CS395T: Introduction to Scientific and Technical Computing

Performance and Profiling Issues in Computing

Dr. Kent Milfeld, TACC

Dr. Victor Eijkhout, Research Scientist, TACC
Dr. Karl W. Schulz, Research Associate, TACC

Outline

- Performance Optimizations
  - Single processor optimizations
- Profiling and Analysis
  - Using Timer routines
  - Using Profiling Tools
    - gprof, PAPI, TAU
- Using Debuggers
  - Totalview

Using Compiler options

- With the *-On* options
  - **0**: Fast compilation, disables optimization
  - **1,2**: low to moderate optimization, partial debugging support, disables inlining
- Using other compiler options
  - Options are vendor specific, but can be generally categorized as follows:
    - Target machine optimizations
    - Program behavior optimizations
    - Link loader options

Tuning for the memory subsystem

- Maximize cache reuse
  - Minimize stride
  - Block arrays
  - Avoid leading dimensions that are a multiple of a high power of two
- Encourage data prefetching to hide memory latency, e.g.
  - Loop fusion
  - Loop bisection
- Work within available physical memory

Minimize Stride

- Low-stride access increases cache efficiency, prefetch streaming
- Stride 1 is best (vector systems *love* stride-1 access)

- Fortran:

```
Real*8 :: a(m,n), b(m,n),
   c(m,n)
…
do i=1,n
  do j=1,m
     a(j,i)=b(j,i)+c(j,i)
  end do
end do
```

- C:

```
Double a[m][n], b[m][n],
   c[m][n];
…
for (i=0;i<m;i++){
  for (j=0;j<n;j++){
    a[i][j]=b[i][j]+c[i][j];
  }
}
```

(Note index order difference between Fortran and C examples)

Avoid Leading Dimensions That Are a Multiple of a High Power of 2

- When performing non-unit stride access of a 2D array, keep in mind the size of the cache line and cache associativity. Performance degrades when the stride is a multiple of the cache line size.

Example: consider an L1 cache that's 16 K in size and 4-way set associative, with a cache line of 64 B.

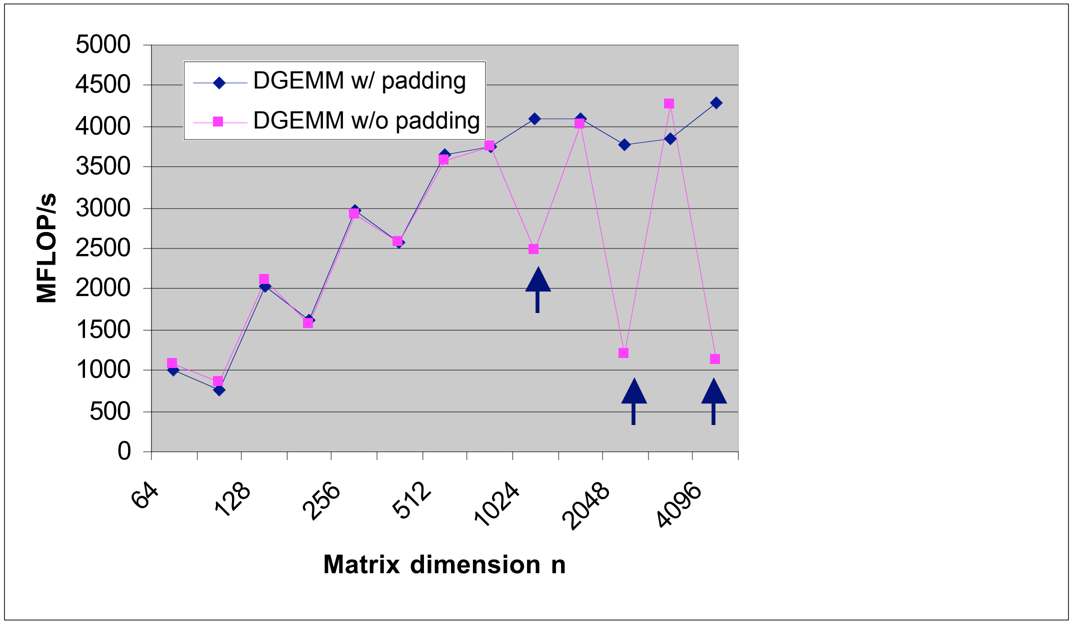
```
Real*8 :: a(1024,50)
. . .
do i=1,n
    a(1,i)=0.50*a(1,i)
end do
```

Effectively reduces L1 from to 256 cache lines to only *4!* You end up with a 256 byte cache.

Solution: Change leading dimension to 1028 (1024 + 1/2 cache line)

Example: The Itanium L2 Data Cache

Use of a high power of two for the leading dimension reduces effective size of cache



Matrix multiplication example:

C = ***A*** *x* ***B***

To prevent cache thrashing:

A(n,n) → A(n+pad,n)

B(n,n) → B(n+pad,n)

C(n,n) → C(n+pad,n)

where *pad* = 1/2 cache line

Matrix multiplication code which calls DGEMM and executed in parallel on two threads. Arrows indicate data points where the leading dimension of matrices A, B and C is a high power of two.

TACC

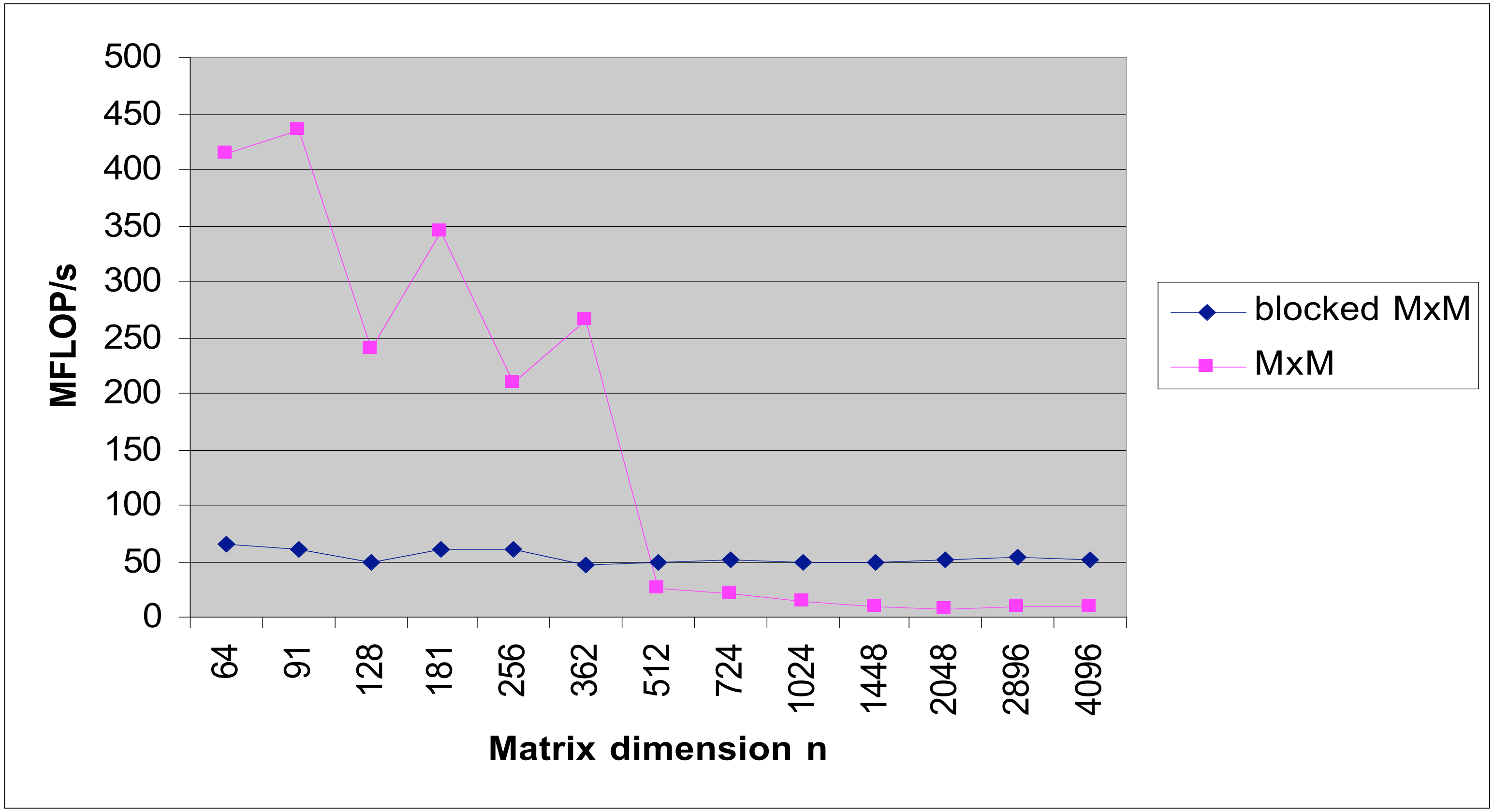
Data Cache Blocking

Example: Matrix multiplication

```
Real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb  ! <- nb is blocking factor
 do jj=1,n,nb
   do kk=1,n,nb
     do i=ii,min(n,ii+nb-1)
      do j=jj,min(n,jj+nb-1)
       do k=kk,min(n,kk+nb-1)
         c(i,j)=c(i,j)+a(j,k)*b(k,i)
       end do
      end do
     end do
   end do
 end do
end do
```

- Structure the computation into cache-sized pieces
- Pay the cost of moving data between memory and cache as infrequently as possible
- Reuse the data while it is in cache as much as possible before it is moved back to memory

Effect of data cache blocking on matrix-matrix multiplication on an Itanium system



Expose prefetch streams

- Loop fusion

```
do i=1,n
   x(i)=a(i)+b(i)
end do
do j=1,n+2
   y(i)=y(i)+c*x(i)
end do
```

```
do i=1,n
   x(i)=a(i)+b(i)
   y(i)=y(i)+c*(a(i)+b(i))
end do
y(n+1)=y(n+1)+c*(a(n+1)+b(n+1))
y(n+2)=y(n+2)+c*(a(n+2)+b(n+2))
```

- Loop bisection

```
do i=1,n
   x(i)=a(i)*b(i)
end do
```

```
n2=n/2
do i=1,n/2
   x(i)=a(i)*b(i)
   x(i+n2)=a(i+n2)*b(i+n2)
end do
if (mod(n,2) .ne. 0) x(n)=a(n)*b(n)
```

Expose prefetch streams

Example: dot-product

```
do i=1,n
 s=s+x(i)*y(i)
end do
dotp=s
```

2 streams

```
do i=1,n/2
 s0=s0+x(i)*y(i)
 s1=s1+x(i+n/2)*y(i+n/2)
end do
s0=s0+x(i)*y(i)
dotp=s0+s1
```

4 streams

```
do i=1,n/3
 s0=s0+x(i)*y(i)
 s1=s1+x(i+n/3)*y(i+n/3)
 s2=s2+x(i+2*n/3)*y(i+2*n/3)
end do
do i=3*(n/3)+1,n
 s0=s0+x(i)*y(i)
end do
dotp=s0+s1+s2
```

6 streams

Prefetching is the ability to predict the next cache line to be accessed and start bringing it in from memory. If data is requested far enough in advance, the latency to memory can be hidden. Compiler inserts prefetch instructions into loop—instructions that move data from main memory into cache in advance of their use. Prefetching may also be specified by the user using directives.

Work within available physical memory

- Working in virtual memory leads to performance degradation
  - TLB misses can be expensive
  - Swapping from memory to disk is slow
- Swapping can cause problems on Linux systems

Avoid using virtual memory by spreading data across more processes (works only for the MPP programming model)

Tuning for FP performance

1. Unroll inner loops to pipeline FP operations.
2. Avoid divides. When possible, multiply by the reciprocal.
3. Unroll outer loop to maximize FP/memory access ratio.

Unroll Inner Loops to Hide FP Latency

- Example: dot-product

```
. . .
do i=1,n,k
   s1 = s1 + x(i)*y(i)
   s2 = s2 + x(i+1)*y(i+1)
   s3 = s3 + x(i+2)*y(i+2)
   s4 = s4 + x(i+3)*y(i+3)
   . . .
   sk = sk + x(i+k)*y(i+k)
end do
. . .
dotp = s1 + s2 + s3 + s4+ … + sk
```

Unroll Outer Loops to Maximize FMA-to-Data Access Ratio

- Example: matrix-vector multiply

```
do i=1,n,k
  s1=y(i)
  s2=y(i+1)
  ...
  sk=y(i+k-1)
  do j=1,n
     s1=s1+A(j,i)*x(j)
     s2=s2+A(j,i+1)*x(j)
     ...
     sk=sk+A(j,i+k)*x(j)
  end do
  y(i)=s1
  y(i+1)=s2
  ...
  y(i+k)=sk
end do
```

- No unrolling:
  data access/FMA (in inner loop) = 2
- Unrolling to *k* depth:
  data access/FMA (in inner loop) = (*k*+1)/*k*

Avoid Divide Operations

- Example: Replace multiple divides with a multiply by the reciprocal

```
a=...
do i=1,n
   x(i)=x(i)/a
end do
```

```
a=...
ainv=1.0/a
do i=1,n
   x(i)=x(i)*ainv
end do
```

Even better, replace entire loop with call to optimized vector intrinsics library, if available.

Timers: Code Section

Routine	Type	Resolution (usec)	OS/Compiler
times	user/sys	1000	Linux/AIX/IRIX/ UNICOS
getrusage	wall/user/sys	1000	Linux/AIX/IRIX
gettimeofday	wall clock	1	Linux/AIX/IRIX/ UNICOS
rdtsc	wall clock	0.1	Linux
read_real_time	wall clock	0.001	AIX
system_clock	wall clock	system dependent	Fortran 90 Intrinsic
MPI_Wtime	wall clock	system dependent	MPI Library (C & Fortran)

Timers: Code Section

The **times**, **getrussage**, **gettimeofday**, **rdtsc**, and **read_real_time** timers have been packaged into a group of C wrapper routines (also callable from Fortran).

```
external   x_timer
real*8   ::  x_timer
real*8   :: sec0, sec1, tseconds
…
sec0     = x_timer()
  …Fortran Code
sec1     = x_timer()
tseconds = sec1-sec0
```

```
double x_timer(void);
        …
double sec0, sec1, tseconds;
…
sec0     = x_timer();
…C Codes
sec1     = x_timer();
tseconds = sec1-sec0
```

***X* = {one of rusage, gtod, rdtsc, rrt}**

www.tacc.utexas.edu/resources/user_guides/porting

Profilers: gprof (instrumentation)

- Intel: <compiler> -g –p prog.<x>
- IBM: <compiler> -pg prog.<x>
- Intel & IBM:

  gprof <executable> gmon.out

e.g.

ifc –g –p prog.f90

./a.out

gprof ./a.out gmon.out or gprof

-g provides more info on intrinsics & libs

generates gmon.out

a.out & gmon.out are defaults

Profilers: Example Code

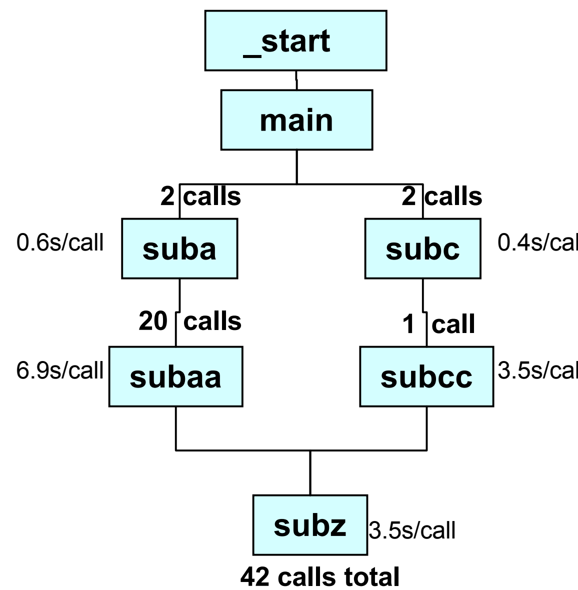
- Program Structure

Code Outline

```
program prof1
  ...
  do i = 1,2
    call suba(n,a,b,c)
  enddo
  do i = 1,2
    call subc(n,a,b,c)
  enddo
end

subroutine suba(n,a,b,c)
  ...
  call subaa(n,a,b,c)
end

subroutine subc(n,a,b,c)
  ...
  call subcc(n,a,b,c)
end
```

```
subroutine subaa(n,a,b,c)
  ...
  do i = 1,20
    ...
    call subz(n,a,b,c)
  end do
end

subroutine subcc(n,a,b,c)
  ...
  call subz(n,a,b,c)
end

subroutine subz(n,a,b,c)
...
end
```

Call Graph



Profiler Example: gprof (output)

```
granularity: each sample hit covers 4 byte(s)
             for 0.01% of 168.94 seconds
index % time  self  children   called  name
               0.44  168.49      1/1        _start [2]
[1]   100      0.44  168.49      1       main        [1]
               1.22  152.52      2/2        suba_   [3]
               0.88   13.87      2/2        subc_   [6]
-----------------------------------------------------
               1.22  152.52      2/2        main    [1]
[3]     91     1.22  152.52      2       suba_       [3]
              13.82  138.70      2/2        subaa_ [4]
-----------------------------------------------------
              13.82  138.70      2/2        suba_   [3]
[4]     90    13.82  138.70      2       subaa_      [4]
             138.70    0.00     40/42       subz_   [5]
-----------------------------------------------------
```

Profiler Example: gprof (output cont.)

```
            6.94    0.00      2/42       subcc_ [7]
          138.70    0.00     40/42       subaa_ [4]
[5]   86  145.64    0.00     42       subz_       [5]
---------------------------------------------------
            0.88   13.87     2/2          main    [1]
[6]    8    0.88   13.87     2        subc_       [6]
            6.93    6.94     2/2         subcc_ [7]
---------------------------------------------------
            6.93    6.94     2/2         subc_   [6]
[7]    8    6.93    6.94     2        subcc_      [7]
            6.94    0.00     2/42        subz_   [5]
---------------------------------------------------
```

Profiler Example: gprof (output)

- A common Unix profiling tool is **gprof**. Compiler options and libraries provide wrappers for each routine call (mcount), and periodic sampling the program counter (0.01 sec).

% time	cumulative secs	self secs	calls	self ms/call	total ms/call	name
86.21	145.6	145.6	42	3468	3468	subz_
8.18	159.4	13.8	2	6910	76262	subaa_
4.10	166.4	6.9	2	3465	6933	subcc_
0.72	167.6	1.2	2	610	76872	suba_
0.52	168.5	0.88	2	440	7372	subc_
0.26	168.9	0.44	2	440	168930	main
0.01	168.9	0.01	1			write

Profiling Parallel Programs (gprof)

mpif90 -qp prog.f90

Instruments code

setenv GMON_OUT_PREFIX gout.*

Forces each task to produce a gout.<pid>

Submit parallel job for executable
(in this case named a.out)

Produces gmon.out trace file

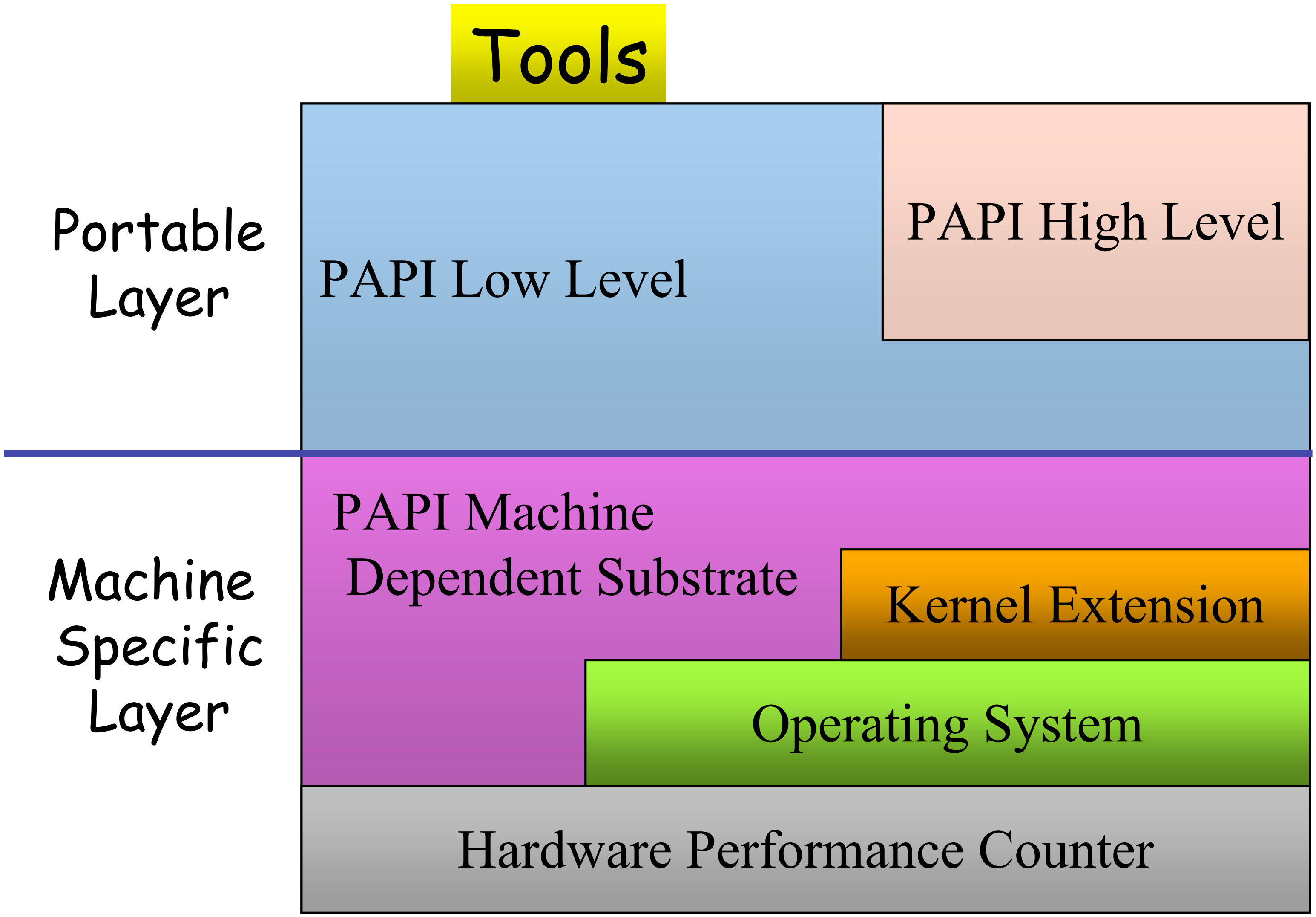
gprof -s gout.*

Combines gout.<pid> files into gmon.sum file

gprof a.out gmon.sum

Reads executable (a.out) & gmon.sum, report sent to STDOUT

PAPI Implementation



PAPI Performance Monitor

- Provides high level counters for events:
  - Floating point instructions/operations,
  - Total instructions and cycles
  - Cache accesses and misses
  - Translation Lookaside Buffer (TLB) counts
  - Branch instructions taken, predicted, mispredicted

- PAPI_flops routine for basic performance analysis
  - Wall and processor times
  - Total floating point operations and MFLOPS

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

- Low level functions are thread-safe, high level are not

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_flips
- PAPI_ipc

- Fortran interface
  PAPIF_start_counters
  PAPIF_read_counters
  PAPIF_stop_counters
  PAPIF_accum_counters
  PAPIF_num_counters
  PAPIF_flips
- PAPIF_ipc

Low-level Interface

- Increased efficiency and functionality over the high level PAPI interface
- About 40 functions
- Obtain information about the executable and the hardware
- Thread-safe
- Fully programmable
- Callbacks on counter overflow

PAPI Usage Example

```
 ievents(1)= PAPI_FP_INS
 ievents(2)= PAPI_L1_DCM
 call PAPIF_start_counters(ievents,ivnt,ierr)
 call PAPIF_read_counters( icounts,ivnt,ierr)
 icounts(1:ivnt) = 0

% Do Work
    :
%
 call PAPIF_accum_counters(icounts_a,ivnt,ierr)

% Do more work
   :
%
   call PAPIF_accum_counters( icounts_b,ivnt,ierr)
 print "(a35, 3i12)",” work counts”, icounts_a(1:ivnt)
 print "(a35, 3i12)",” more work counts", icounts_b(1:ivnt)
```

TACC

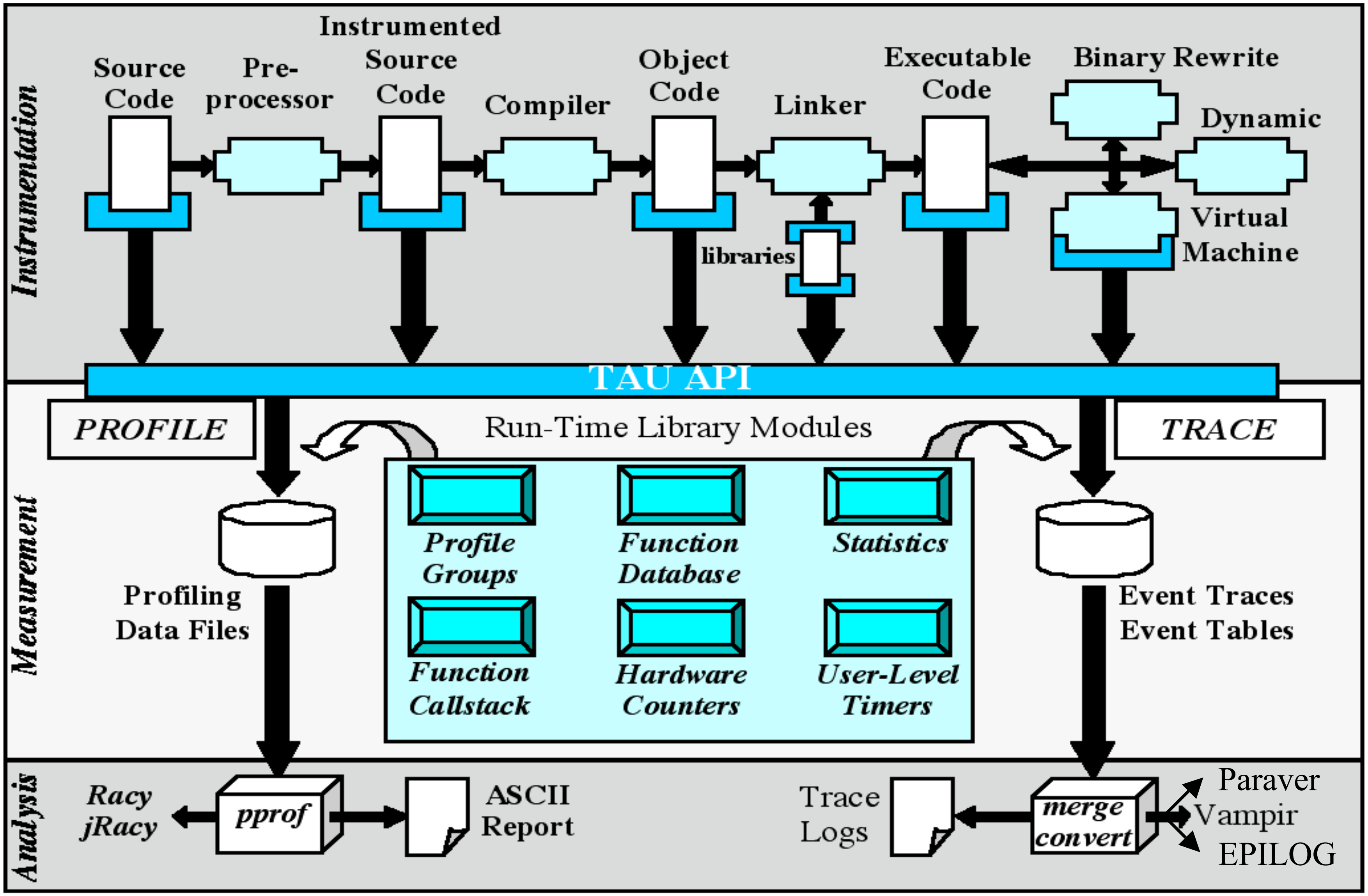
TAU Performance System Framework

- Tuning and Analysis Utilities
- 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, configurable performance profiling/tracing facility
  - Open software approach
- University of Oregon, LANL, FZJ Germany
- http://www.cs.uoregon.edu/research/paracomp/tau

TAU Performance Systems Goals

- Multi-level performance instrumentation
  - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
  - Computer system architectures and operating systems
  - Different programming languages and compilers
- Support for multiple parallel programming paradigms
  - Multi-threading, message passing, mixed-mode, hybrid
- Support for performance mapping
- Support for object-oriented and generic programming
- Integration in complex software systems and applications

TAU Performance System Architecture



TACC

Definitions – Profiling

- Profiling
  - Recording of summary information during execution
    - inclusive, exclusive time, # calls, hardware statistics, …
  - Reflects performance behavior of program entities
    - functions, loops, basic blocks
    - user-defined “semantic” entities
  - Very good for low-cost performance assessment
  - Helps to expose performance bottlenecks and hotspots
  - Implemented through
    - sampling: periodic OS interrupts or hardware counter traps
    - instrumentation: direct insertion of measurement code

Definitions – 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

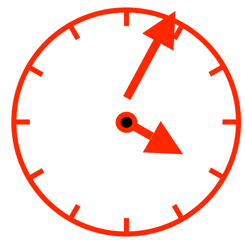
Event Tracing: Instrumentation, Monitor, Trace

CPU A:

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

CPU B:

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



timestamp

MONITOR

Event definition

1	master
2	slave
3	...

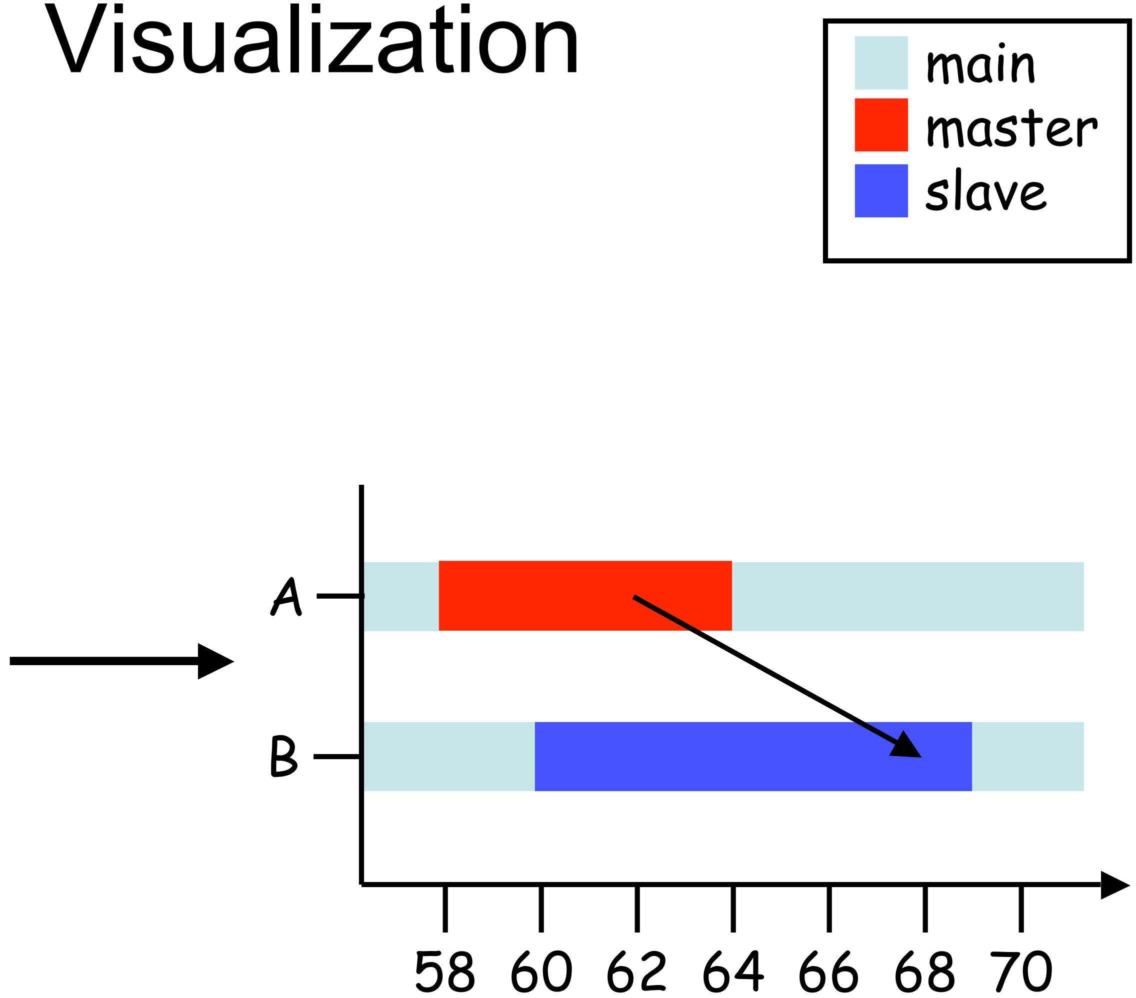
...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

TACC

Event 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
...			



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 (eliminate instrumentation in lightweight routines)

TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - manual (TAU API, TAU Component API)
    - 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*)
  - Proxy Components

Using TAU – A tutorial

- Configuration
- Instrumentation
  - Manual
  - MPI – Wrapper interposition library
  - PDT- Source rewriting for C,C++, F77/90/95
  - OpenMP – Directive rewriting
  - Component based instrumentation – Proxy components
  - Binary Instrumentation
    - DyninstAPI – Runtime Instrumentation/Rewriting binary
    - Java – Runtime instrumentation
    - Python – Runtime instrumentation
- Measurement
- Performance Analysis

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
  - -vtf=<dir> Specify location of VTF3 trace package

TAU Measurement System Configuration

- configure [OPTIONS]

-TRACE	Generate binary TAU traces
-PROFILE (default)	Generate profiles (summary)
-PROFILECALLPATH	Generate call path profiles
-PROFILEMEMORY	Track heap memory for each routine
-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

TAU Measurement Configuration – Examples (IBM)

- ./configure -c++=xlC_r –pthread
  - Use TAU with xlC_r and pthread library under AIX
  - Enable TAU profiling (default)
- ./configure -TRACE –PROFILE
  - Enable both TAU profiling and tracing
- ./configure -c++=xlC_r -cc=xlc_r
  -papi=/usr/local/packages/papi
  -pdt=/usr/local/pdtoolkit-3.2.1 –arch=ibm64
  -mpiinc=/usr/lpp/ppe.poe/include
  -mpilib=/usr/lpp/ppe.poe/lib -MULTIPLECOUNTERS
  - Use IBM's xlC_r and xlc_r compilers with PAPI, PDT, MPI packages and multiple counters for measurements
- Typically configure multiple measurement libraries

Including TAU's stub Makefile

```
include $(PET_HOME)/tau-2.13.7/rs6000/lib/Makefile.tau-mpi-pdt
F90 = $(TAU_F90)
CC  = $(TAU_CC)
LIBS = $(TAU_MPI_FLIBS) $(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 $@
```

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 – 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
```

AutoInstrumentation using TAU_COMPILER

- $(TAU_COMPILER) stub Makefile variable in beta release (v2.13.7+)
- Invokes PDT parser, TAU instrumentor, compiler through **`tau_compiler.sh`** shell script
- Requires minimal changes to application Makefile
  - Compilation rules are not changed
  - User adds $(TAU_COMPILER) before compiler name
    - F90=mpxlf90
      Changes to
      F90= $(TAU_COMPILER) mpxlf90
- Passes options from TAU stub Makefile to the four compilation stages
- Uses original compilation command if an error occurs

Using MPI Wrapper Interposition

Step I: Configure TAU with MPI:

```
% configure -mpiinc=/usr/lpp/ppe.poe/include
  -mpilib=/usr/lpp/ppe.poe/lib -arch=ibm64 -c++=xlC_r
  -cc=xlc_r -pdt=$PET_HOME/pdtoolkit-3.2.1
% make clean; make install
```

Builds <taudir>/<arch>/lib/libTauMpi<options>,

<taudir>/<arch>/lib/Makefile.tau<options> and libTau<options>.a

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

Description of Optional Packages

- PAPI – Measures hardware performance data e.g., floating point instructions, L1 data cache misses etc.[UTK]
- DyninstAPI – Helps instrument an application binary at runtime or rewrites the binary [UMD/U.Wisc Madison]
- EPILOG – Trace library. Epilog traces can be analyzed by EXPERT [UTK, FZJ], an automated bottleneck detection tool. Part of KOJAK (CUBE, EPILOG, Opari)
- Opari – Tool that instruments OpenMP programs (UTK, FZJ)
- Traceanalyzer (Intel) – Commercial trace visualization tool developed by TU Dresden/FZJ [Intel/Pallas]
- Paraver – Trace visualization tool [CEPBA]

TACC

Program Database Toolkit (PDT)

- Program code analysis framework for developing 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
- Target and integrate multiple source languages
- Use in TAU to build automated performance instrumentation tools

Performance Mapping

- Associate performance with "significant" entities (events)
- Source code points are important
  - Functions, regions, control flow events, user events
- Execution process and thread entities are important
- Some entities are more abstract, harder to measure

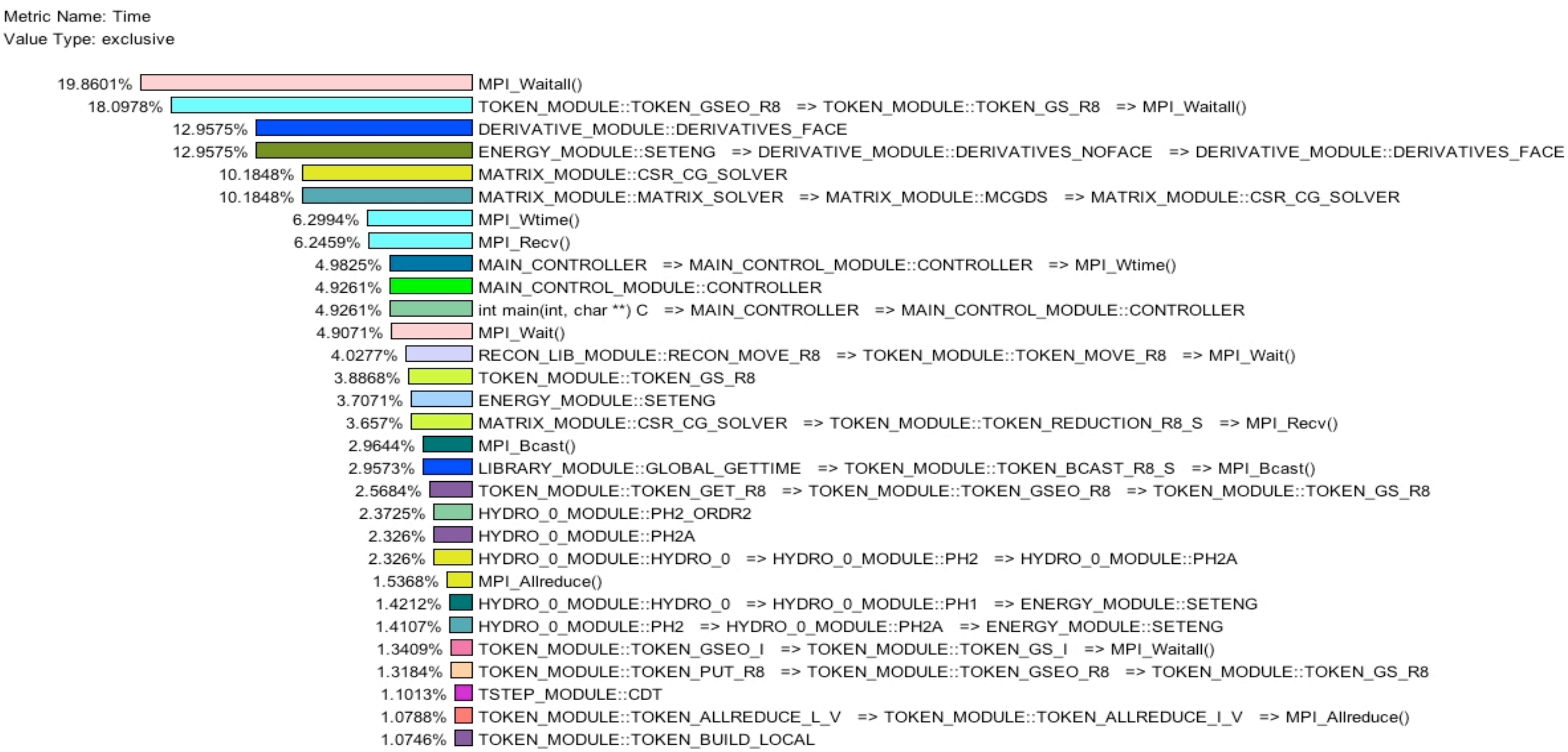
Performance Mapping in Callpath Profiling

- Consider callgraph (callpath) profiling
  - Measure time (metric) along an edge (path) of callgraph
    - Incident edge gives parent / child view
    - Edge sequence (path) gives parent / descendant view
- Callpath profiling when callgraph is unknown
  - Must determine callgraph dynamically at runtime
  - Map performance measurement to dynamic call path state
- Callpath levels
  - 1-level: current callgraph node/flat profile
  - 2-level: immediate parent (descendant)
  - *k*-level: *k*th nodes in the calling path

k-Level 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

k-Level Callpath Implementation in TAU



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]
     0.1133        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]
```

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 (SDDF, VTF, Paraver)
  - Trace visualization using Intel Trace Analyzer (Pallas VAMPIR/Intel)

Pprof Output (NAS Parallel Benchmark – LU)

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI

emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1	3:11.293	1	15	191293269	applu
99.6	3,667	3:10.463	3	37517	63487925	bcast_inputs
67.1	491	2:08.326	37200	37200	3450	exchange_1
44.5	6,461	1:25.159	9300	18600	9157	buts
41.0	1:18.436	1:18.436	18600	0	4217	MPI_Recv()
29.5	6,778	56,407	9300	18600	6065	blts
26.2	50,142	50,142	19204	0	2611	MPI_Send()
16.2	24,451	31,031	301	602	103096	rhs
3.9	7,501	7,501	9300	0	807	jacld
3.4	838	6,594	604	1812	10918	exchange_3
3.4	6,590	6,590	9300	0	709	jacu
2.6	4,989	4,989	608	0	8206	MPI_Wait()
0.2	0.44	400	1	4	400081	init_comm
0.2	398	399	1	39	399634	MPI_Init()
0.1	140	247	1	47616	247086	setiv
0.1	131	131	57252	0	2	exact
0.1	89	103	1	2	103168	erhs
0.1	0.966	96	1	2	96458	read_input
0.0	95	95	9	0	10603	MPI_Bcast()
0.0	26	44	1	7937	44878	error
0.0	24	24	608	0	40	MPI_Irecv()
0.0	15	15	1	5	15630	MPI_Finalize()
0.0	4	12	1	1700	12335	setbv
0.0	7	8	3	3	2893	l2norm
0.0	3	3	8	0	491	MPI_Allreduce()
0.0	1	3	1	6	3874	pintgr
0.0	1	1	1	0	1007	MPI_Barrier()
0.0	0.116	0.837	1	4	837	exchange_4
0.0	0.512	0.512	1	0	512	MPI_Keyval_create()
0.0	0.121	0.353	1	2	353	exchange_5
0.0	0.024	0.191	1	2	191	exchange_6
0.0	0.103	0.103	6	0	17	MPI_Type_contiguous()

--:-- NPB_LU.out (Fundamental)--L8--Top---

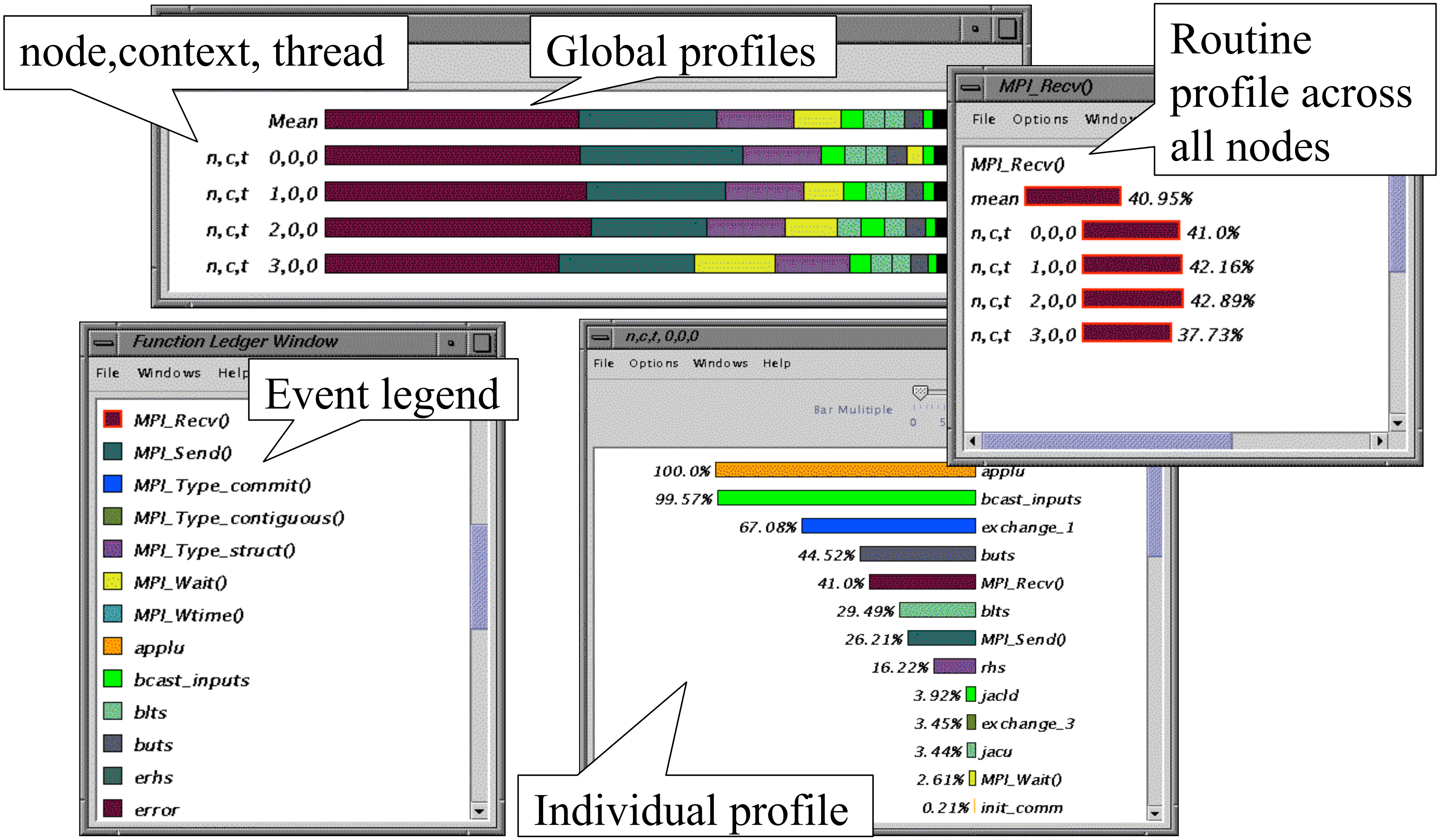
Terminology – Example

- For routine “int main( )”:
- Exclusive time
  - 100-20-50-20=10 secs
- Inclusive time
  - 100 secs
- Calls
  - 1 call
- Subrs (no. of child routines called)
  - 3
- Inclusive time/call
  - 100secs

```
int main( )
{ /* takes 100 secs */

  f1(); /* takes 20 secs */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */

  /* other work */
}

/*
Time can be replaced by  counts
from PAPI e.g., PAPI_FP_INS. */
```

TACC

ParaProf (NAS Parallel Benchmark – LU)



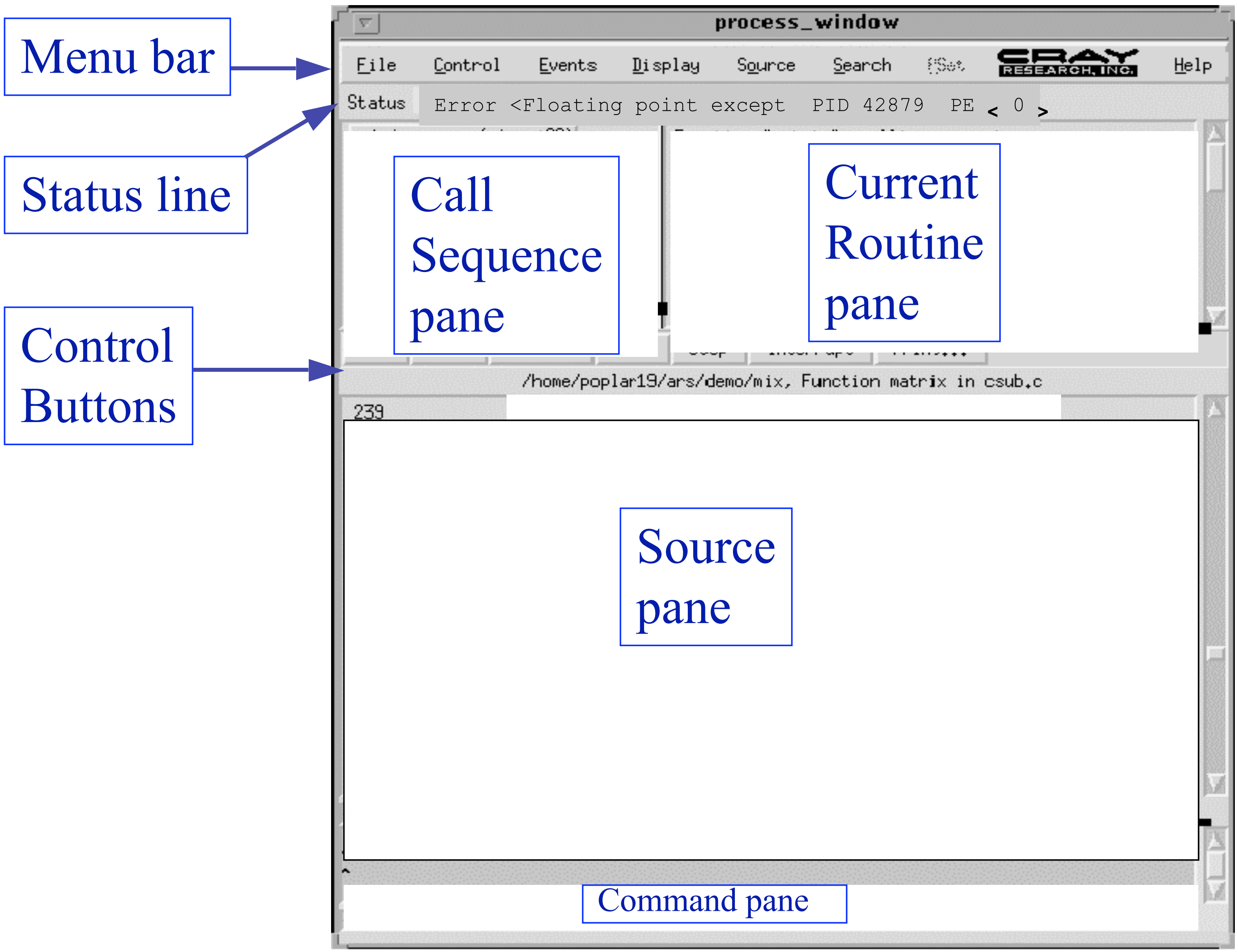
Debugging: Totalview

Interactive, parallel, symbolic debuggers with GUI interface

- Works with C, C++ and Fortran Compilers
- Available on my different platforms.
  (IBM, CRAY, AMD, INTEL, SUN, SGI, …)
- Supports Pthread, OpenMP & MPI
  (and hybrid paradigm)
- Support 32- and 64-bit architectures
- Simple to use (intuitive)

Instrumenting Code and Running TotalView

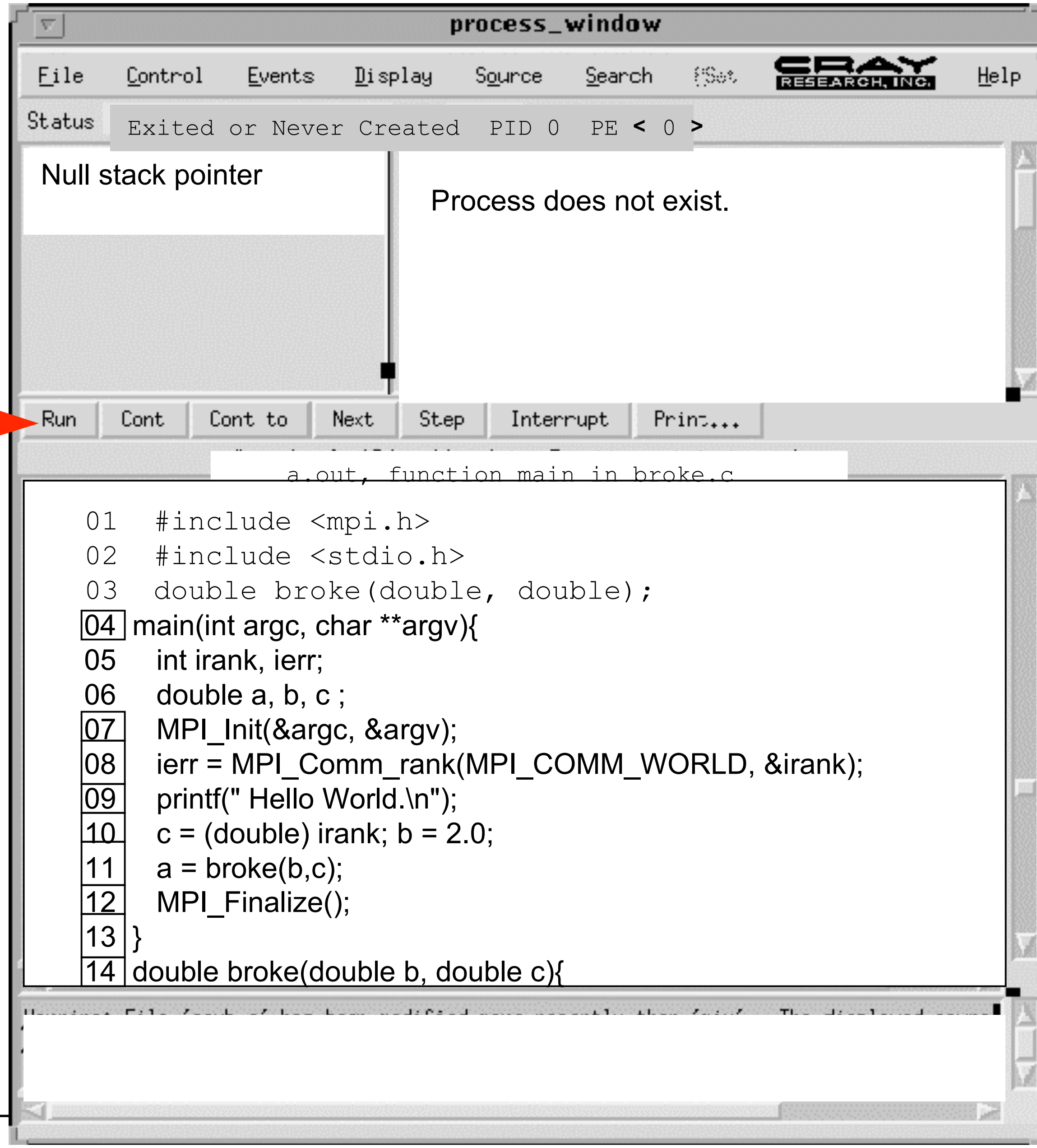
% module load totalview {sets environment variables}
% <compiler> –g prog.f90
% setenv DISPLAY tnek.tacc.utexas.edu:0.0
% totalview a.out


Menu bar
Status line
Control Buttons
process_window
File Control Events Display Source Search Help
CRAY RESEARCH, INC.
Status
Error <Floating point except PID 42879 PE < 0 >
Call Sequence pane
Current Routine pane
/home/poplar19/ars/demo/mix, Function matrix in csub.c
239
Source pane
Command pane

Process Window active mode, with symbol table.

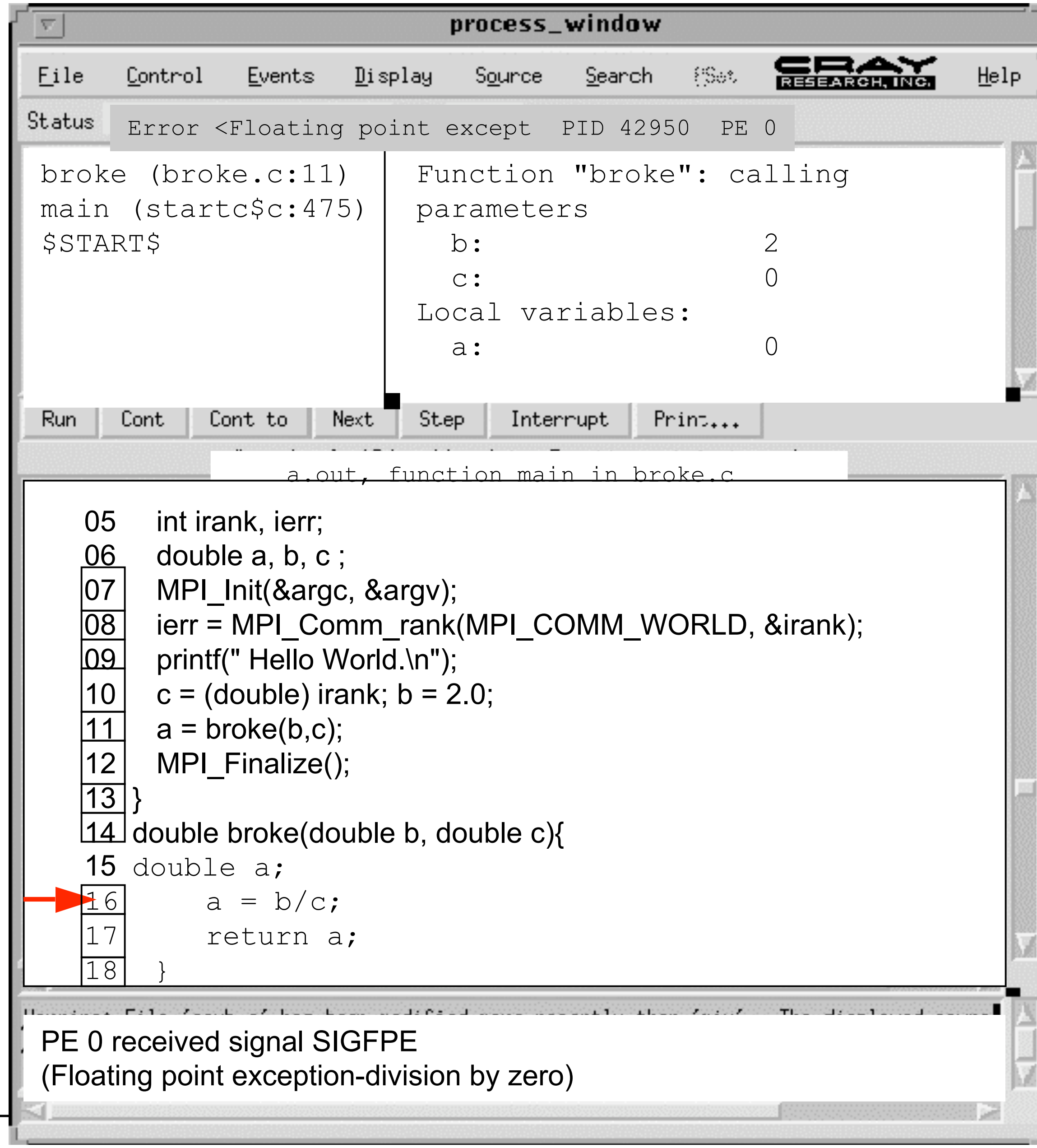
Left Click

% cc -g broke.c



Process Window active mode, with symbol table.

% cc -g broke.c



Diving:
Right click on variable to display more information.

Right Click