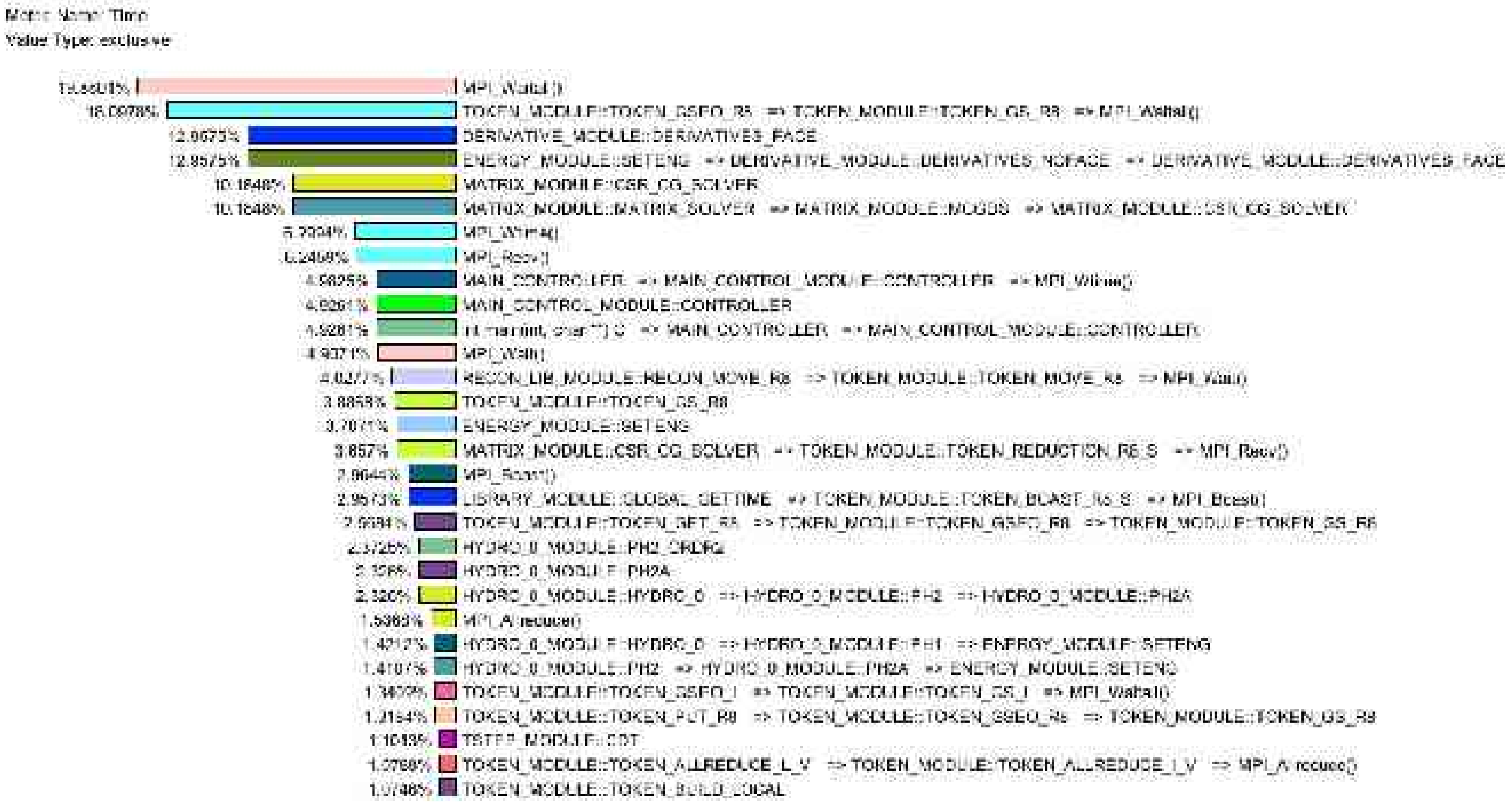
```
COUNTER NAME: P_WALL_CLOCK_TIME (seconds)

345.5474  MPI_Allreduce()
116.4951  algs::HyperbolicLevelIntegrator3::advance_bdry_fill_create
103.2566  algs::HyperbolicLevelIntegrator3::advanceLevel()
 59.0096  algs::HyperbolicLevelIntegrator3::fill_new_level_create
 37.4482  mesh::GriddingAlgorithm3::load_balance_boxes
 32.8548  algs::HyperbolicLevelIntegrator3::advance_bdry_fill_comm
 21.4095  mesh::GriddingAlgorithm3::findRefinementBoxes()
 13.4925  algs::HyperbolicLevelIntegrator3::coarsen_fluxsum_create
 12.6572  algs::HyperbolicLevelIntegrator3::coarsen_sync_create
 10.4408  mesh::GriddingAlgorithm3::find_boxes_containing_tags
  8.9215  MPI_Init()
  8.6893  mesh::GriddingAlgorithm3::bdry_fill_tags_create
  7.2717  MPI_Bcast()
  7.1321  MPI_Wait()
  4.0833  algs::HyperbolicLevelIntegrator3::error_bdry_fill_comm
  3.6778  MPI_Finalize()
  3.1405  MPI_Isend()
  3.0156  MPI_Waitall()
  2.3457  mesh::GriddingAlgorithm3::remove_intersections_regrid_all
  1.7275  MPI_Test()
  1.6515  algs::HyperbolicLevelIntegrator3::fill_new_level_comm
  1.3919  MPI_Comm_rank()
```

http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

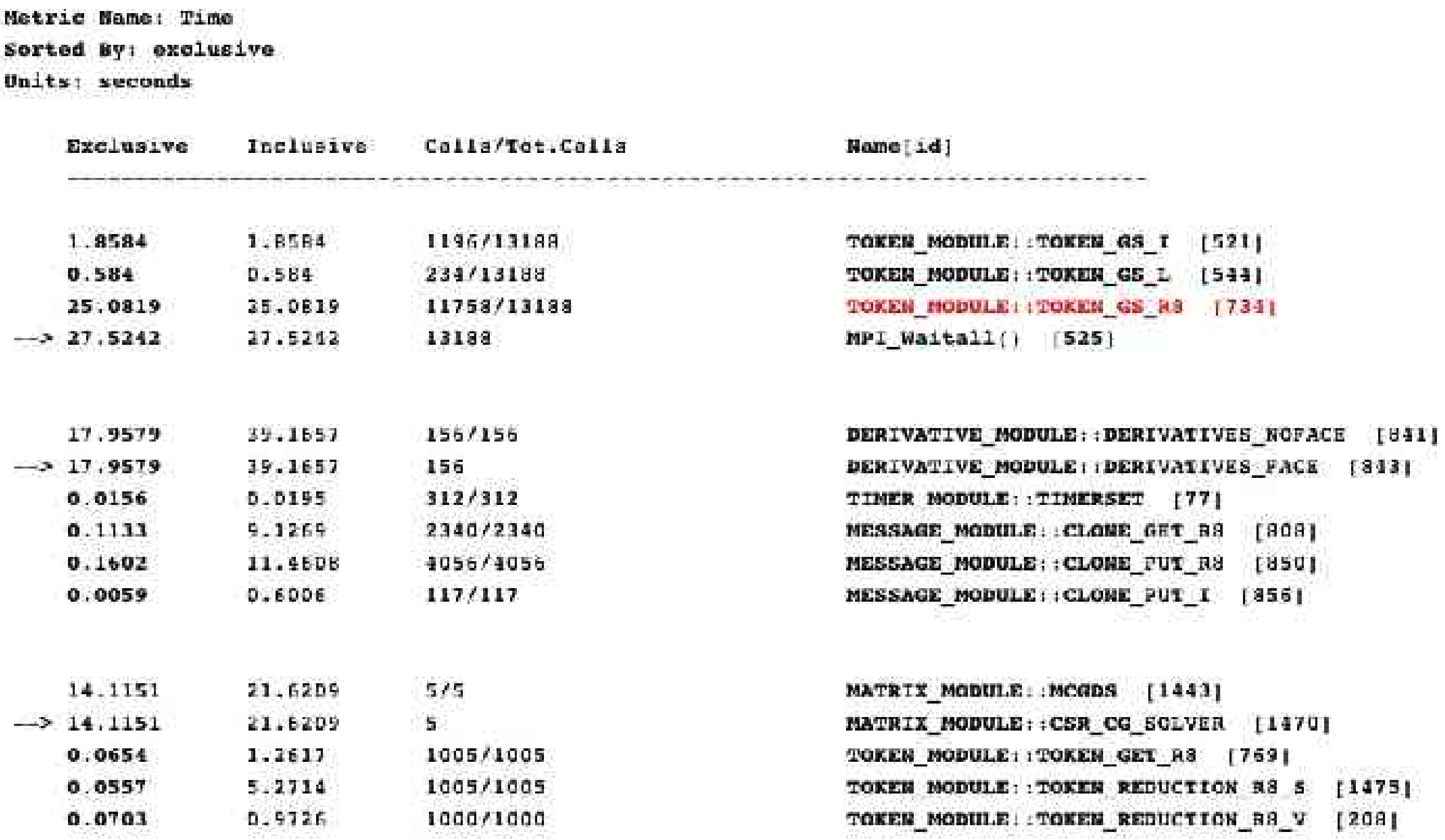
Callpath Implementation in TAU

- TAU maintains a performance event (routine) callstack
- Profiled routine (child) looks in callstack for parent
  - Previous profiled performance event is the parent
  - A *callpath profile structure* created first time parent calls
  - TAU records parent in a *callgraph map* for child
  - String representing k-level callpath used as its key
    - “**a( )=>b( )=>c()**” : name for time spent in “c” when called by “b” when “b” is called by “a”
- Map returns pointer to callpath profile structure
  - k-level callpath is profiled using this profiling data
  - Set environment variable TAU_CALLPATH_DEPTH to depth
- Build upon TAU’s performance mapping technology
- Measurement is independent of instrumentation
- Use – PROFILECALLPATH to configure TAU

Example Callpath Data



Gprof Style Callpath View in Paraprof



```
Metric Name: Time
Sorted By: exclusive
Units: seconds

    Exclusive     Inclusive     Calls/Tot.Calls              Name[id]
    ----------------------------------------------------------------------------------

    1.8584        1.8584        1196/13188                   TOKEN_MODULE::TOKEN_GS_I  [521]
    0.584         0.584         234/13188                    TOKEN_MODULE::TOKEN_GS_L  [544]
    25.0819       25.0819       11758/13188                  TOKEN_MODULE::TOKEN_GS_R8  [734]
--> 27.5242       27.5242       13188                        MPI_Waitall()  [525]


    17.9579       39.1657       156/156                      DERIVATIVE_MODULE::DERIVATIVES_NOFACE  [841]
--> 17.9579       39.1657       156                          DERIVATIVE_MODULE::DERIVATIVES_FACE  [843]
    0.0156        0.0195        312/312                      TIMER_MODULE::TIMERSET  [77]
    0.1131        9.1269        2340/2340                    MESSAGE_MODULE::CLONE_GET_R8  [808]
    0.1602        11.4608       4056/4056                    MESSAGE_MODULE::CLONE_PUT_R8  [850]
    0.0059        0.6006        117/117                      MESSAGE_MODULE::CLONE_PUT_I  [856]


    14.1151       21.6209       5/5                          MATRIX_MODULE::MCGDS  [1443]
--> 14.1151       21.6209       5                            MATRIX_MODULE::CSR_CG_SOLVER  [1470]
    0.0654        1.2617        1005/1005                    TOKEN_MODULE::TOKEN_GET_R8  [769]
    0.0557        5.2714        1005/1005                    TOKEN_MODULE::TOKEN_REDUCTION_R8_S  [1475]
    0.0703        0.9726        1000/1000                    TOKEN_MODULE::TOKEN_REDUCTION_R8_V  [208]
```

Building TAU

- Download TAU from
  - http://www.cs.uoregon.edu/research/paracomp/tau
- TAU requires the configuration and building of multiple measurement libraries for different environments.
- First build PDT. (only once)
  - `./configure -arch=x86_64`
  - `make install`
- Here, we build with PDT, PAPI and callpath profiling.
  - `./configure -arch=x86_64 -papi=<directory> -pdt=<directory> -MULTIPLECOUNTERS -PROFILECALLPATH -PROFILE -PAPIWALLCLOCK -PAPIVIRTUAL`

TAU Measurement System Configuration

configure [OPTIONS]

– {-c++=<CC>, -cc=<cc>}	Specify C++ and C compilers
– {-pthread, -sproc}	Use pthread or SGI sproc threads
– -openmp	Use OpenMP threads
– -jdk=<dir>	Specify Java instrumentation (JDK)
– -opari=<dir>	Specify location of Opari OpenMP tool
– -papi=<dir>	Specify location of PAPI
– -pdt=<dir>	Specify location of PDT
– -dyninst=<dir>	Specify location of DynInst Package
– -mpi[inc/lib]=<dir>	Specify MPI library instrumentation
– -python[inc/lib]=<dir>	Specify Python instrumentation
– -epilog=<dir>	Specify location of EPILOG

TAU Measurement System Configuration

configure [OPTIONS]

-TRACE	Generate binary TAU traces
-PROFILE (default)	Generate profiles (summary)
-PROFILECALLPATH	Generate call path profiles
-PROFILESTATS	Generate std. dev. statistics
-MULTIPLECOUNTERS	Use hardware counters + time
-COMPENSATE	Compensate timer overhead
-CPUTIME	Use usertime+system time
-PAPIWALLCLOCK	Use PAPI's wallclock time
-PAPIVIRTUAL	Use PAPI's process virtual time
-SGITIMERS	Use fast IRIX timers
-LINUXTIMERS	Use fast x86 Linux timers

Description of Optional Packages

PAPI – Measures hardware performance data e.g., floating point instructions, L1 data cache misses etc.

DyninstAPI – Helps instrument an application binary at runtime or rewrites the binary

EPILOG – Trace library. Epilog traces can be analyzed by EXPERT [FZJ], an automated bottleneck detection tool.

Opari – Tool that instruments OpenMP programs

Vampir – Commercial trace visualization tool [Pallas]

Paraver – Trace visualization tool [CEPBA]

Compiling: TAU Makefiles

- Include TAU Stub Makefile (<arch>/lib) in the user's Makefile.
- Variables:

TAU_CXX	Specify the C++ compiler used by TAU
TAU_CC, TAU_F90	Specify the C, F90 compilers
TAU_DEFS	Defines used by TAU. Add to CFLAGS
TAU_LDFLAGS	Linker options. Add to LDFLAGS
TAU_INCLUDE	Header files include path. Add to CFLAGS
TAU_LIBS	Statically linked TAU library. Add to LIBS
TAU_SHLIBS	Dynamically linked TAU library
TAU_MPI_LIBS	TAU's MPI wrapper library for C/C++
TAU_MPI_FLIBS	TAU's MPI wrapper library for F90
TAU_FORTRANLIBS	Must be linked in with C++ linker for F90
TAU_CXXLIBS	Must be linked in with F90 linker
TAU_INCLUDE_MEMORY	Use TAU's malloc/free wrapper lib
TAU_DISABLE	TAU's dummy F90 stub library

- Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs (TAU_DISABLE for f90).

Including TAU's Stub Makefile

```
include /usr/local/tau/x86_64/lib/Makefile.tau-papiwallclock-
multiplecounters-papivirtual-papi-pdt
CC   = $(TAU_CC)
LIBS = $(TAU_LIBS) $(TAU_CXXLIBS)
LD_FLAGS = $(TAU_LDFLAGS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
	$(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.f.o:
	$(F90) $(FFLAGS) -c $< -o $@
```

Setup: Running Applications

```
% set path=($path <taudir>/<arch>/bin)
% set path=($path $PET_HOME/PTOOLS/tau-2.13.5/src/rs6000/bin)
% setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH\:<taudir>/<arch>/lib

For PAPI (multiplecounters):
% setenv COUNTER1 PAPI_FP_INS      (PAPI’s Floating point ins)
% setenv COUNTER2 PAPI_TOT_CYC     (PAPI’s Total cycles)
% setenv COUNTER3 P_VIRTUAL_TIME     (PAPI’s virtual time)
% setenv COUNTER4 LINUX_TIMERS      (Wallclock time)
% mpirun -np <n> <application>
% llsubmit job.sh
% paraprof   (for performance analysis)
```

Using PDT: tau_instrumentor

```
% tau_instrumentor
Usage : tau_instrumentor <pdbfile> <sourcefile> [-o <outputfile>] [-noinline] [-g
groupname] [-i headerfile] [-c|-c++|-fortran] [-f <instr_req_file> ]
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.

BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

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

TAU Makefile for PDT (C++)

```
include /usr/local/tau/x86_64/lib/Makefile.tau-pdt
CXX = $(TAU_CXX)
CC  = $(TAU_CC)
PDTPARSE = $(PDTDIR)/$(PDTARCHDIR)/bin/cxxparse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
CFLAGS = $(TAU_DEFS) $(TAU_INCLUDE)
LIBS = $(TAU_LIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
      $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.cpp.o:
      $(PDTPARSE) $<
      $(TAUINSTR) $*.pdb $< -o $*.inst.cpp -f select.dat
      $(CC) $(CFLAGS) -c $*.inst.cpp -o $@
```

TAU Makefile for PDT (F90)

```
include /usr/local/tau/x86_64/lib/Makefile.tau-pdt
F90 = $(TAU_F90)
CC  = $(TAU_CC)
PDTPARSE = $(PDTDIR)/$(PDTARCHDIR)/bin/f95parse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
LIBS = $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS = f1.o f2.o f3.o
TARGET = a.out
PDB = merged.pdb
TARGET:$(PDB) $(OBJS)
      $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
$(PDB): $(OBJS:.o=.f)
      $(PDTF95PARSE) $(OBJS:.o=.f) -o$(PDB) -R free
# This expands to f95parse *.f -omerged.pdb -R free
.f.o:
      $(TAU_INSTR) $(PDB) $< -o $*.inst.f -f sel.dat;\
      $(FCOMPILE) $*.inst.f -o $@;
```

TAU Manual Instrumentation API for C/C++

- Initialization and runtime configuration

```
TAU_PROFILE_INIT(argc, argv);
TAU_PROFILE_SET_NODE(myNode);
TAU_PROFILE_SET_CONTEXT(myContext);
TAU_PROFILE_EXIT(message);
TAU_REGISTER_THREAD();
```

- Function and class methods for C++ only:

```
TAU_PROFILE(name, type, group);
```

- Template

```
TAU_TYPE_STRING(variable, type);
TAU_PROFILE(name, type, group);
CT(variable);
```

- User-defined timing

```
TAU_PROFILE_TIMER(timer, name, type, group);
TAU_PROFILE_START(timer);
TAU_PROFILE_STOP(timer);
```

TAU Measurement API (continued)

- User-defined events

  TAU_REGISTER_EVENT(variable, event_name);
  TAU_EVENT(variable, value);
  TAU_PROFILE_STMT(statement);

- Heap Memory Tracking:

  TAU_TRACK_MEMORY();

  TAU_SET_INTERRUPT_INTERVAL(seconds);

  TAU_DISABLE_TRACKING_MEMORY();

  TAU_ENABLE_TRACKING_MEMORY();

- Reporting

  TAU_REPORT_STATISTICS();

  TAU_REPORT_THREAD_STATISTICS();

Manual Instrumentation – C++ Example

```
#include <TAU.h>
int main(int argc, char **argv)
{
  TAU_PROFILE(“int main(int, char **)”, “ ”, TAU_DEFAULT);
  TAU_PROFILE_INIT(argc, argv);
  TAU_PROFILE_SET_NODE(0); /* for sequential programs */
  foo();
  return 0;
}

int foo(void)
{
  TAU_PROFILE(“int foo(void)”, “ ”, TAU_DEFAULT); // measures entire foo()
  TAU_PROFILE_TIMER(t, “foo(): for loop”, “[23:45 file.cpp]”, TAU_USER);
  TAU_PROFILE_START(t);
  for(int i = 0; i < N ; i++){
    work(i);
  }
  TAU_PROFILE_STOP(t);
  // other statements in foo …
}
```

Manual Instrumentation – C Example

```
#include <TAU.h>
int main(int argc, char **argv)
{
  TAU_PROFILE_TIMER(tmain, “int main(int, char **)”, “ ”, TAU_DEFAULT);
  TAU_PROFILE_INIT(argc, argv);
  TAU_PROFILE_SET_NODE(0); /* for sequential programs */
  TAU_PROFILE_START(tmain);
  foo();
  …
  TAU_PROFILE_STOP(tmain);
  return 0;
}
int foo(void)
{
  TAU_PROFILE_TIMER(t, “foo()”, “ ”, TAU_USER);
  TAU_PROFILE_START(t);
  for(int i = 0; i < N ; i++){
    work(i);
  }
  TAU_PROFILE_STOP(t);
}
```

Manual Instrumentation – F90 Example

```
cc34567 Cubes program – comment line
      PROGRAM SUM_OF_CUBES
       integer profiler(2)
       save profiler
      INTEGER :: H, T, U
        call TAU_PROFILE_INIT()
        call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
        call TAU_PROFILE_START(profiler)
        call TAU_PROFILE_SET_NODE(0)
      ! This program prints all 3-digit numbers that
      ! equal the sum of the cubes of their digits.
      DO H = 1, 9
        DO T = 0, 9
          DO U = 0, 9
          IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
             PRINT "(3I1)", H, T, U
          ENDIF
          END DO
        END DO
      END DO
      call TAU_PROFILE_STOP(profiler)
      END PROGRAM SUM_OF_CUBES
```

Instrumentation of OpenMP Constructs

OpenMP **P**ragma **A**nd **R**egion **I**nstrumentor

- Source-to-Source translator to insert POMP calls around OpenMP constructs and API functions

Done: Supports

Fortran77 and Fortran90, OpenMP 2.0

C and C++, OpenMP 1.0

POMP Extensions

- EPILOG and TAU POMP implementations
- Preserves source code information (`#line line file`)

Using Opari with TAU

Step I: Configure KOJAK/opari [Download from http://www.fz-juelich.de/zam/kojak/]

```
% cd kojak-0.99; cp mf/Makefile.defs.ibm Makefile.defs;
  edit Makefile
% make
```

Builds opari

Step II: Configure TAU with Opari (used here with PDT)

```
% configure -opari=/usr/contrib/TAU/kojak-0.99/opari
   -pdt=/usr/contrib/TAU/pdtoolkit-3.1
% make clean; make install
```

!$OMP PARALLEL DO Instrumentation

```
call pomp_parallel_fork(d)
!$OMP PARALLEL other-clauses...
      call pomp_parallel_begin(d)
      call pomp_do_enter(d)
      !$OMP DO schedule-clauses, ordered-clauses,
               lastprivate-clauses
            do loop
      !$OMP END DO NOWAIT
      call pomp_barrier_enter(d)
      !$OMP BARRIER
      call pomp_barrier_exit(d)
      call pomp_do_exit(d)
      call pomp_parallel_end(d)
!$OMP END PARALLEL DO
call pomp_parallel_join(d)
```

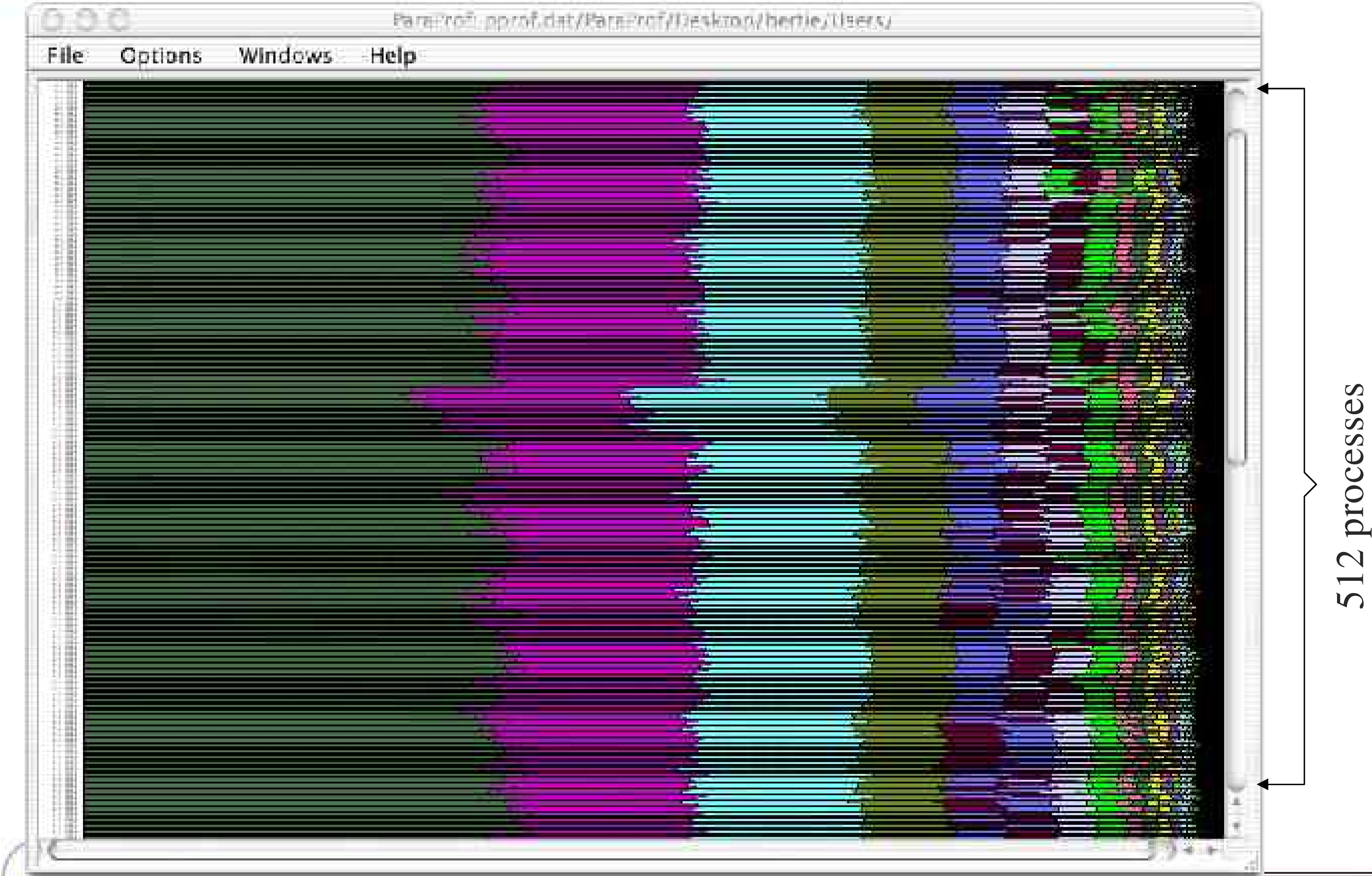
OPARI: Makefile Template (C/C++)

```
OMPCC  = ...           # insert C OpenMP compiler here
OMPCXX = ...           # insert C++ OpenMP compiler here

.c.o:
       opari $<
       $(OMPCC) $(CFLAGS) -c $*.mod.c
.cc.o:
       opari $<
       $(OMPCXX) $(CXXFLAGS) -c $*.mod.cc

opari.init:
       rm -rf opari.rc

opari.tab.o:
       opari -table opari.tab.c
       $(CC) -c opari.tab.c

myprog: opari.init myfile*.o ... opari.tab.o
       $(OMPCC) -o myprog myfile*.o opari.tab.o -lpomp

myfile1.o: myfile1.c myheader.h
myfile2.o: ...
```

OPARI: Makefile Template (Fortran)

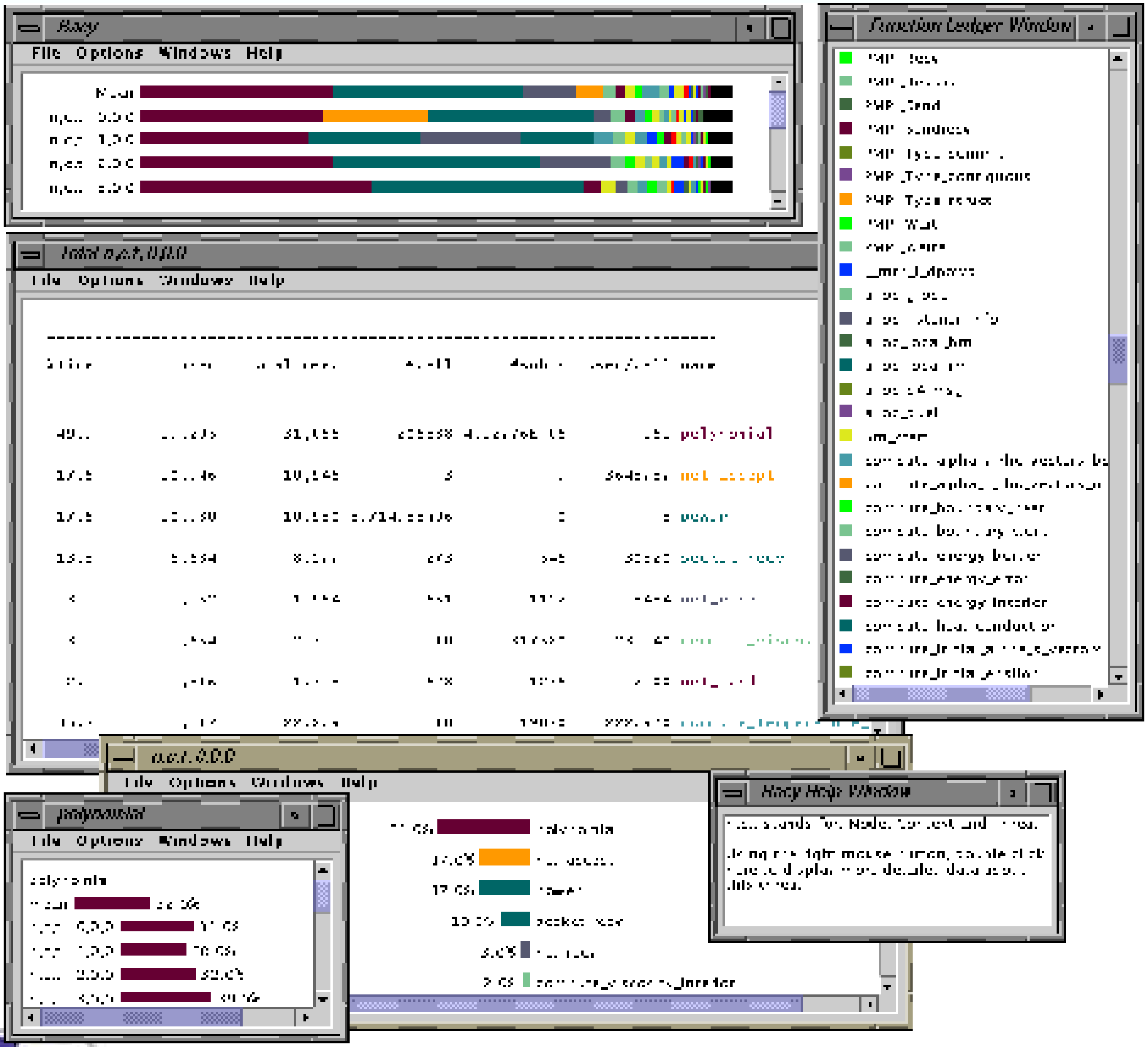
```
OMPF77 = ...          # insert f77 OpenMP compiler here
OMPF90 = ...          # insert f90 OpenMP compiler here

.f.o:
       opari $<
       $(OMPF77) $(CFLAGS) -c $*.mod.F
.f90.o:
       opari $<
       $(OMPF90) $(CXXFLAGS) -c $*.mod.F90

opari.init:
       rm -rf opari.rc

opari.tab.o:
       opari -table opari.tab.c
       $(CC) -c opari.tab.c

myprog: opari.init myfile*.o ... opari.tab.o
       $(OMPF90) -o myprog myfile*.o opari.tab.o -lpomp

myfile1.o: myfile1.f90
myfile2.o: ...
```

Full Profile Window (Exclusive Time)

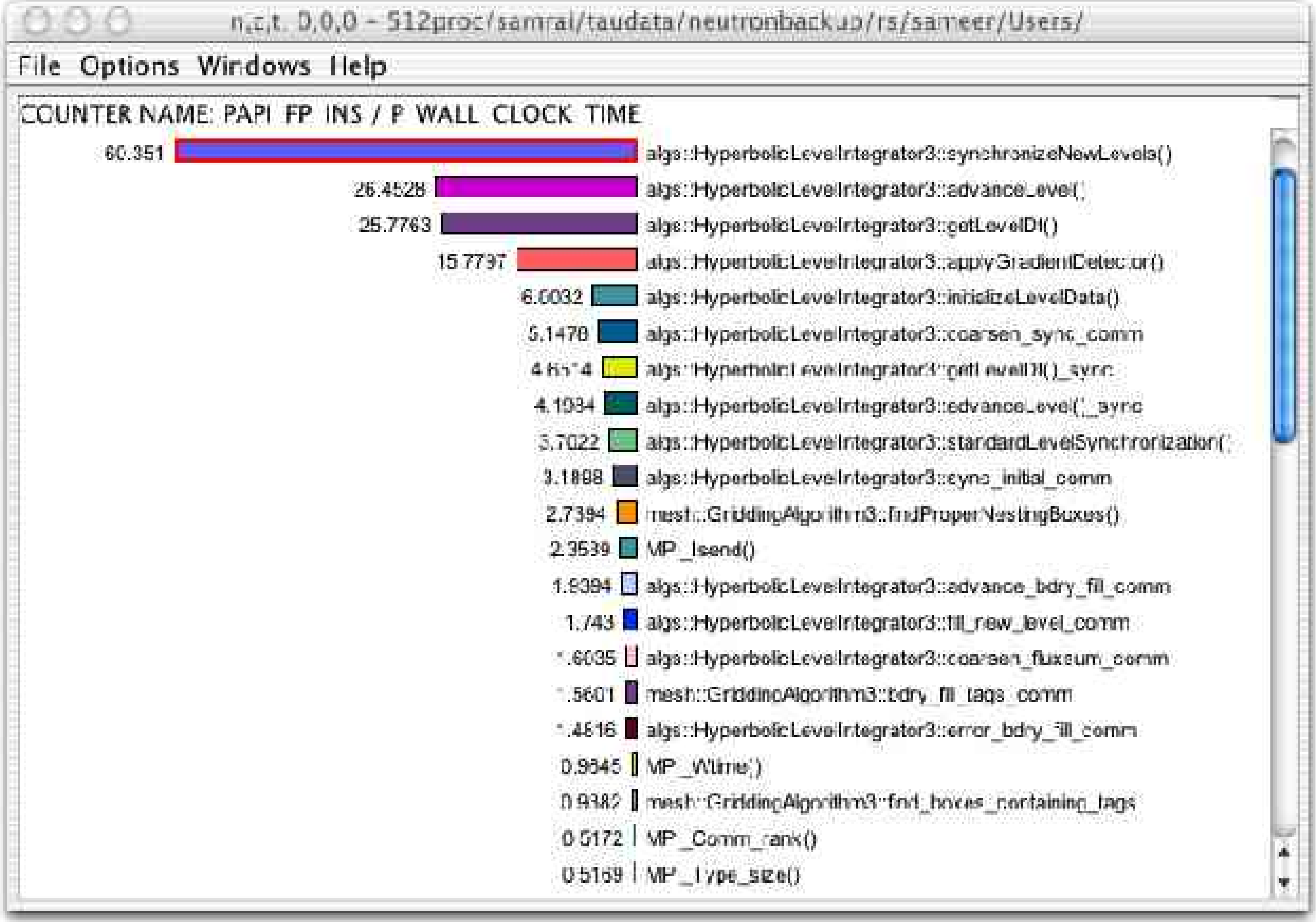


http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

SIMPLE Hydrodynamics Benchmark

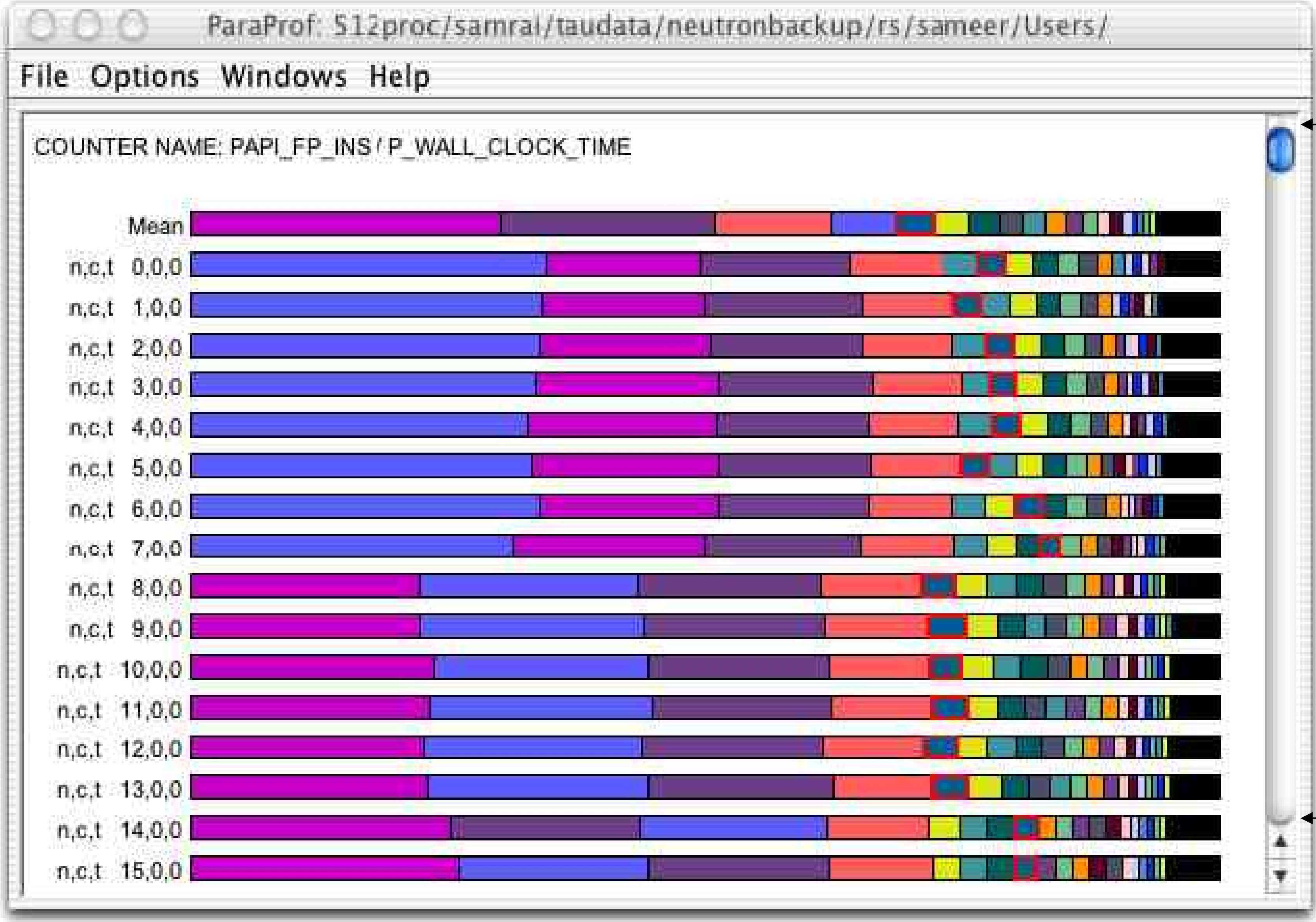


Derived Metrics



http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

Profile Window for Derived Metrics



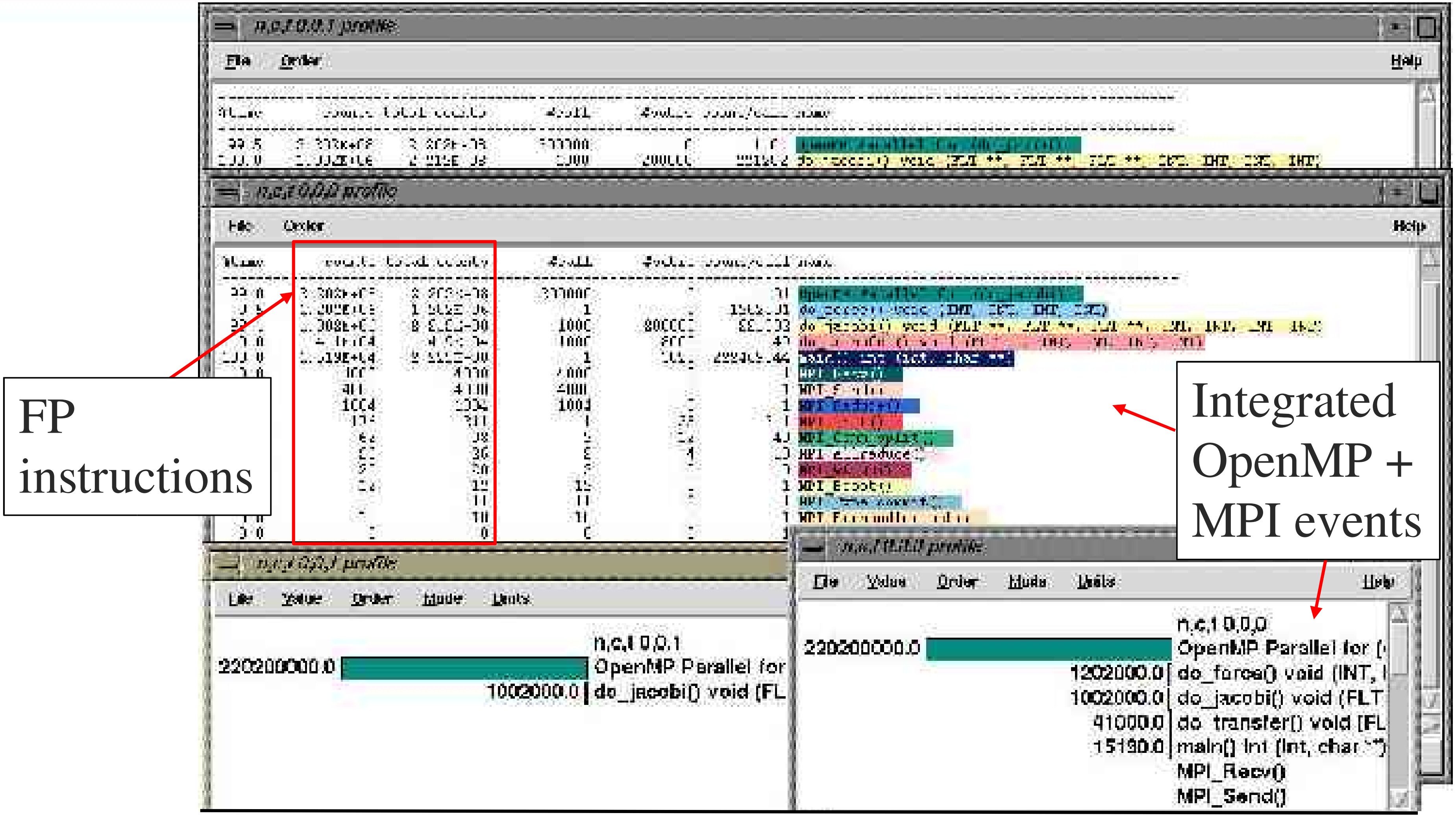
ParaProf: 512proc/samrai/taudata/neutronbackup/rs/sameer/Users/

COUNTER NAME: PAPI_FP_INS / P_WALL_CLOCK_TIME

512 processes

http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

OpenMP + MPI Ocean Modeling (HW Profile)



n,c,t 0,0,1
220200000.0 OpenMP Parallel for
1002000.0 do_jacobi() void (FL

n,c,t 1,0,0,0
220200000.0 OpenMP Parallel for (
1202000.0 do_force() void (INT,
1002000.0 do_jacobi() void (FLT
41000.0 do_transfer() void (FL
15190.0 main() int (int, char **)
MPI_Recv()
MPI_Send()

FP instructions

Integrated OpenMP + MPI events

```
% configure -papi=../packages/papi -openmp -c++=pgCC -cc=pgcc
  -mpiinc=../packages/mpich/include -mpilib=../packages/mpich/lib
```

http://www.cs.utk.edu/~mucci/latest/pubs/cw2004.pdf

TAU Support Acknowledgements

Department of Energy (DOE)

- Office of Science contracts
- University of Utah DOE ASCI Level 1 sub-contract
- DOE ASCI Level 3 (LANL, LLNL)

NSF National Young Investigator (NYI) award

Research Centre Juelich

- John von Neumann Institute for Computing
- Dr. Bernd Mohr

Los Alamos National Laboratory

Tools that use PAPI

- TAU (Sameer Shende, U Oregon)
  http://www.cs.uoregon.edu/research/paracomp/tau/
- SvPablo (Celso Mendes, UIUC)
  http://www-pablo.cs.uiuc.edu/Project/SVPablo/
- HPCToolkit (J. Mellor-Crummey, Rice U)
  http://hipersoft.cs.rice.edu/hpctoolkit/
- psrun (Rick Kufrin, NCSA, UIUC)
  http://www.ncsa.uiuc.edu/~rkufrin/perfsuite/psrun/
- Titanium (Dan Bonachea, UC Berkeley)
  http://www.cs.berkeley.edu/Research/Projects/titanium/

Tools that use PAPI

- SCALEA (Thomas Fahringer, U Innsbruck)
  http://www.par.univie.ac.at/project/scalea/
- KOJAK (Bernd Mohr, FZ Juelich; U Tenn)
  http://www.fz-juelich.de/zam/kojak
- Cone (Felix Wolf, U Tenn)
  http://icl.cs.utk.edu/kojak/cone
- HPMtoolkit (Luiz Derose, IBM)
  http://www.alphaworks.ibm.com/tech/hpmtoolkit
- CUBE (Felix Wolf, U Tenn)
  http://icl.cs.utk.edu/kojak/cube

Tools that use PAPI

- ParaVer (J. Labarta, CEPBA)

  http://www.cepba.upc.es/paraver

- VAMPIR (Pallas)

  http://www.pallas.com/e/products/vampir/index.htm

- DynaProf (P. Mucci, U Tenn)

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

Questions?

This talk:

- http://www.cs.utk.edu/~mucci/latest/mucci_talks.html

The PAPI Homepage:

- http://icl.cs.utk.edu/papi