Tuning and Analysis Utilities

TAU Portable Performance Profiling Tools

Sameer Shende
Department of CIS, University of Oregon,
Advanced Computing Laboratory, Los Alamos National Laboratory
sameer@cs.uoregon.edu

TAU Profiling Team Members

(In alphabetical order)

Peter Beckman (LANL)
Prof. Janice Cuny (UO)
Steve Karmesin (LANL)
Kathleen Lindlan (UO)
Prof. Allen D. Malony (UO)
Sameer Shende (UO, LANL)

Tuning and Analysis Utilities

TAU: Tuning and Analysis Utilities

Tau is...

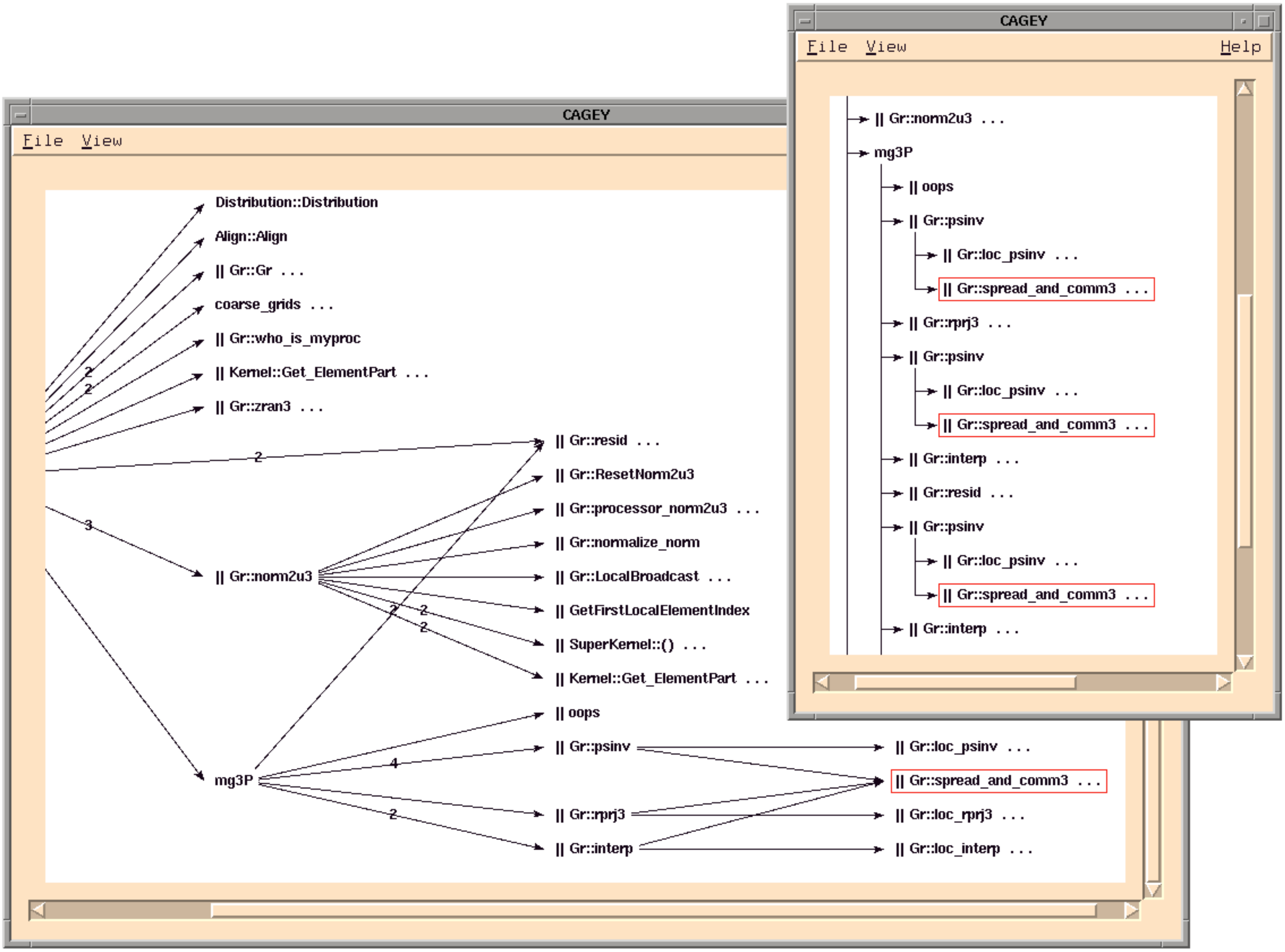
- An extensible tool framework supporting tool interactions with the program, the user, and each other
- A graphical, program development environment with several distinct and unique, but completely integrated, tools
- A performance analysis environment facilitating *static* and *dynamic* interactions with programs

Tau is designed to...

- Operate on language-level program objects of high-level parallel programming languages
- Be extensible in many dimensions, easing creation of additional tools, retargeting to new languages, and porting to new machine environments
- Be tightly integrated through well-defined interfaces with both compilers and runtime systems

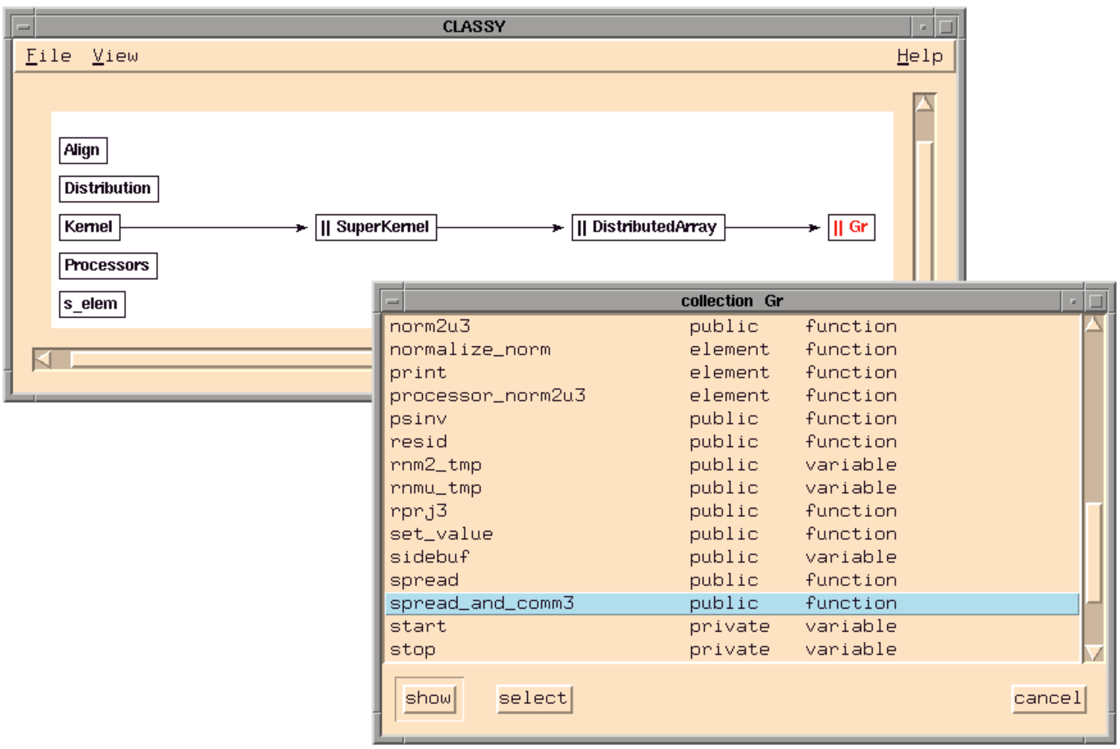
Tuning and Analysis Utilities

TAU Static Callgraph Display Tool (CAGEY)



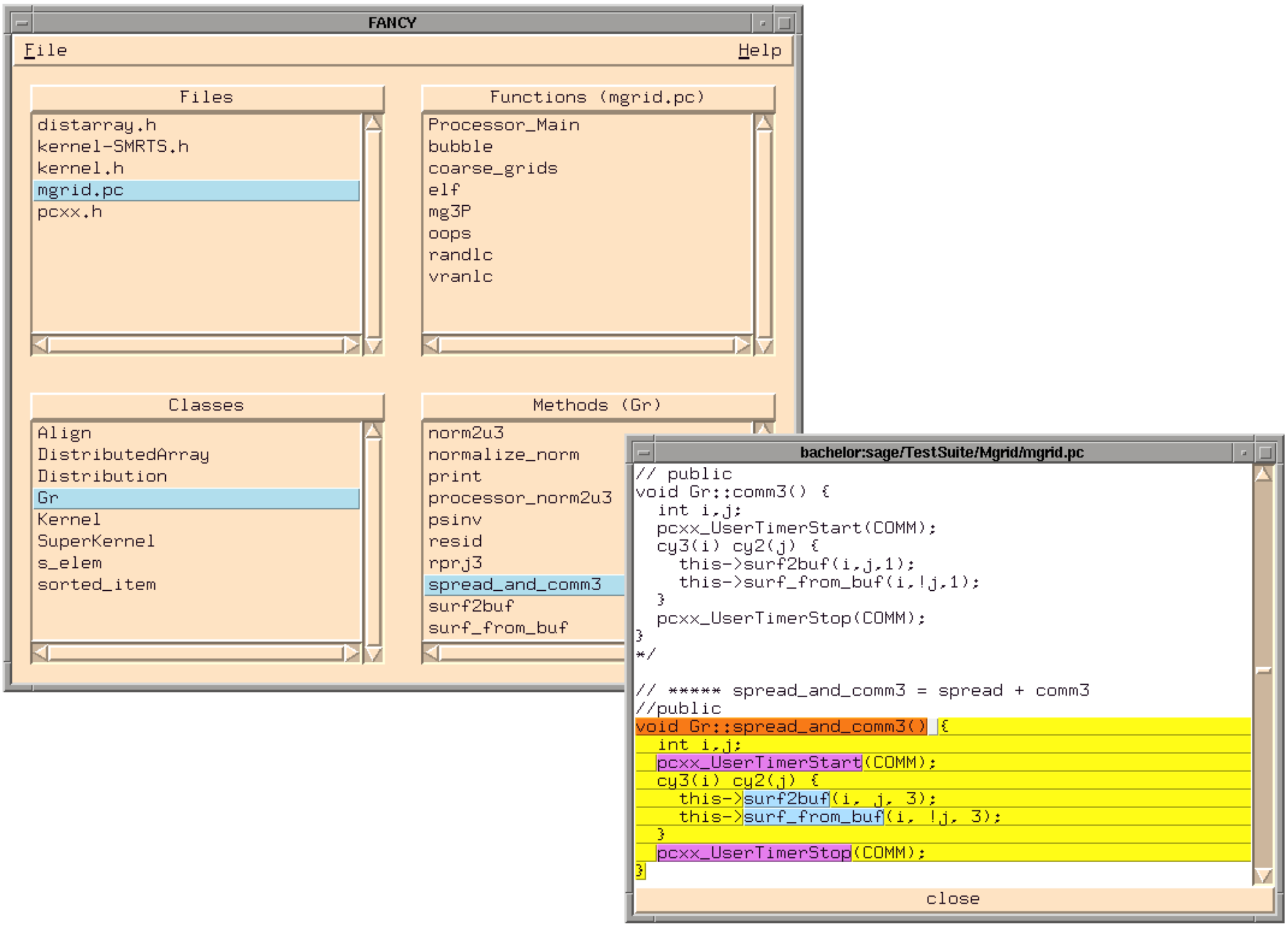
Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

TAU Class Hierarchy Browser (CLASSY)



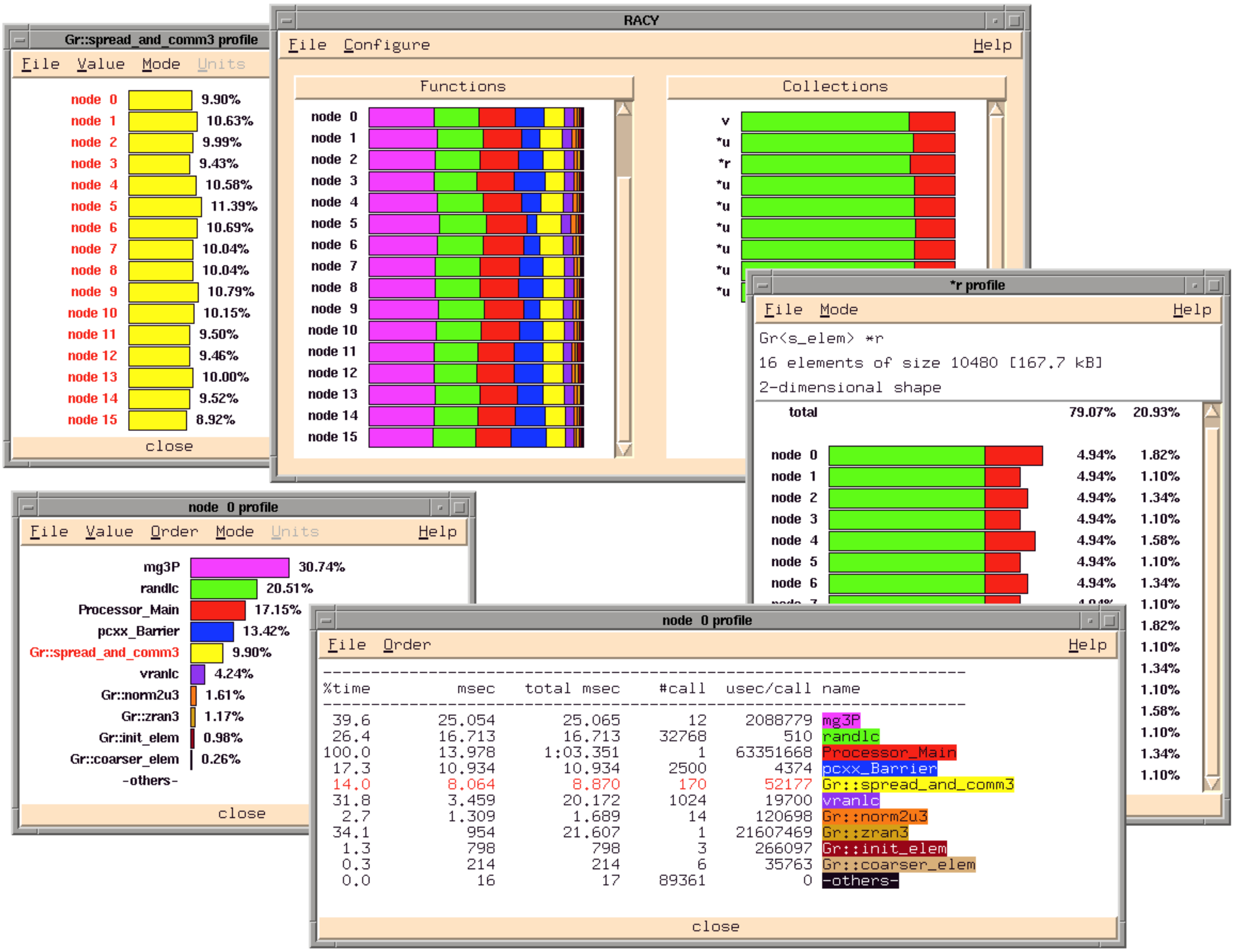
Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

TAU File and Class Browser (FANCY)



Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

TAU Performance Profiling Tool (RACY)



Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

Performance Profiling

"The purpose of performance profiling is to find out where a program is spending its time - in precisely which procedures or lines of code"

Performance Profiling Approaches

- Callstack sampling:
  Periodically examine and record the program counter and call-stack and resource consumption. Estimate performance based on samples. (e.g., CC -p, Speedshop SGI)

- Instrumentation:
  For counting the exact number of times a function executed. Insert binary instrumentation to generate a trap at function exit. (e.g., ideal CPU time (pixie) experiment SGI).

- Binary instrumentation:
  Runtime instrumentation to identify bottlenecks (e.g., Dyn-Inst, Paradyne U. Wisconsin).

- Trace based performance prediction:
  Generate trace of each function entry/exit and report performance statistics (e.g., Pablo UIUC, Vampir Pallas, Germany)

- Exception Trace:
  Records the location of floating point exceptions (e.g., Speedshop SGI)

- Event based sampling :
  (hardware performance counters, VTune)

TAU Performance Profiling Model

Instrumentation of source code

- Instrumentation is inserted either manually or by a preprocessor in the source code.
- Function (block) entry and exit points are used to record the exact time spent in the instrumented function.
- Statistics maintained in the instrumented program and small profile files generated at the end of execution.
- Supported for C++ and its parallel derivatives (pC++, HPC++)
- Compiler support needed for automatic instrumentation (Sage++, EDG).
- Higher level profiling. Suitable for C++ libraries, applications where source code is available for instrumentation.

Design goals for TAU Profiling and Tracing

- Provide a consistent, portable profiling and tracing library that would work with multiple compilers, operating systems and platforms for parallel C++ libraries and applications.
- Profiling should report the exact time spent in each instrumented function instead of an estimate.
- Overhead of profiling or tracing should be limited to the groups of profiled functions selected at runtime.
- Lightweight profiling facility.
- When profiling and tracing are disabled, the instrumented code should run without any runtime overhead (default).
- Time spent in templates should be reported based on distinct template instantiations.
- TAU should support the ISO/ANSI standard for C++.
- It should be possible to use statement level user defined timers for profiling a set of statements.
- Hardware performance counters may be used instead of time.

Terminology: Exclusive and Inclusive

"Inclusive time is the total time spent in a function and all the ***other*** *functions it calls directly or indirectly.*

Exclusive time is the time spent in the function only."

What data can TAU generate?

- What was the time spent exclusively and inclusively in each function? template? (based on instantiations)
- How many times was each function called?
- How many profiled functions (subroutines) did it call?
- What is the mean time/call for each function?
- What was the mean time spent in a function over all nodes? contexts? threads?
- For each invocation of a function, what was the exclusive and inclusive time spent in it? (Trace)
- What is the standard deviation of exclusive time ? (Statistics)
- Can we replace “Time” by “flops”? Instructions issued? Cycles? Secondary data cache misses?(R10000 HW counters)
- Can we profile only Communication functions? Comm + IO? (Selective Profiling)
- Can we profile a set of statements (finer granularity) instead of functions? Can we profile blocks such as for loops?

Profiling Templates and Functions in C++

```
template<class ForwardIterator>
Message& Message::put(ForwardIterator beg,
    ForwardIterator end) {
// Code
}

int main(int argc, char **argv) {
// Code
}
```

- Identify functions based on:

    1. Name: “main()”
    2. Type information: “int (int, char **)”
       e.g., “Message::put() Message (vec-tor<int>::iterator, vec-tor<int>::iterator)”

- Group related functions in a profile group e.g., TAU_FIELD, TAU_IO, TAU_USER

- Insert instrumentation macro in each function.

TAU Profiling API

- **TAU_PROFILE(func_name, type_string, profile_group);**

Inserted in each function that is to be profiled. e.g.,

```
int main(int argc, char **argv){
   TAU_PROFILE("main()", "int (int, char **)",
         TAU_DEFAULT);
```

- **TAU_TYPE_STRING(string_varname, string);**
- **TAU_PROFILE(func_name, string_varname, profile_group);**
- **CT(object);**

For profiling template member functions.

- **TAU_PROFILE_TIMER(var, name, type, group);**
- **TAU_PROFILE_START(var);**
- **TAU_PROFILE_STOP(var);**

To start and stop a timer. To time one or more statements in the code.

```
TAU_PROFILE_TIMER(timer1,"main-loop1", "int (int, char
    **)", TAU_USER);
        ...
        TAU_PROFILE_START(timer1);
         for(i=0; i < N; i++) {// loop1 profiled using
             timer1 var }
        TAU_PROFILE_STOP(timer1);
```

- **TAU_PROFILE_STMT(stmt);**

To declare a variable that is used only during profiling.

- **TAU_PROFILE_INIT(argc, argv);**
- **TAU_PROFILE_SET_NODE(myNode);**

To initialize profile groups and to set the current node id for each context

- **TAU_PROFILE_EXIT(const char *message);**

To abort the program, and dump profiling data to disk.

- **TAU_TRACE_SENDMSG(type, destination, length);**
- **TAU_TRACE_RECVMSG(type, source, length);**

pprof showing template instantiations

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive    #Call    #Subrs  Inclusive Name
              msec     total msec                   usec/call
---------------------------------------------------------------------------------------
100.0       23,145    36:36.980        1     14378 2196980000 main
    int(int,char*[])
 12.1     4:26.306     4:26.306    45792         0       5816 apply BrickExpres-
    sion<3U, LField<double, 3U>::iterator, LField<double, 3U>::iterator, OpEx-
    trapolate<double>>
 11.2          234     4:06.693      100       196    2466930 assign(Indexed-
    BareField) PETE_TBTree<OpDivide, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpMultiply, Indexed-
    BareField<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpMultiply, IndexedBare-
    Field<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpMultiply,
    IndexedBareField<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<double,
    3U, 3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpMulti-
    ply, IndexedBareField<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<dou-
    ble, 3U, 3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t,
    PETE_TBTree<OpMultiply, IndexedBareField<double, 3U, 3U>::PETE_Expr_t,
    IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpMultiply,
    IndexedBareField<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<double,
    3U, 3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpMulti-
    ply, IndexedBareField<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<dou-
    ble, 3U, 3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t,
    PETE_TBTree<OpMultiply, IndexedBareField<double, 3U, 3U>::PETE_Expr_t,
    IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t>::PETE_Expr_t, PETE_TBTree<OpAdd,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, IndexedBare-
    Field<double, 3U, 3U>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t, IndexedBareField<double, 3U,
    3U>::PETE_Expr_t>::PETE_Expr_t>
10.9     3:59.772     3:59.772       96         0    2497625 apply BrickExpres-
    sion<3U, LField<double, 3U>::iterator, PETE_Expr<PETE_TBTree<OpDivide,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd,
    PETE_TBTree<OpAdd, PETE_TBTree<OpAdd, PETE_TBTree<OpAdd,
```

Selective Profiling

Table 1: Selecting profile groups on the command line at runtime

ProfileGroup	Description	Runtime Identifier	Example
TAU_DEFAULT	All profiling groups enabled		--profile
TAU_MESSAGE	Message class includes MPI, PVM, ACLMPL	‘m’	--profile M
TAU_PETE	Portable Expression Template Engine	‘p’	--profile pete+message
TAU_VIZ	ACLVIZ	‘v’	--profile Pete+viz
TAU_ASSIGN	Assign expression	‘a’	--profile a+m
TAU_IO	IO functions	‘i’	--profile io
TAU_FIELD	Field functions	‘f’	--profile field+viz+assign
TAU_LAYOUT	Field Layout	‘l’	--profile layout
TAU_SPARSE	Sparse Index	‘s’	--profile sparse+assign
TAU_DOMAINMAP	Domain Map	‘d’	--profile i+d+viz
TAU_UTILITY	Utility	‘ut’	--profile utils+io
TAU_USER	User	‘u’	--profile user+region
TAU_USER1	User1	‘1’	--profile 1+d
TAU_USER2	User2	‘2’	--profile 2+d
TAU_USER3	User3	‘3’	--profile 3+d
TAU_USER4	User4	‘4’	--profile 4+d

% mpirun -np 4 app --profile io+field+message

Hardware Performance Counters

- TAU approach: Function entry and exit points reset counters and read counter values from 64 bit register and accumulate these. Precise counts associated with each profiled function are reported. Counter lookup is expensive.

- SpeedShop: Program counter sampling done when a counter reaches a threshold. Kernel can preload the 64 bit register with some initial value. Interrupt received every "n" counter overflows.

- libperfex API provided by SGI is used.

- Configuration:

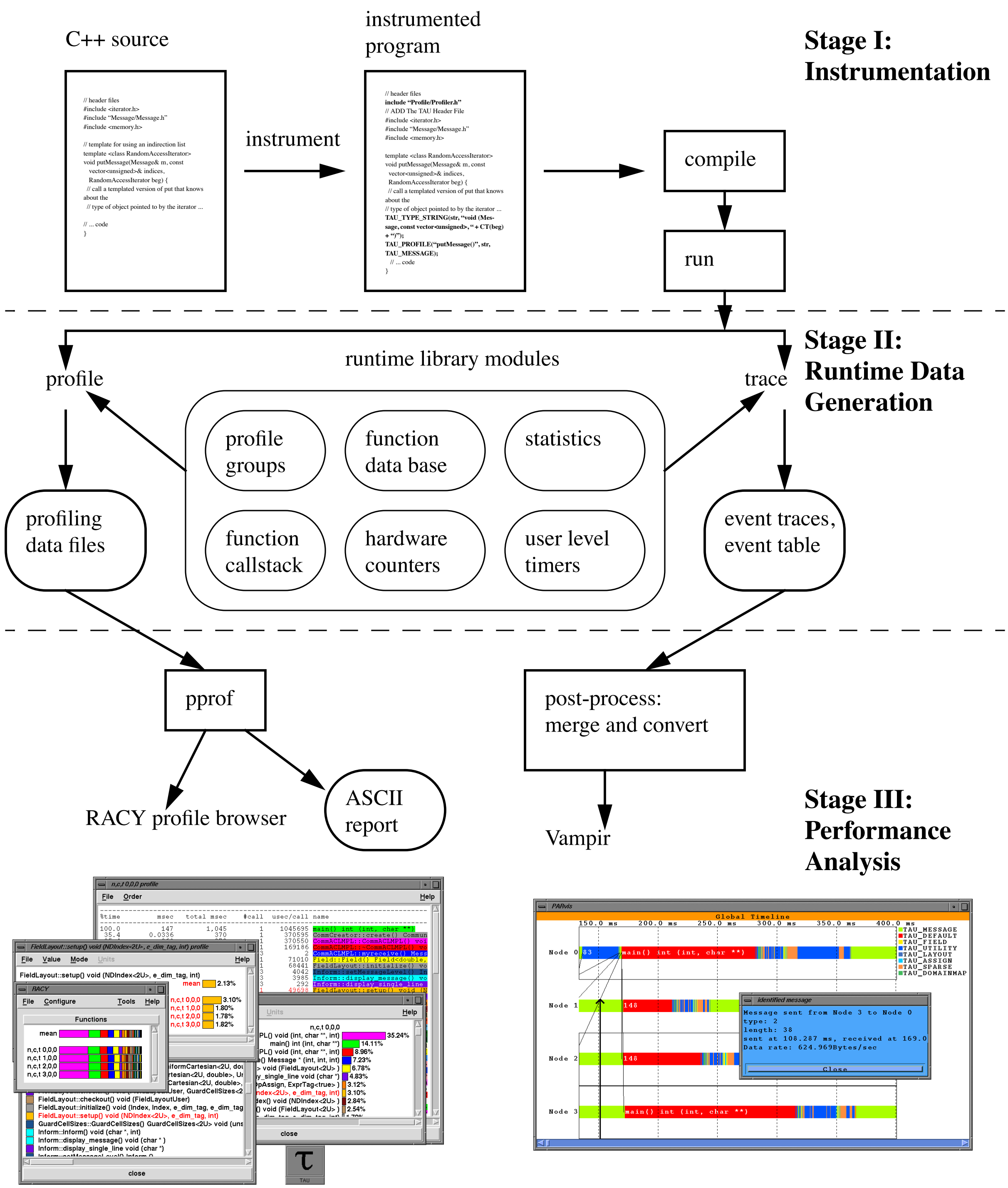
```
% ./configure -PROFILECOUNTERS
% make install
% setenv T5_EVENT0 26
# profile secondary data cache misses
% mpirun -np 4 app --profile io+field
```

SGI R10000 Performance Counters

Table 1: Values of T5_EVENT0 environment variable

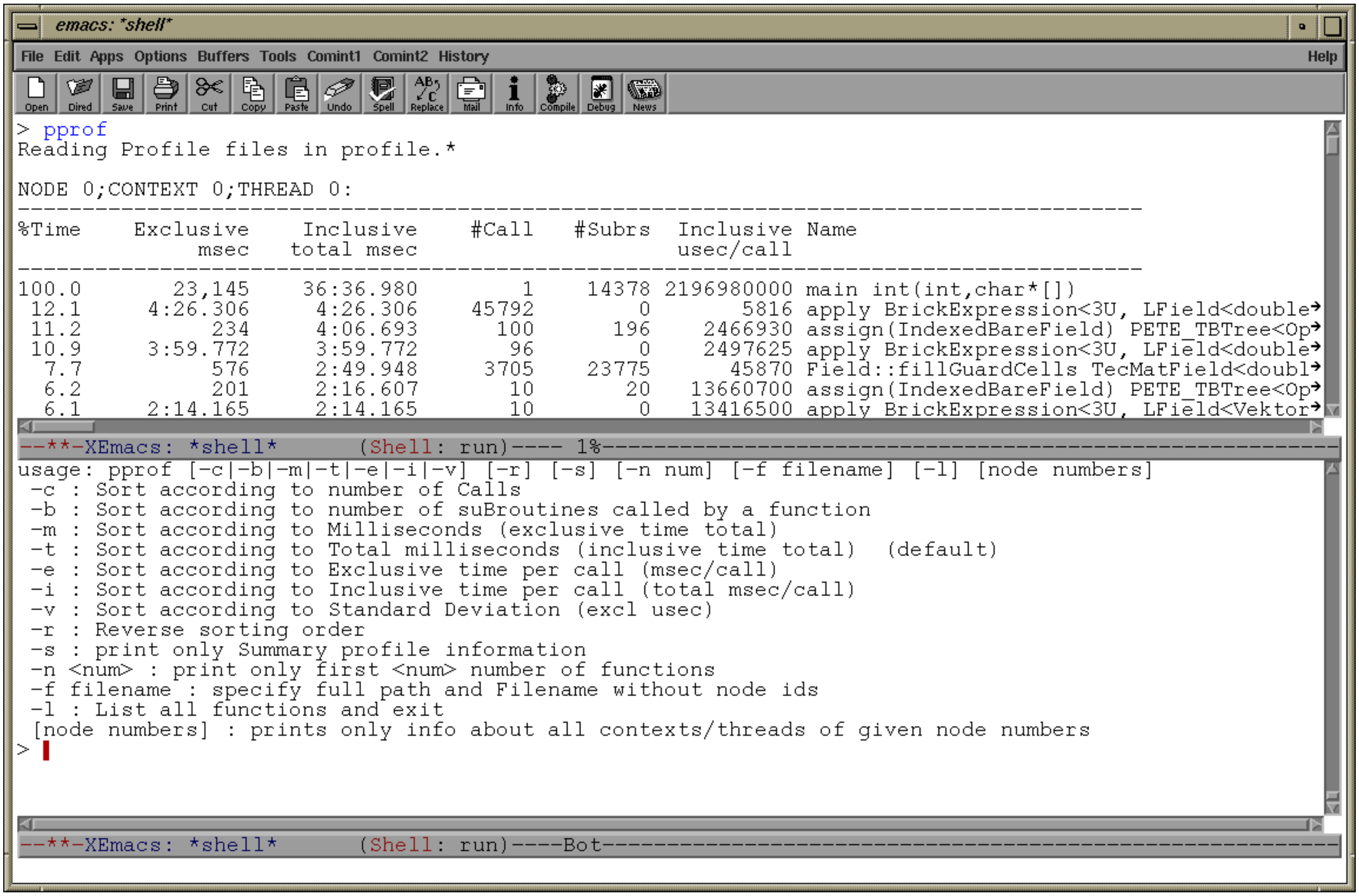
	Quantity Profiled		Quantity Profiled
0	Cycles	16	Graduated cycles
1	Issued instructions	17	Graduated instructions
2	Issued loads	18	Graduated loads
3	Issued stores	19	Graduated stores
4	Issued store conditionals	20	Graduated store conditionals
5	Failed store conditionals	21	Graduated floating point instructions
6	Decoded branches	22	Quadwords written back from primary data cache
7	Quadwords written back from scache	23	TLB misses
8	Correctable scache data array ECC errors	24	Mispredicted branches
9	Primary instruction cache misses	25	Primary data cache misses
10	Secondary instruction cache misses	26	Secondary data cache misses
11	Instruction misprediction from scache way prediction table	27	Data misprediction from scache way prediction table
12	External interventions	28	External intervention hits in scache
13	External invalidations	29	External invalidation hits in scache
14	Virtual coherency conditions	30	Store/prefetch exclusive to clean block in scache
15	Graduated instructions	31	Store/prefetch exclusive to shared block in scache

TAU Portable Profiling and Tracing



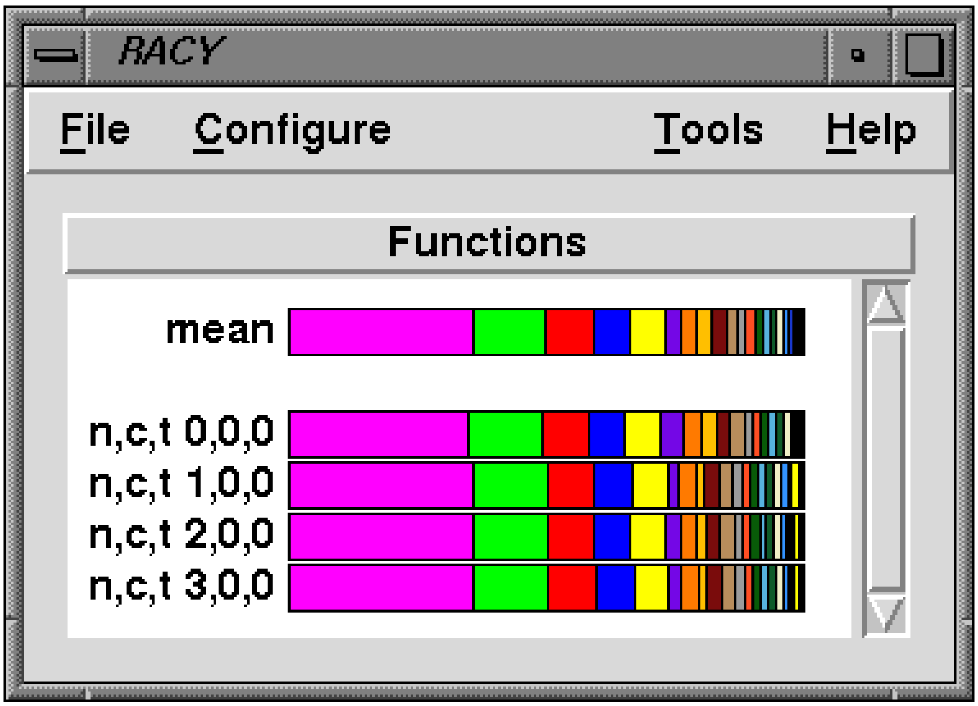
Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

pprof - Profile Data Sorter and Display



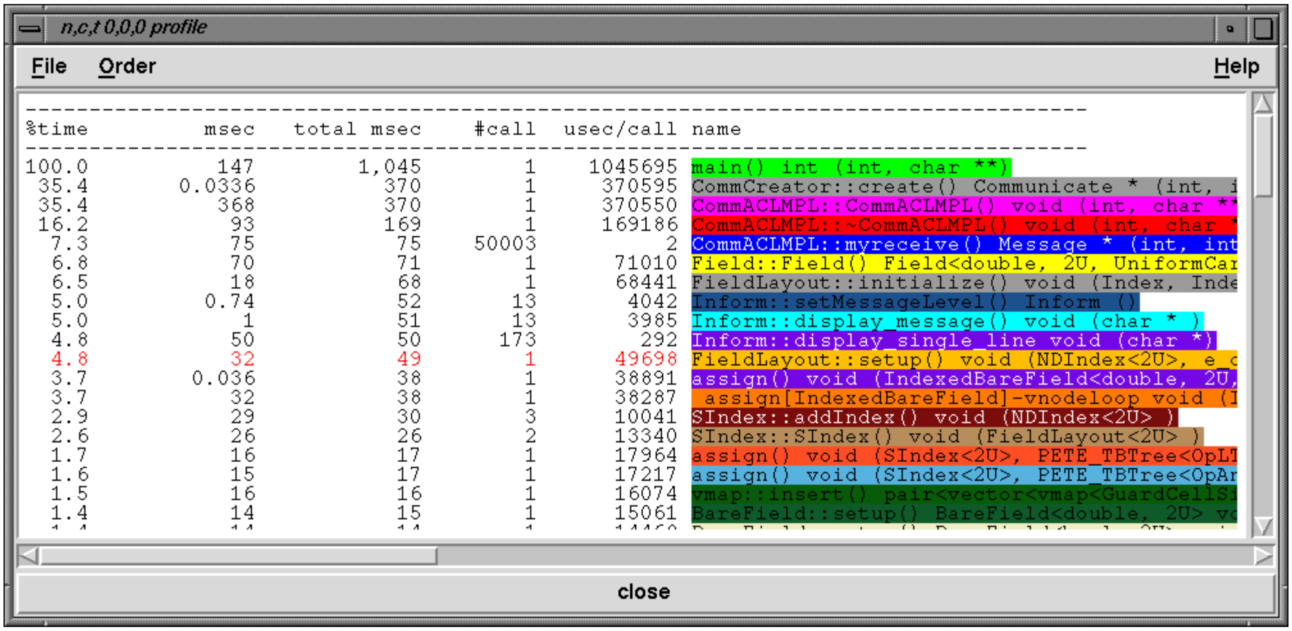
```
> pprof
Reading Profile files in profile.*

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive      #Call      #Subrs  Inclusive Name
              msec   total msec                         usec/call
---------------------------------------------------------------------------------------
100.0       23,145    36:36.980          1       14378 2196980000 main int(int,char*[])
 12.1     4:26.306     4:26.306      45792           0       5816 apply BrickExpression<3U, LField<double→
 11.2          234     4:06.693        100         196    2466930 assign(IndexedBareField) PETE_TBTree<Op→
 10.9     3:59.772     3:59.772         96           0    2497625 apply BrickExpression<3U, LField<double→
  7.7          576     2:49.948       3705       23775      45870 Field::fillGuardCells TecMatField<doubl→
  6.2          201     2:16.607         10          20   13660700 assign(IndexedBareField) PETE_TBTree<Op→
  6.1     2:14.165     2:14.165         10           0   13416500 apply BrickExpression<3U, LField<Vektor→
--**-XEmacs: *shell*        (Shell: run)---- 1%-----------------------------------------------
usage: pprof [-c|-b|-m|-t|-e|-i|-v] [-r] [-s] [-n num] [-f filename] [-l] [node numbers]
 -c : Sort according to number of Calls
 -b : Sort according to number of suBroutines called by a function
 -m : Sort according to Milliseconds (exclusive time total)
 -t : Sort according to Total milliseconds (inclusive time total)  (default)
 -e : Sort according to Exclusive time per call (msec/call)
 -i : Sort according to Inclusive time per call (total msec/call)
 -v : Sort according to Standard Deviation (excl usec)
 -r : Reverse sorting order
 -s : print only Summary profile information
 -n <num> : print only first <num> number of functions
 -f filename : specify full path and Filename without node ids
 -l : List all functions and exit
 [node numbers] : prints only info about all contexts/threads of given node numbers
>
--**-XEmacs: *shell*        (Shell: run)----Bot-----------------------------------------------
```

Routine Access and Data Display (RACY)

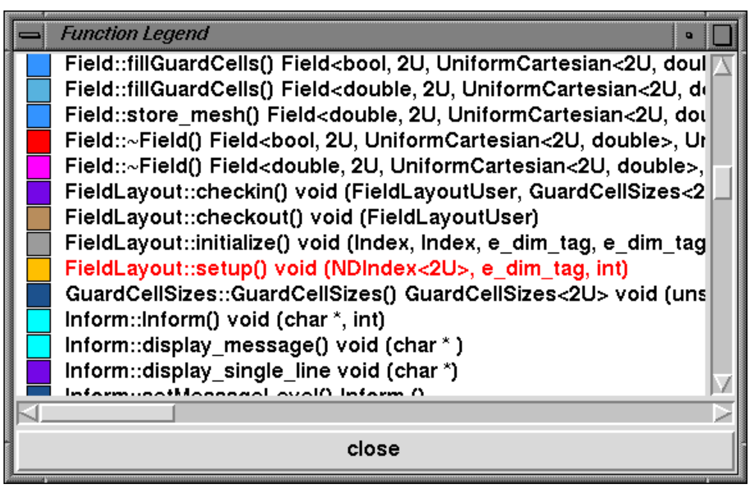


Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

RACY

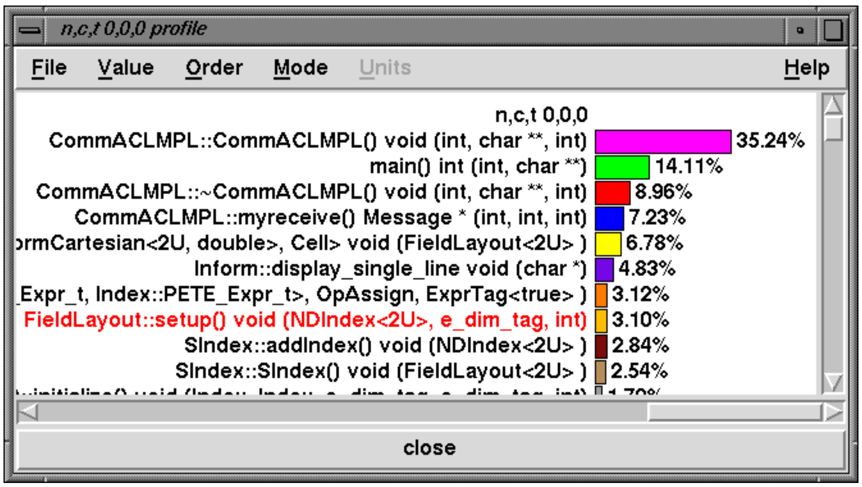


n,c,t 0,0,0 profile
File
Order
Help
%time msec total msec #call usec/call name
100.0 147 1,045 1 1045695 main() int (int, char **)
35.4 0.0336 370 1 370595 CommCreator::create() Communicate * (int, i
35.4 368 370 1 370550 CommACLMPL::CommACLMPL() void (int, char **
16.2 93 169 1 169186 CommACLMPL::~CommACLMPL() void (int, char *
7.3 75 75 50003 2 CommACLMPL::myreceive() Message * (int, int
6.8 70 71 1 71010 Field::Field() Field<double, 2U, UniformCar
6.5 18 68 1 68441 FieldLayout::initialize() void (Index, Inde
5.0 0.74 52 13 4042 Inform::setMessageLevel() Inform ()
5.0 1 51 13 3985 Inform::display_message() void (char *)
4.8 50 50 173 292 Inform::display_single_line void (char *)
4.8 32 49 1 49698 FieldLayout::setup() void (NDIndex<2U>, e_d
3.7 0.036 38 1 38891 assign() void (IndexedBareField<double, 2U,
3.7 32 38 1 38287 assign[IndexedBareField]-vnodeloop void (I
2.9 29 30 3 10041 SIndex::addIndex() void (NDIndex<2U>)
2.6 26 26 2 13340 SIndex::SIndex() void (FieldLayout<2U>)
1.7 16 17 1 17964 assign() void (SIndex<2U>, PETE_TBTree<OpLT
1.6 15 17 1 17217 assign() void (SIndex<2U>, PETE_TBTree<OpAr
1.5 16 16 1 16074 vmap::insert() pair<vector<vmap<GuardCellSi
1.4 14 15 1 15061 BareField::setup() BareField<double, 2U> vo
close

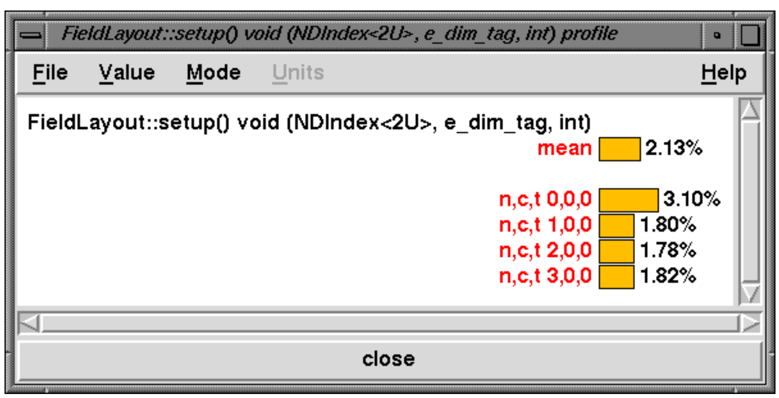


Function Legend
Field::fillGuardCells() Field<bool, 2U, UniformCartesian<2U, doul
Field::fillGuardCells() Field<double, 2U, UniformCartesian<2U, d
Field::store_mesh() Field<double, 2U, UniformCartesian<2U, dou
Field::~Field() Field<bool, 2U, UniformCartesian<2U, double>, Ur
Field::~Field() Field<double, 2U, UniformCartesian<2U, double>,
FieldLayout::checkin() void (FieldLayoutUser, GuardCellSizes<2
FieldLayout::checkout() void (FieldLayoutUser)
FieldLayout::initialize() void (Index, Index, e_dim_tag, e_dim_tag
FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int)
GuardCellSizes::GuardCellSizes() GuardCellSizes<2U> void (uns
Inform::Inform() void (char *, int)
Inform::display_message() void (char *)
Inform::display_single_line void (char *)
close

RACY



n,c,t 0,0,0 profile
File
Value
Order
Mode
Units
Help
n,c,t 0,0,0
CommACLMPL::CommACLMPL() void (int, char **, int) 35.24%
main() int (int, char **) 14.11%
CommACLMPL::~CommACLMPL() void (int, char **, int) 8.96%
CommACLMPL::myreceive() Message * (int, int, int) 7.23%
ormCartesian<2U, double>, Cell> void (FieldLayout<2U>) 6.78%
Inform::display_single_line void (char *) 4.83%
_Expr_t, Index::PETE_Expr_t>, OpAssign, ExprTag<true>) 3.12%
FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int) 3.10%
SIndex::addIndex() void (NDIndex<2U>) 2.84%
SIndex::SIndex() void (FieldLayout<2U>) 2.54%
close



FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int) profile
File
Value
Mode
Units
Help
FieldLayout::setup() void (NDIndex<2U>, e_dim_tag, int)
mean 2.13%
n,c,t 0,0,0 3.10%
n,c,t 1,0,0 1.80%
n,c,t 2,0,0 1.78%
n,c,t 3,0,0 1.82%
close

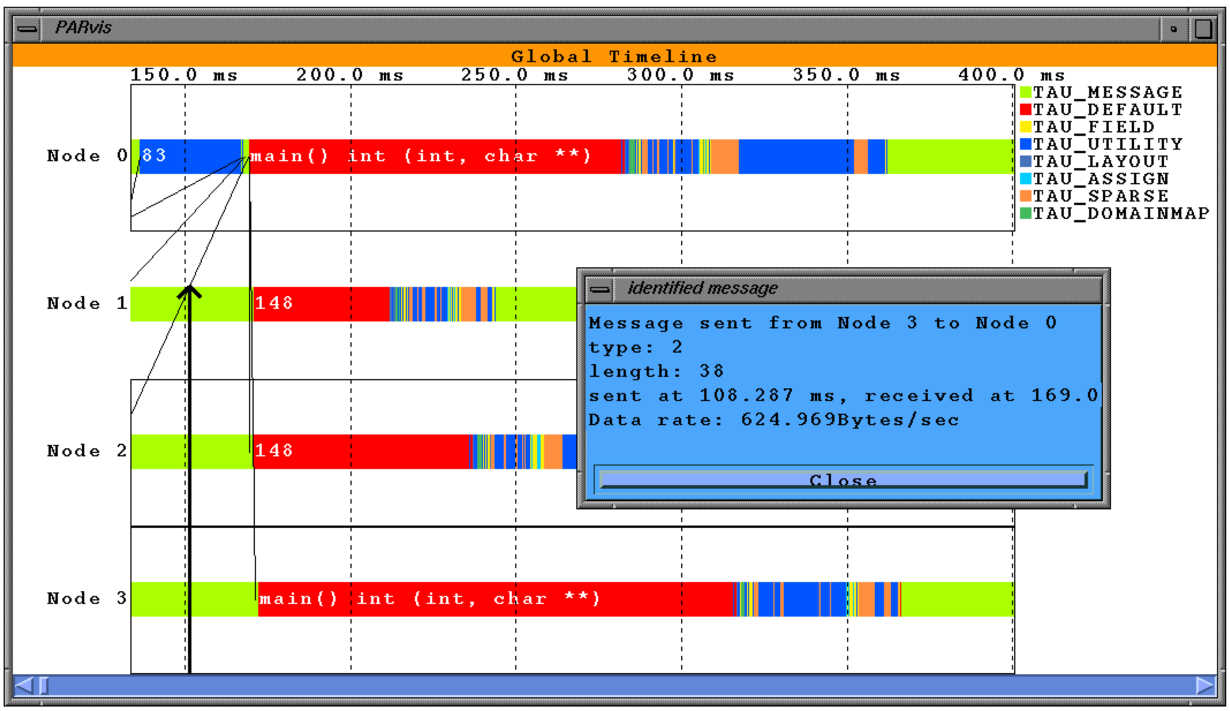
TAU Tracing

- Tracing library can be used with or without profiling.
- Reuses TAU profiling instrumentation for function entry/exit.
- Shows details of message passing and function entry/exit.
- Converts traces from TAU to:

    1. ASCII text format
    2. VAMPIR trace format
    3. ALOG (Upshot) trace format
    4. SDDF (Pablo) trace format

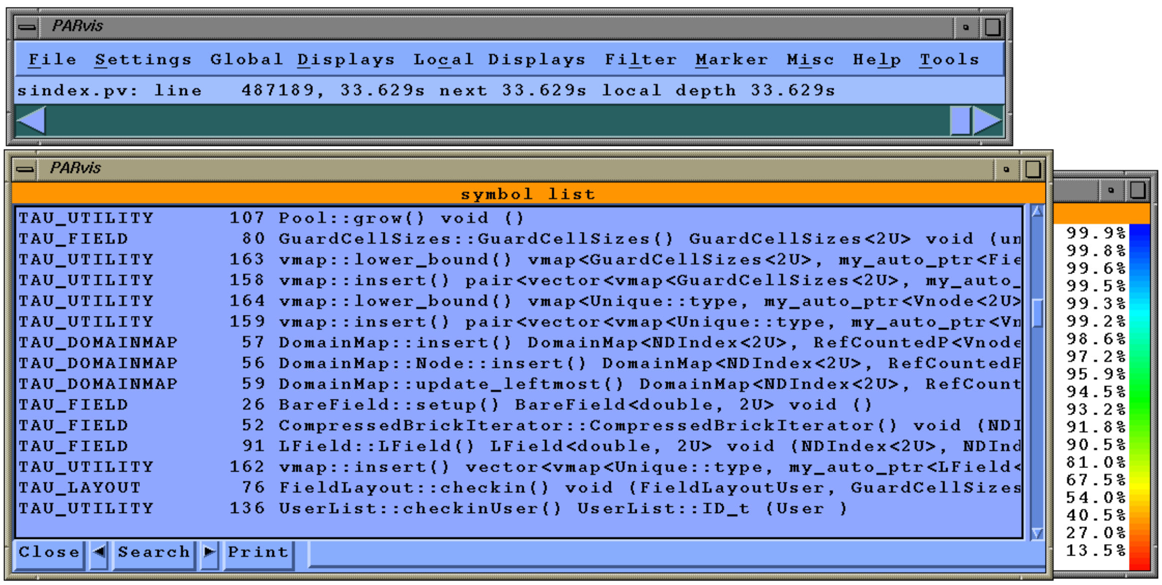
Vampir Trace Visualization Tool (Pallas, KFA Jülich, Germany)

http://www.pallas.de



Trace of a POOMA sparse index application displayed in VAMPIR

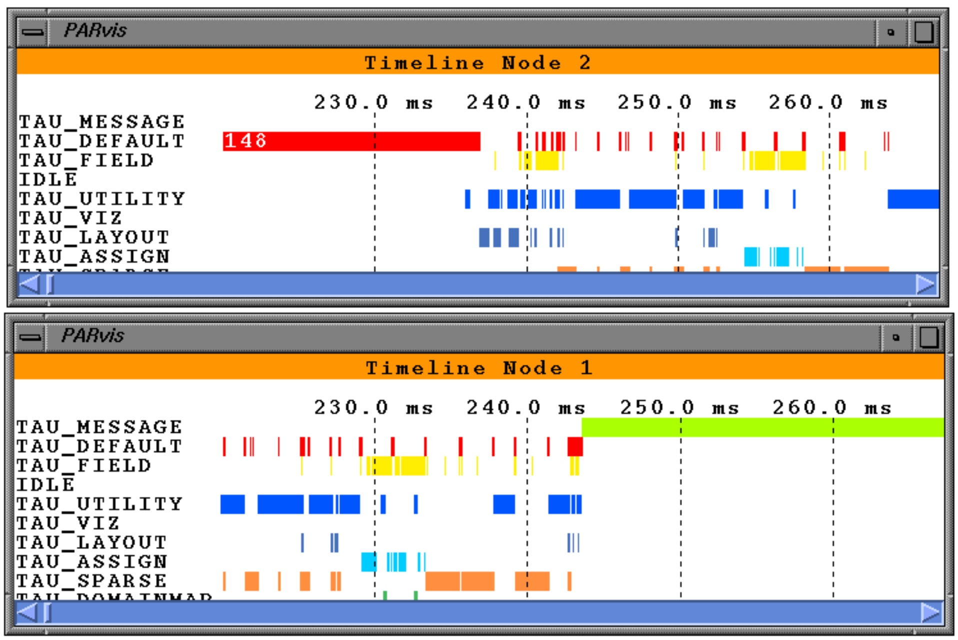
Vampir and TAU



VAMPIR displays templates in C++
using TAU instrumentation

Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

VAMPIR and TAU



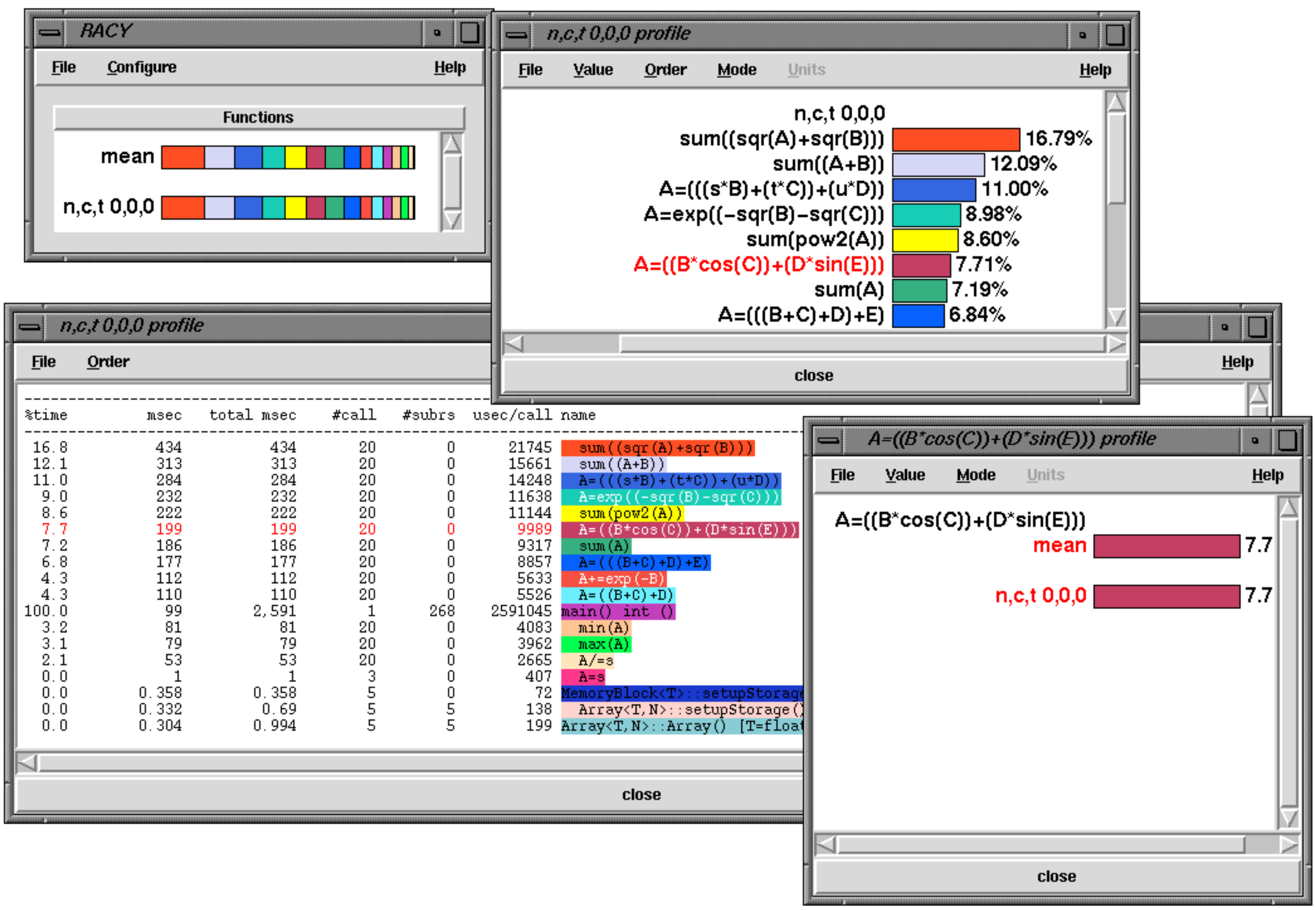
Comparison of two nodes for different TAU groups in VAMPIR

Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

Profiling Expression Templates

(Todd Veldhuizen, Lawrence Berkeley National Laboratory)

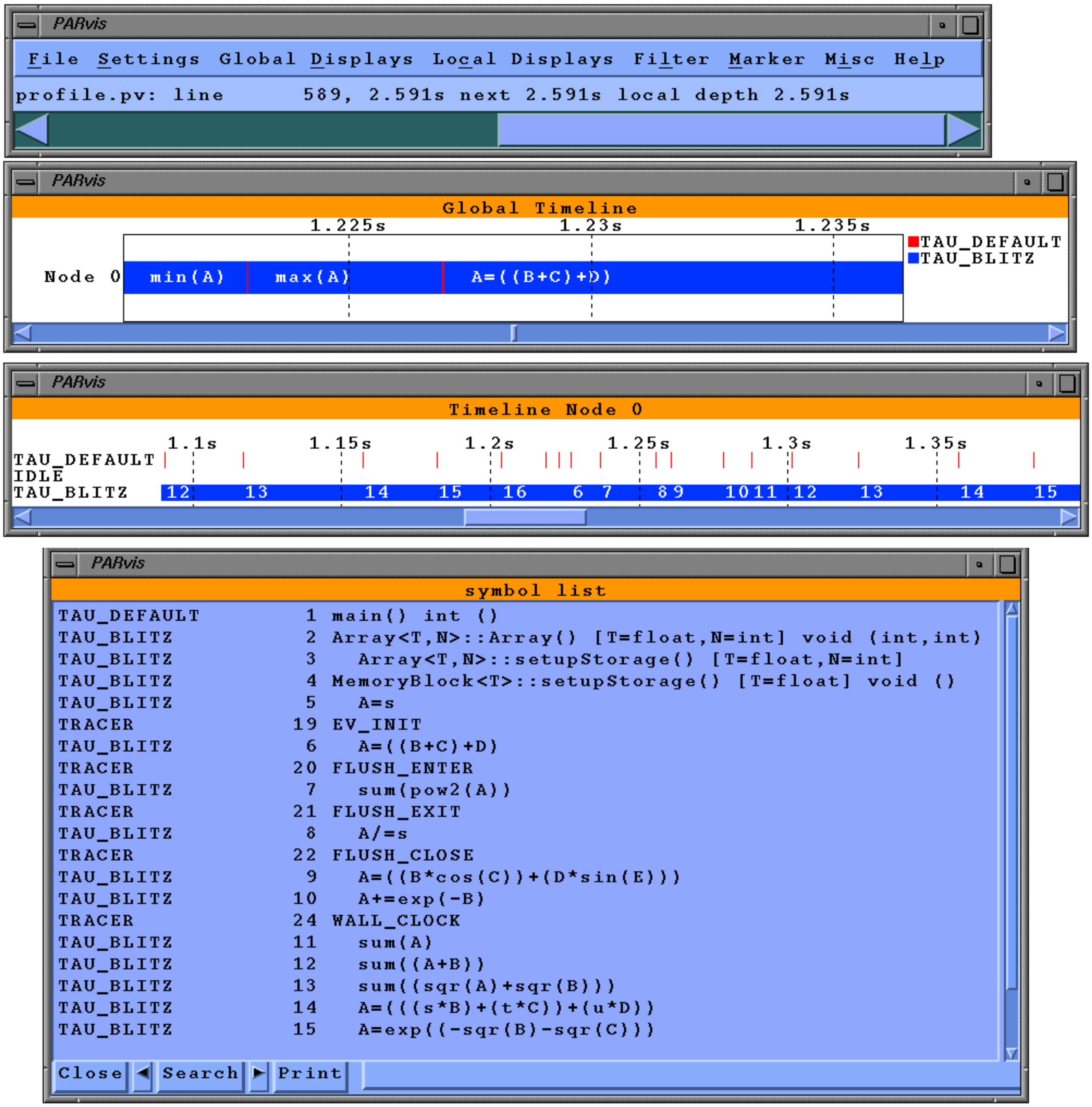
- Blitz++ numerical library
- Expression templates



Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

Tracing Expression Templates

(Todd Veldhuizen, Lawrence Berkeley National Laboratory)



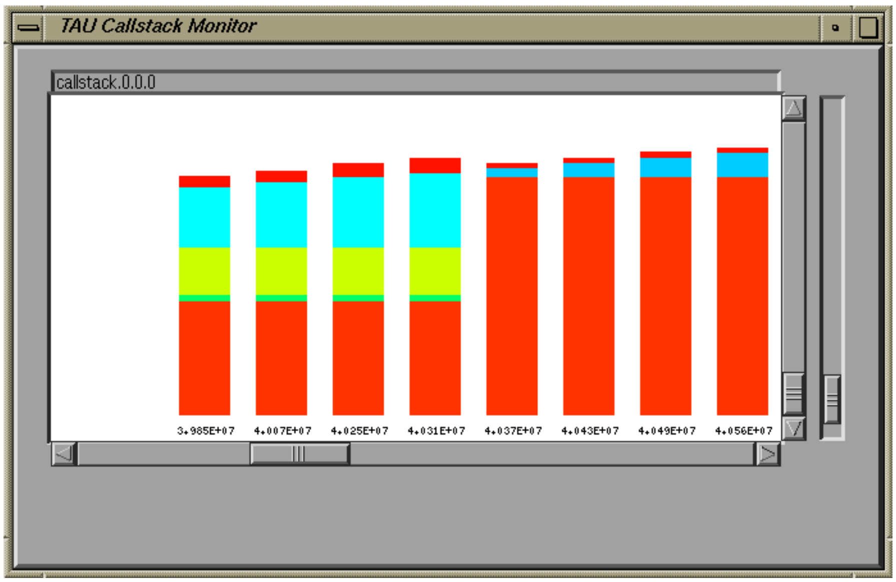
Tuning and Analysis Utilities
http://www.acl.lanl.gov/tau

TAU: Current Focus

- KAI (Kuck and Associates) - static program database for static TAU tools (callgraph display, file and class browsers, class hierarchy browsers)
- KAI - Profiling instrumentation triggered at runtime
- EDG - Automatic insertion of profiling instrumentation
- Profiling expression templates (Todd Veldhuizen, Blitz++)
- New TAU tools: template browser, performance callstack monitor, performance callstack trace
- VAMPIR - binary trace data generation, new tools to merge and convert to binary trace format
- Support for other trace visualization tools (VAMPIR, ALOG, SDDF formats supported)

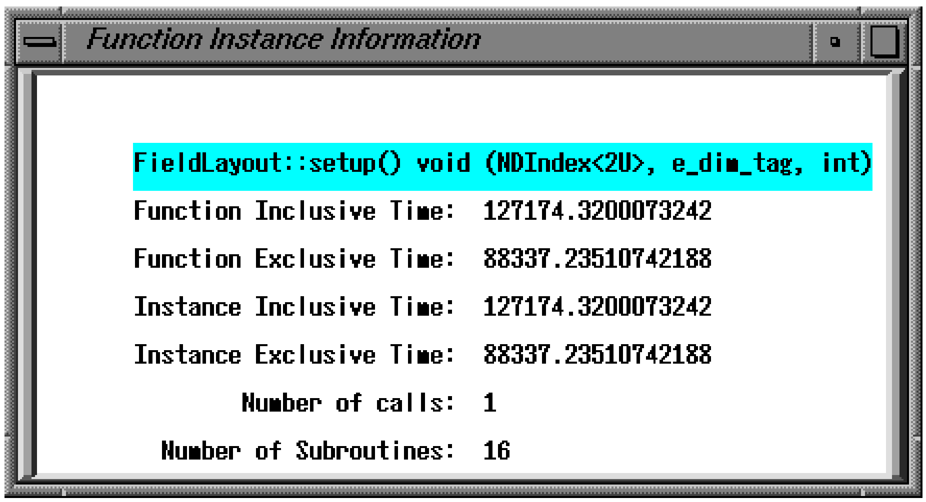
Performance Callstack Trace

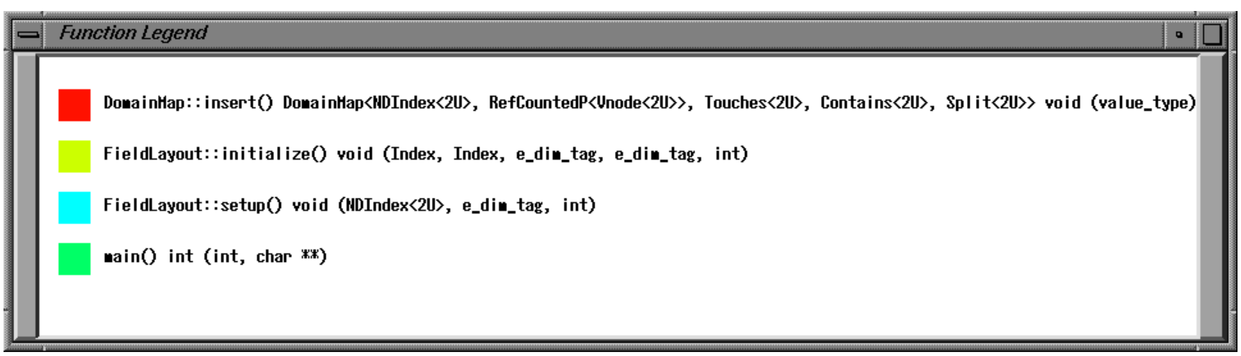
(under implementation, prototype released in ver 2.2)



Performance Callstack Trace

(under implementation)





TAU Status

- TAU profiling and tracing instrumentation API defined and implemented and available for downloading at: **http://www.acl.lanl.gov/tau**
- Profiling deliverable: profiling library, profile data visualization tools - RACY, pprof
- Trace deliverable: tracing library, trace merging and conversion tools
- Support for trace visualization with VAMPIR (PALLAS, KFA Jülich)
- Instrumentation of SciTL components:
  - POOMA (Parallel Object Oriented Methods and Applns)
  - Tecolote
  - ACLMPL (ACL Message Passing Library)
  - A++/P++ (Array library and Parallel Array Library)
  - PADRE (Parallel Asynchronous Data Routing Env.)
  - ACLVIS (ACL Visualization Library)
  - PETE (Portable Expression Template Engine)
  - MC++ (Montecarlo Simulation)

  among others...

Conclusion

- TAU provides the building blocks for implementing different performance profiling models and features.
- Profiling overhead depends on modules selected.
- Common API for profiling and tracing.
- Performance visualization tools.
- Tracing and profiling portable across multiple compilers (KAI, SGI CC, gnu g++), platforms(SGI Origin 2000, ASCI Red Intel Teraflop, Cray T3E, Linux PC Cluster, Solaris, HP)

Statistics	Counters	Trace
Lightweight profiling core. Selective profiling		