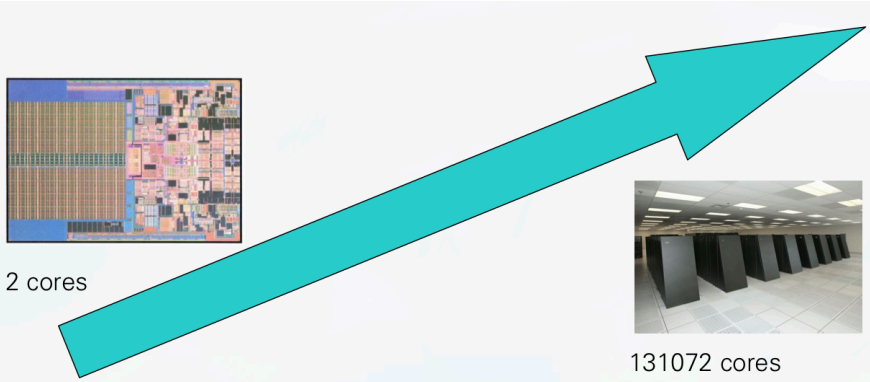
ECE-451/ECE-566 - Introduction to Parallel and Distributed Programming

Performance Evaluation

Department of Electrical & Computer Engineering
Rutgers University

Performance Evaluation Tools

Motivation

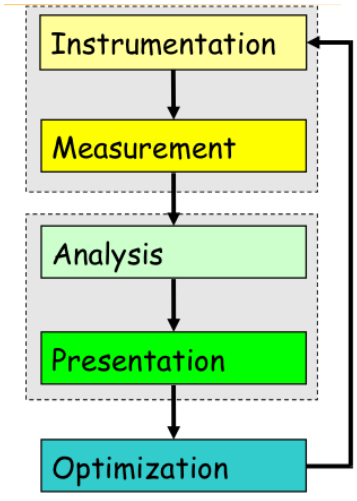


Scalability!!

Performance Optimization

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Tune Performance

This is an "art"



Steps of Performance Evaluation

° Collect basic routine-level timing profile to determine where most time is being spent

° Collect routine-level hardware counter data to determine types of performance problems

° Collect callpath profiles to determine sequence of events causing performance problems

° Conduct finer-grained profiling and/or tracing to pinpoint performance bottlenecks

- Loop-level profiling with hardware counters
- Tracing of communication operations

Parallel Performance Properties

° Parallel code performance is influenced by both sequential and parallel factors?

° Sequential factors

- Computation and memory use
- Input / output

° Parallel factors

- Thread / process interactions
- Communication and synchronization

Performance Analysis Questions

- How does performance vary with different compilers?
- Is poor performance correlated with certain OS features?
- Has a recent change caused unanticipated performance?
- How does performance vary with MPI variants?
- Why is one application version faster than another?
- What is the reason for the observed scaling behavior?
- Did two runs exhibit similar performance?
- How are performance data related to application events?
- Which machines will run my code the fastest and why?
- Which benchmarks predict my code performance best?

Performance Considerations and Strategies

- The most important goal of performance tuning is to reduce a program's wall clock execution time.
  - Reducing resource usage in other areas, such as memory or disk requirements, may also be a tuning goal.
- Performance tuning is an iterative process used to optimize the efficiency of a program. It usually involves finding your program's *hot spots* and eliminating the *bottlenecks* in them.
  - *Hot Spot*: An area of code within the program that uses a disproportionately high amount of processor time
  - *Bottleneck*: An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays
- Performance tuning usually involves profiling - using software tools to measure a program's run-time characteristics and resource utilization
- Use profiling tools and techniques to learn which areas of your code offer the greatest potential performance increase BEFORE you start the tuning process.
  - Then, target the most time consuming and frequently executed portions of a program for optimization.

An example

° Consider optimizing your underlying algorithm: an extremely fine-tuned O(N * N) sorting algorithm may perform significantly worse than a untuned O(N log N) algorithm

° For data dependent computations, benchmark based on a variety of realistic (both size and values) input data sets. Maintain consistent input data during the fine-tuning process.

° Take advantage of compiler and preprocessor optimizations when possible

° Finally, know when to stop - there are diminishing returns in successive optimizations. Consider a program with the following breakdown of execution time percentages for the associated parts of the program:

Procedure	%CPU Time
main()	13%
procedure1()	17%
procedure2()	20%
procedure3()	50%

° A 20% increase in the performance of procedure3() results in a 10% performance increase overall.

° A 20% increase in the performance of main() results in only a 2.6% performance increase overall.

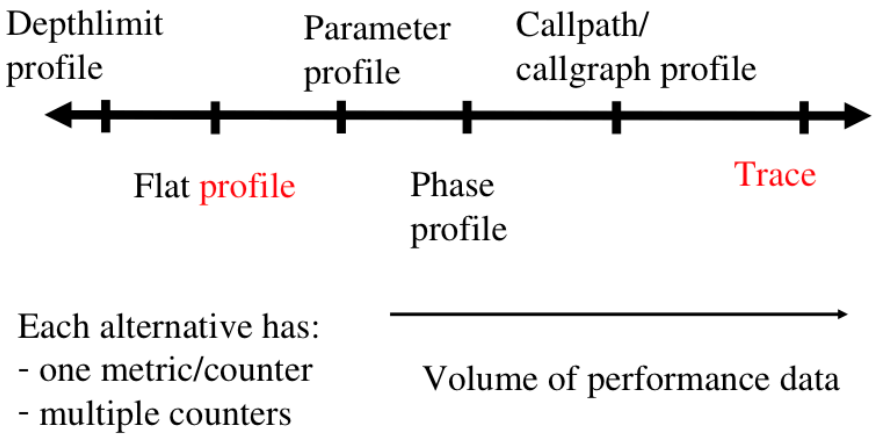
Critical Issues

° Accuracy
- Timing and counting accuracy depends on resolution
- Any performance measurement generates overhead
  - Execution on performance measurement code
- Measurement overhead can lead to intrusion
- Intrusion can cause perturbation
  - Alters program behavior

° Granularity
- How many measurements are made
- How much overhead per measurement

° Tradeoff (general wisdom)
- Accuracy is inversely correlated with granularity

Performance Evaluation Alternatives



Profiling / Tracing Comparison

° Profiling
- \+ Finite, bounded performance data size
- \+ Applicable to both direct and indirect methods
- = Loses time dimension (not entirely)
- \- Lacks ability to fully describe process interaction

- Tracing
- \+ Temporal and spatial dimension to performance data
- \+ Capture parallel dynamics and process interaction
- = Some inconsistencies with indirect methods
- \- Unbounded performance data size (large)
- \- Complex event buffering and clock synchronization

Timers (example: IBM SP2 System)

Timer	Usage	Wallclock / CPU Time	Resolution	Languages	Portable?
time	shell / script	both	1/100th second	any	yes
timex	shell / script	both	1/100th second	any	yes
gettimeofday	subroutine	wallclock	microsecond	C/C++	yes
read_real_time	subroutine	wallclock	nanosecond	C/C++	no
rtc	subroutine	wallclock	microsecond	Fortran	no
irtc	subroutine	wallclock	nanosecond	Fortran	no
dtime_	subroutine	CPU	1/100th second	Fortran	no
etime_	subroutine	CPU	1/100th second	Fortran	no
mclock	subroutine	CPU	1/100th second	Fortran	no
timef	subroutine	wallclock	millisecond	Fortran	no
MPI_Wtime	subroutine	wallclock	microsecond	C/C++, Fortran	yes
AIX Trace Facility	shell / script / subroutine	wallclock	microsecond	any	no

Time

- The time command returns the total execution time of your program.
- The format of the output is different for the Korn shell and the C shell. The basic information is : Real time: the total wall clock (start to finish) time your program took to load, execute, and exit.
- User time: the total amount of CPU time your program took to execute.
- System time: the amount of CPU time spent on operating system calls in executing your program.
- The system and user times are defined differently across different computer architectures.
- Example csh time output:

```
   1      2       3      4      5          6      7   8
1.150u 0.020s 0:01.76 66.4% 15+3981k 24+10io 0pf+0w
```

- Explanation:
- 1.15 seconds of user CPU time
- 0.02 seconds of system (kernel) time used on behalf of user
- 1.76 seconds real time (wall clock time)
- 66.4% total CPU time (user+system) during execution as a percentage of elapsed time.
- 15 Kbytes of shared memory usage and 3981 Kbytes of unshared data space
- 24 block input operations and 10 block output operations
- no page faults
- no swaps

Time (cont.)

° Example ksh time output:

real 0m2.58s

user 0m1.14s

sys 0m0.03s

Explanation:

0 minutes, 2.58 seconds of wall clock time

0 minutes, 1.14 seconds of user CPU time

0 minutes, 0.03 seconds of system CPU time

gettimeofday()

° The gettimeofday() routine is part of the Standard C Library (libc.a) on most Unix systems. It returns the time in seconds and microseconds since midnight, January 1, 1970.

° Can be inserted anywhere within a C program and used to determine the start and end times of code fragments.

° Fortran programs must perform an interlanguage call in order to use the gettimeofday function.

° Timer resolution is hardware dependent. Microsecond resolution on the IBM SP systems.

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

struct timeval start_time, end_time;

main()
{
   int total_usecs;

   gettimeofday(&start_time, (struct timeval*)0);

   /*   do some work */

   gettimeofday(&end_time, (struct timeval*)0);

   total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
                 (end_time.tv_usec-start_time.tv_usec);
   printf("Total time was %d uSec.\n", total_usecs);
}
```

read_real_time() & time_base_to_time()

- Both of these routines are part of the Standard C Library (libc.a) on IBM AIX systems.
- Designed to be used for making high-resolution measurement of elapsed time, using the processor real time clock or time base registers.
- Nanosecond resolution on the IBM SP.
- The time_base_to_time() routine is used to guarantee correct time units across different IBM RS/6000 architectures.

```
#include <stdio.h>
#include <sys/time.h>

int main(void)
{
   timebasestruct_t start, finish;
   int secs, n_secs;

   read_real_time(&start, TIMEBASE_SZ);

   /* do some work */

   read_real_time(&finish, TIMEBASE_SZ);

   /* Make sure both values are in seconds and nanoseconds */
   time_base_to_time(&start, TIMEBASE_SZ);
   time_base_to_time(&finish, TIMEBASE_SZ);

   /* Subtract the starting time from the ending time */
   secs = finish.tb_high - start.tb_high;
   n_secs = finish.tb_low - start.tb_low;

  /* Fix carry from low-order to high-order during the measurement */
   if (n_secs < 0)  {
      secs--;
      n_secs += 1000000000;
      }

   (void) printf("Sample time was %d seconds %d nanoseconds\n",
                 secs, n_secs);

   exit(0);
}
```

AIX Trace Facility

- AIX's built in Trace Facility provides the ability to timestamp many different system events.
- It can be used at both the command line level and subroutine level.
- Using the Trace Facility to collect timing statistics for an event is a three step process:
  - Determine the code numbers for the events you wish to trace by consulting the /usr/include/sys/trchkid.h file.
  - Generate a tracefile from either the command line or from within your program.
  - Produce a "readable" trace report from your tracefile after it has been completed.
- Example:
- Get event tags from /usr/include/sys/trchkid.h for desired events. Thread create=465, thread terminate=467, wait lock=46d.
- Remove any existing tracefile (can cause problems): rm trcfile
- Execute program, creating a tracefile with specified events: trace -a -o trcfile -j 465,467,46d; dotprod; trcstop
- Produce a readable trace report: trcrpt -o trace.report -O pid=on trcfile
- View report using your favorite editor.

AIX Trace Facility

```
Fri Jul 19 00:02:05 2002
System: AIX frost067 Node: 5
Machine: 006008594C00
Internet Address: 86092F43 134.9.47.67
The system contains 16 cpus, of which 16 were traced.
Buffering: Kernel Heap
This is from a 32-bit kernel.
Tracing only these hooks, 465,467,46d

trace -a -o trcfile -j 465,467,46d


ID  PID      I    ELAPSED_SEC     DELTA_MSEC   APPL    SYSCALL KERNEL  INTERRUPT

001 70640         0.000000000       0.000000                  TRACE ON channel
0

                                                              Fri Jul 19 00:02:
05 2002
465 -1            0.000482511       0.482511          thread_create:   pid=7825
0 tid=226787 priority=60 policy=0
46D 85616         0.000495484       0.012973          wait_on_lock:   pid=85616
 tid=238427 lockaddr=5819F0
465 -1            0.027897260      27.401776          thread_create:   pid=7731
8 tid=146769 priority=61 policy=0
467 77318         0.028225717       0.328457          thread_terminate:   pid=7
7318 tid=146769
465 -1            0.028235827       0.010110          thread_create:   pid=7731
8 tid=206215 priority=61 policy=0
46D -1            0.028251016       0.015189          wait_on_lock:   pid=77318
 tid=238429 lockaddr=5819F0
465 -1            0.028633968       0.382952          thread_create:   pid=7731
8 tid=151151 priority=61 policy=0
467 77318         0.028795141       0.161173          thread_terminate:   pid=7
7318 tid=206215
467 77318         0.028861449       0.066308          thread_terminate:   pid=7
7318 tid=151151
465 -1            0.028876390       0.014941          thread_create:   pid=7731
8 tid=233391 priority=61 policy=0
46D -1            0.028886636       0.010246          wait_on_lock:   pid=77318
 tid=238429 lockaddr=5819F0
467 77318         0.029303598       0.416962          thread_terminate:   pid=7
7318 tid=233391
002 -1            0.055766970      26.463372                  TRACE OFF channel
 0000 Fri Jul 19 00:02:05 2002
```

Profilers - prof

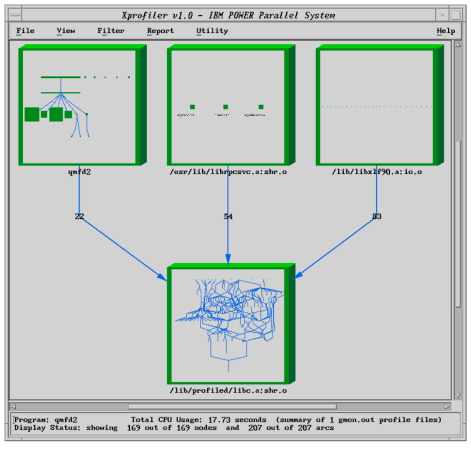
- The prof utility is included in most Unix systems. It is used to profile program execution at the procedure level.
- IBM has enabled the mpcc and mpxlf compiler commands to use prof for parallel programs.
- prof displays the following information: The name of each procedure - in descending order of processing activity
- The percentage of the program's CPU time used by each procedure.
- The execution time in seconds for all references by each procedure.
- The number of times the procedure was called.
- The average time in milliseconds for a call to each procedure.

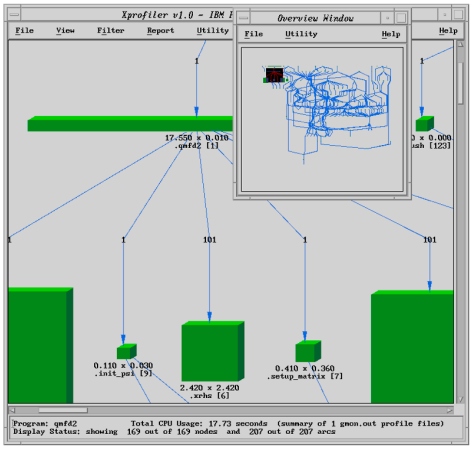
```
Name              %Time     Seconds     Cumsecs  #Calls   msec/call
.fft               51.8        0.59        0.59    1024       0.576
.main              40.4        0.46        1.05       1     460.
.bit_reverse        7.9        0.09        1.14    1024       0.088
.cos                0.0        0.00        1.14     256       0.00
.sin                0.0        0.00        1.14     256       0.00
.catopen            0.0        0.00        1.14       1       0.
.setlocale          0.0        0.00        1.14       1       0.
._doprnt            0.0        0.00        1.14       7       0.
._flsbuf            0.0        0.00        1.14      11       0.0
._xflsbuf           0.0        0.00        1.14       7       0.
._wrtchk            0.0        0.00        1.14       1       0.
._findbuf           0.0        0.00        1.14       1       0.
._xwrite            0.0        0.00        1.14       7       0.
.free               0.0        0.00        1.14       2       0.
.free_y             0.0        0.00        1.14       2       0.
.write              0.0        0.00        1.14       7       0.
.exit               0.0        0.00        1.14       1       0.
.memchr             0.0        0.00        1.14      19       0.0
.atoi               0.0        0.00        1.14       1       0.
.__nl_langinfo_std  0.0        0.00        1.14       4       0.
.gettimeofday       0.0        0.00        1.14       8       0.
.printf             0.0        0.00        1.14       7       0.
```

xprofiler

- xprofiler is an X Windows based profiling tool distributed with IBM's Parallel Environment software. It is based upon the gprof profiling utility, however it provides a graphical representation of profile data in addition to all of the usual gprof reports.
- Graphical information is organized into 3 main components:
  - Library Cluster Boxes
  - Function Boxes
  - Call Arcs
- xprofiler can profile at both the subroutine level, and at the source statement level. Profiling also includes any calls made to library functions.
- Time sampled profiling:
  - Like many profiling tools, xprofiler acquires its data by keeping track of the executing program's location whenever it is interrupted.
  - Interrupts occur at set intervals and are commonly called "ticks". Under AIX, a tick occurs every 1/100th second.
  - Those portions of the program which accumulate the most "ticks" reflect the areas where the program spends most of its time.
- Filters, zooming, and other options allow you to limit displays to only those portions of the call tree you are interested in analyzing.

Xprofile (cont.)





Hardware Performance Counters

- For many years, hardware engineers have designed in specialized registers to measure the performance of various aspects of a microprocessor
- HW performance counters provide application developers with valuable information about code sections that can be improved
- Hardware performance counters can provide insight into:
  - Whole program timing
  - Cache behaviors
  - Branch behaviors
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
  - Subroutine resolution
  - Process or thread attribution

What's PAPI?

- Middleware that provides a consistent and efficient programming interface for the performance counter hardware found in most major microprocessors.
- Started as a Parallel Tools Consortium project in 1998
  - Goal was to produce a specification for a **portable** interface to the hardware performance counters
- 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)

PAPI Hardware Events

° Preset Events
- Standard set of over 100 events for application performance tuning
- No standardization of the exact definition
- 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

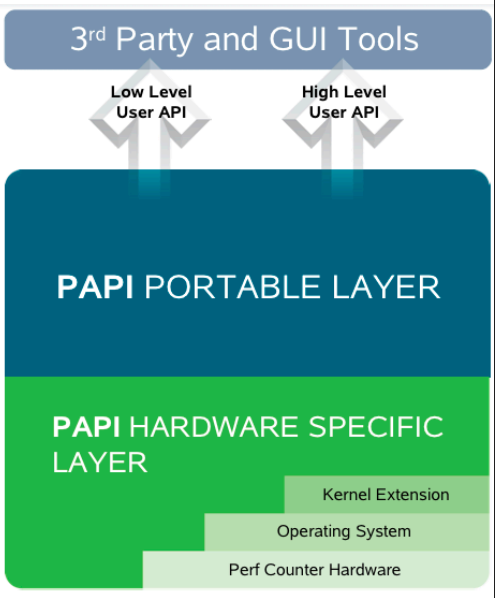
° Use ***papi_event_chooser*** utility to select a compatible set of events

PAPI Counter Interfaces

° PAPI provides 3 interfaces to the underlying counter hardware:

* A Low Level API manages hardware events (present 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.



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*)

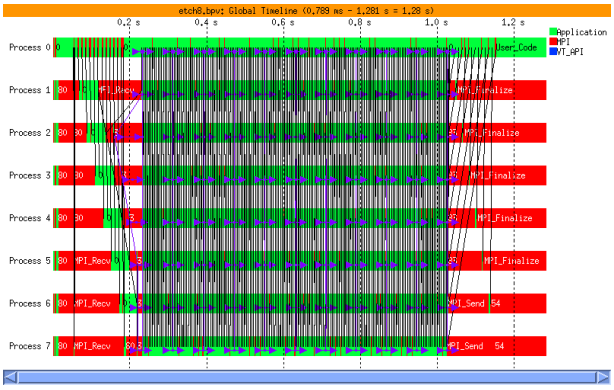
- stop counters and return current counts

Example: Low Level API

```
#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);
```

Tracing tools

- The parallel program is monitored while it is executed
- Monitoring produces performance data (TRACE) that is interpreted in order to reveal areas of poor performance.
- Tracing tools examples:
  - Vampir
  - Paraver



Vampir (Visualization and Analysis of MPI Resources)

- VAMPIR 2.0 is a post-mortem trace visualization tool from Pallas GmbH

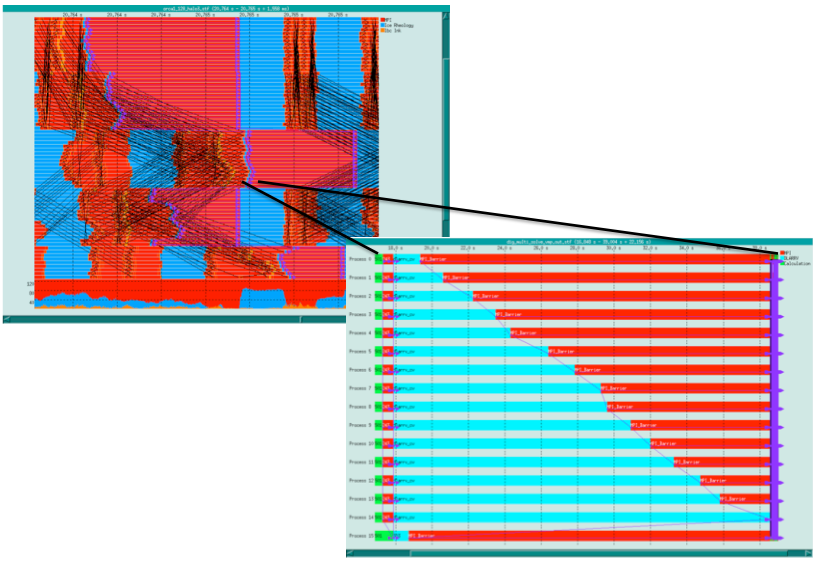
  http://www.pallas.com

  It uses the profile extensions to MPI and permits analysis of the message events where data is transmitted between processors during execution of a parallel program. It has a convenient user-interface and an excellent zooming and filtering. Global displays show all selected processes.

Vampir

- *Global Timeline*: detailed application execution over time axis
- *Activity Chart*: presents per-process profiling information
- *Summaric Chart*: aggregated profiling information
- *Communication Statistics*: message statistics for each process pair
- *Global Communication Statistics*: collective operations statistics
- *I/O Statistics*: MPI I/O operation statistics
- *Calling Tree*: global dynamic calling tree

Vampir



Paraver

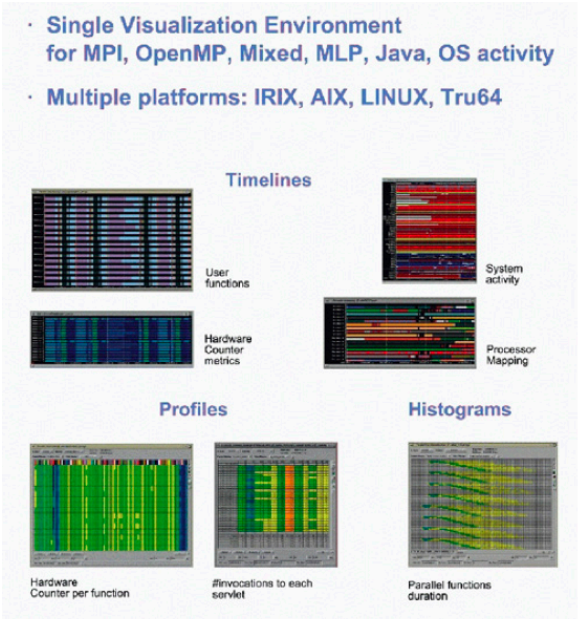
- Powerful **flexible** parallel program visualization tool based on an easy-to-use Motif GUI (graphical user interface)
  - Current version under Linux and Windows (wxWidgets). LGPL license
- Developed by :

  European Center for Parallelism of Barcelona (CEPBA) / Barcelona Supercomputing Center

  (Universitat Politecnica de Catalunya)

  http://www.bsc.es/

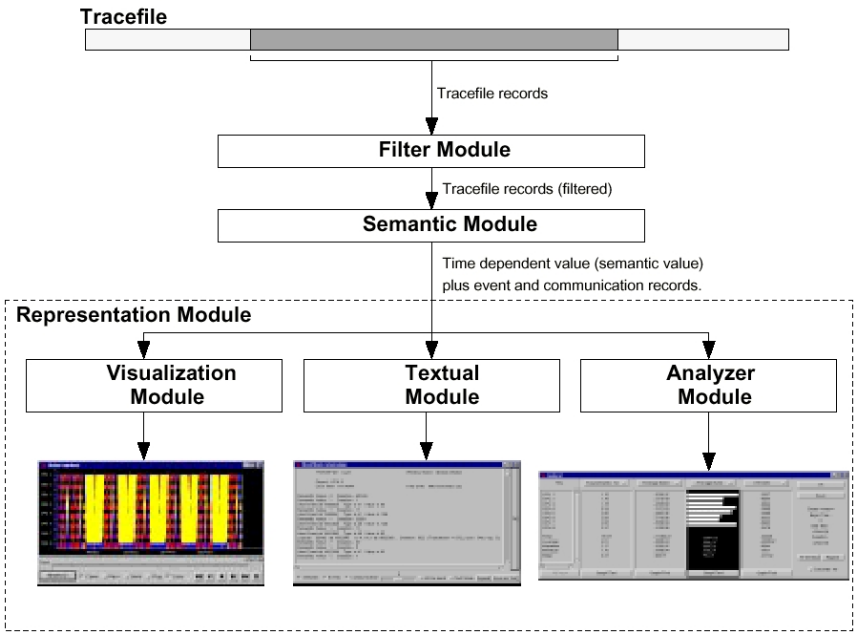
Paraver



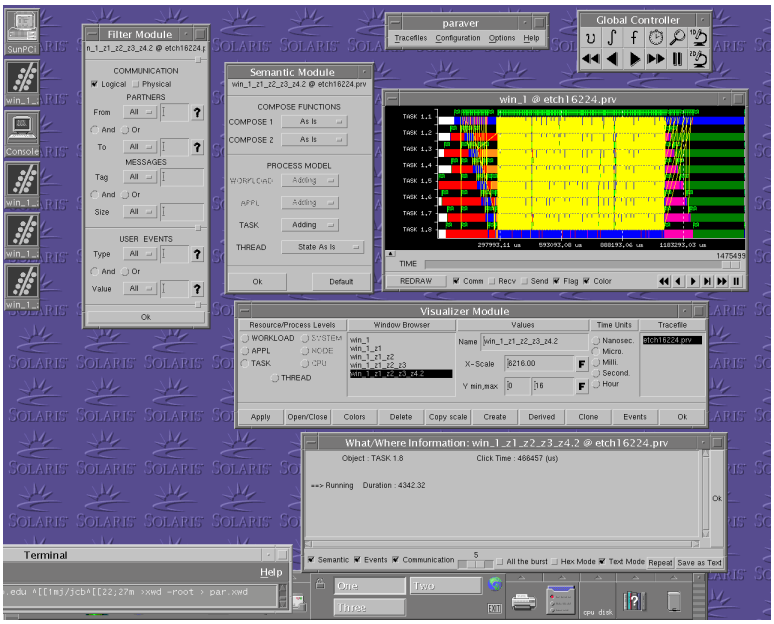
Paraver

- Paraver is designed to visualize and analyze:
  - Communication and load balance
  - Combining OpenMP and MPI
  - Hardware performance and counters
- Usage
  - Compile programs with special libraries
  - Run programs to produce trace files
  - View and analyze traces
  - Designed to help in program understanding and optimization

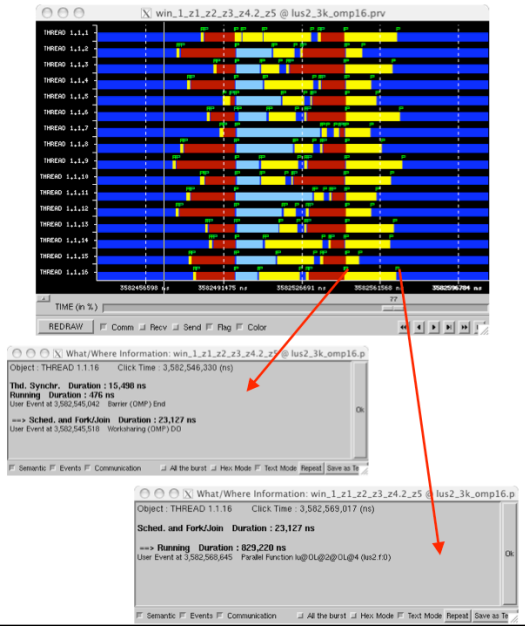
Paraver



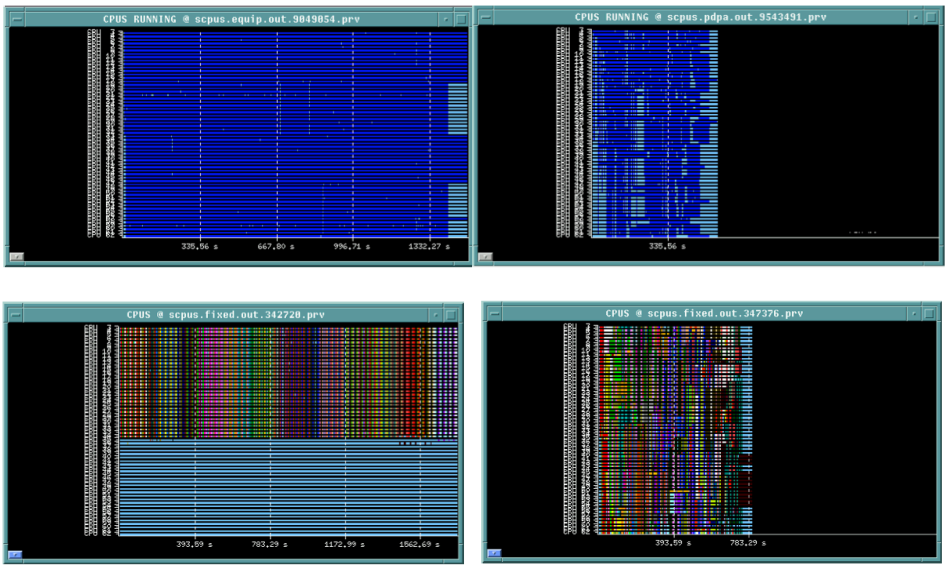
Paraver



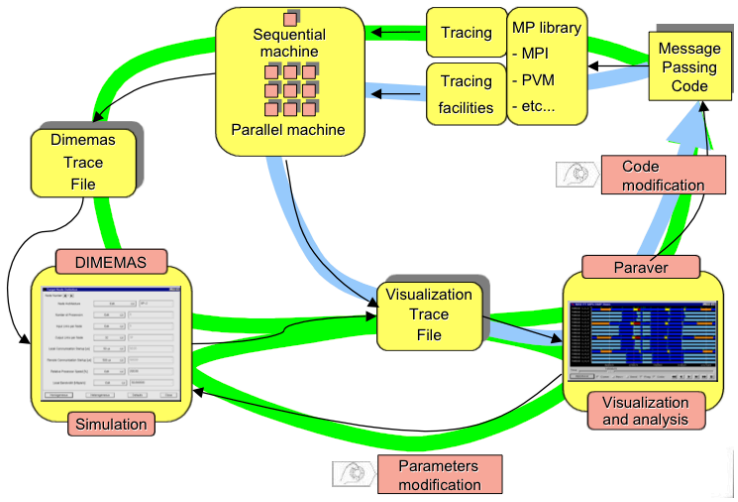
Paraver – Trace Example



Paraver – Example



Tuning Process with Paraver / Dimemas



Vtune

° **Presentation / Discussion in class…**

TAU Performance System Project

° Tuning and Analysis Utilities (15+ year project effort)

° Performance system framework for HPC systems
- Integrated, scalable, and flexible
- Target parallel programming paradigms

° Integrated toolkit for performance problem solving
- Instrumentation, measurement, analysis, and visualization
- Portable performance profiling and tracing facility
- Performance data management and data mining

° Partners
- LLNL, ANL, LANL
- Research Center Jülich, TU Desden

What is TAU?

- TAU is a performance evaluation tool
- It supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you when the events take place in each process along a timeline
- TAU uses a package called PDT for automatic instrumentation of source code
- Profiling and tracing can measure time as well as hardware performance counters from you CPU
- TAU can automatically instrument your source code (routines, loops, I/O, memory, phases, etc.)
- TAU runs on all HPC platforms and it is free (BSD style license)
- TAU has instrumentation, measurement and analysis tools
  - Paraprof is TAU's 3D profile browser
- To use TAU, you need to set a couple of environment variables and substitute the name of your compiler with a TAU shell script

TAU Performance System Interfaces

- PDT [U. Oregon, LANL, FZJ] for instrumentation of C++, C99, F95 source code
- PAPI [UTK] for accessing hardware performance counters data
- DyninstAPI [U. Maryland, U. Wisconsin] for runtime instrumentation
- KOJAK [FZJ, UTK]
  - Epilog trace generation library
  - CUBE callgraph visualizer
  - Opari OpenMP directive rewriting tool
- Vampir/VNG Trace Analyzer [TU Dresden]
- VTF3/OTF trace generation library [TU Dresden]
- Paraver trace visualizer [CEPBA/BSC]
- Jumpshot-4 trace visualizer [MPICH, ANL]
- JVMPI from JDK for Java program instrumentation [Sun]
- Paraprof profile browser/PerfDMF database supports"
  - TAU format
  - Gprof [GNU]
  - Etc.

Using TAU: A brief Introduction

- TAU supports several measurement options (profiling, tracing, profiling with hardware counters, etc.)
- Each measurement configuration of TAU corresponds to a unique stub makefile that is generated when you configure it
- To instrument source code using PDT
  - Choose an appropriate TAU stub makefile in <arch>/lib:

    % export TAU_MAKEFILE=/projects/tau/tau_latest/x86_64/lib/Makefile.tau-mpi-pdt
    % export TAU_OPTIONS='-optVerbose …' (see tau_compiler.sh -help)

    And use tau_f90.sh, tau_cxx.sh or tau_cc.sh as Fortran, C++ or C compilers:

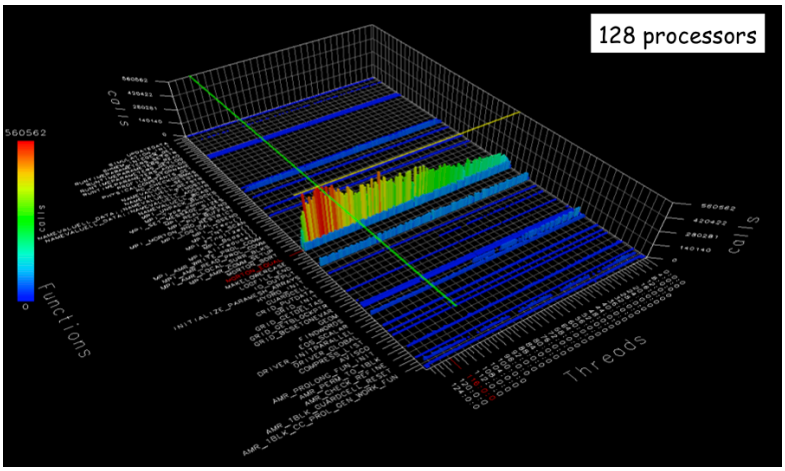
    % mpif90 foo.f90

    changes to

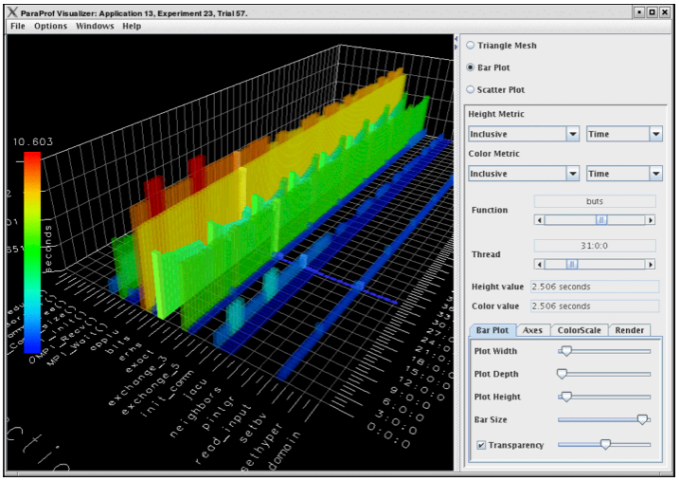
    % tau_f90.sh foo.f90
- Execute application and analyze performance data:
  - At runtime, if more than one metric is measured
    - export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_NATIVE_<native_event_name>
      - Use papi_native_avail, papi_avail, and papi_event_chooser to select these preset and native event names

    % pprof (for text based profile display)
    % paraprof (for GUI)

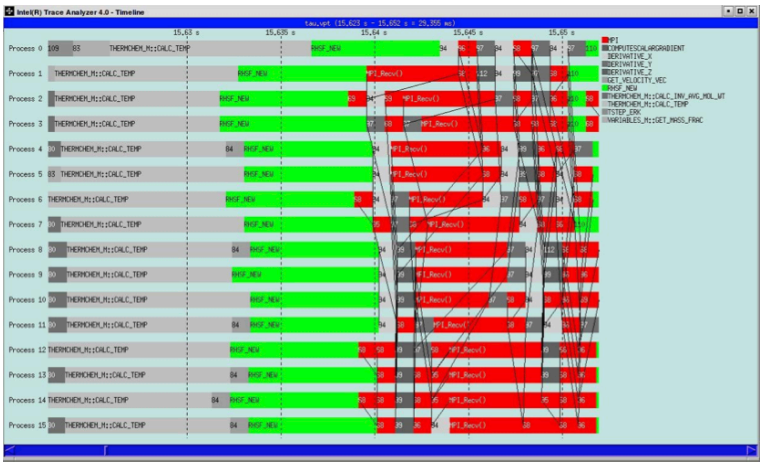
ParaProf – 3D Full Profile Bar Plot



ParaProf Bar Plot (Zoom in/out +/-)



Vampir – Trace Zoomed



Benchmarks

Micro-benchmarks

° Memory: design micro-benchmarks to measure
  - Access time to memory of one processor
  - Maximum bandwidth
  - Effect of contention in the interconnection network

° Communication: design different versions of micro-benchmarks to measure the bandwidth of MPI

Micro-benchmarks

° Memory

```
for(i=0; i<N; i+=B)
   s+=a[i]
```

BW… max
B > cache line size

```
for(i=0; i<N; i+=B)
    p=p->next;
```

Back-to-back latency

```
for(i=0; i<N; i+=B){
    p=p->next;
   for(j=1/j<J;j++)
       s+=1;
}
```

True unloaded latency

HPC benchmarks

° Mbench (Intel Labs)
- *A Memory Consistency Benchmark for OpenMP*
- Applications (7)
  - Body tracker, cloth, face, fb_est, fimi, nee and ode
- Kernels (8)
  - adjust_velocity, binomial_tree, blackScholes, dual_intersection, fft, pcg, svd and svm
- Primitives (9)
  - dense_mvm, dense_mvm_sym, gups, sparse_mvm, sparse_mvm_sym, sparse_mvm_trans

° Parsec (Princeton university)
- *The Princeton Application Repository for Shared-Memory Computers*
- Applications (10)
  - Blackscholes, bodytrack, facesim, ferret, fluidanimate, freqmine, raytrace, swaptions, vips, x264
- Kernels (3)
  - Canneal, dedup, streamcluster

° NPB (NASA)
- NASA Parallel Benchmarks (different types and classes/sizes)

° Others
- SSCA (Graph Analysis)
- LAMMPS (Molecular dynamics – MPI App)
- Sequoia Bechmark (MPI Apps)
  - AMG , IRS, LAMMPS, Pynamic and UMT

HPC Challenge benchmark (7 tests)

1. HPL - the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM - measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. STREAM - a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel.
4. PTRANS (parallel matrix transpose) - exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network.
5. RandomAccess - measures the rate of integer random updates of memory (GUPS).
6. FFT - measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).
7. Communication bandwidth and latency - a set of tests to measure latency and bandwidth of a number of simultaneous communication patterns; based on b_eff (effective bandwidth benchmark)