TAU: New Directions

Sameer Shende

Department of Computer and Information Science,

University of Oregon

sameer@cs.uoregon.edu

Overview

- Introduction to TAU (Tuning and Analysis Utilities)
  - Goals and Challenges
  - Architecture
  - Instrumentation
  - Measurement
  - Analysis
- New research directions
  - Multi-level instrumentation
  - Micro-instrumentation
  - Mapping performance data
  - Hybrid execution models
  - New measurement options
  - Proposed extensions

What is TAU?

- Performance analysis framework for scalable parallel and distributed high performance computing
- Targets a general parallel computation model [HPC++]
  - computer (SMP) <u>nodes</u>
  - shared address space <u>contexts</u>
  - <u>threads</u> of execution
- Integrated toolkit for performance instrumentation, measurement, analysis and visualization
- Portable performance profiling and tracing toolkit
- Tools associated with TAU
  - PDT (Program Database Toolkit)
  - Distributed monitoring framework
- Uses portable, open interfaces

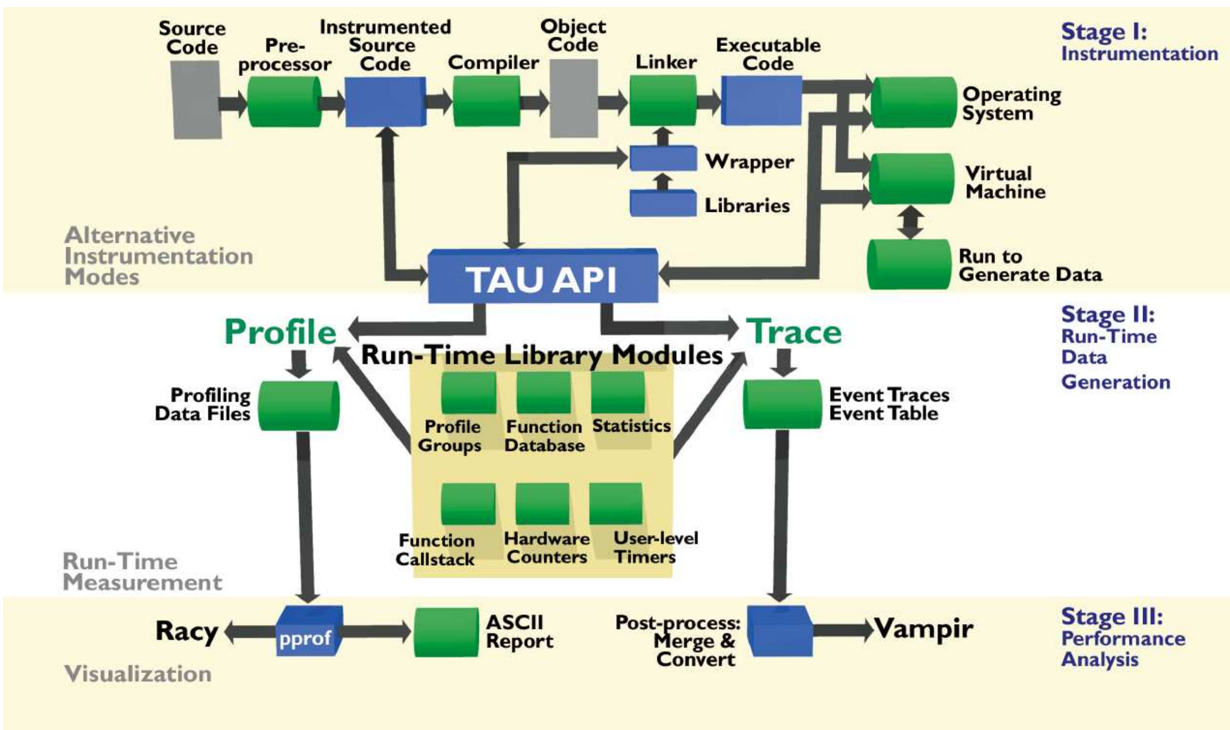
Goal and Challenges

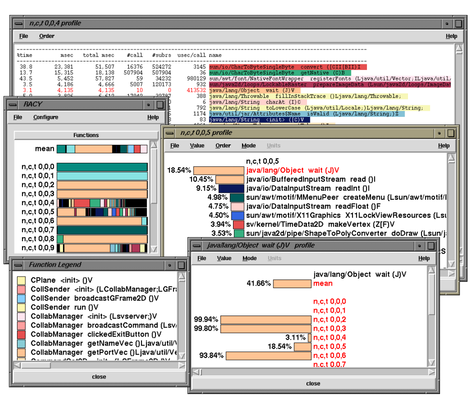
Create robust performance technology for the analysis and tuning of parallel software.

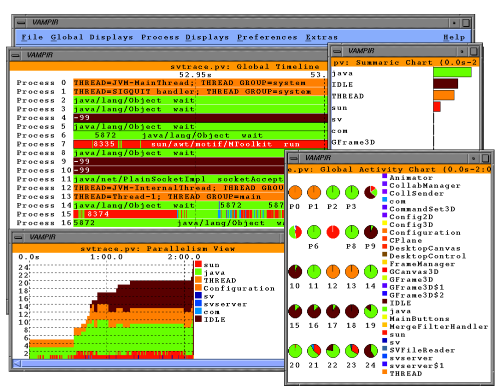
- Challenges
  - different scalable computing platforms
  - different programming languages and compilers
  - different thread models and runtime systems
  - different instrumentation strategies
  - different measurement requirements
  - common, portable framework for analysis
  - extensible, retargetable tool technology
  - complex set of requirements
  - performance experimentation

Architecture of TAU

Tuning and Analysis Utilities







http://www.acl.lanl.gov/tau

TAU Instrumentation

- Flexible, multiple instrumentation mechanisms
  - source code
    - manual (TAU API)
    - automatic using PDT (tau_instrumentor)
  - object code
    - pre-instrumented libraries (ACLMPL)
    - statically linked: MPI Profiling Interface (libTauMpi.a)
  - executable code
    - dynamic instrumentation using DyninstAPI (tau_run)
  - virtual machine
    - Java instrumentation using JVMPI and TAU shared object dynamically loaded in the JVM
- Ability to combine multiple instrumentation options!

TAU Measurement

- Configuration options
  - High resolution wall clock time [PAPI, SGITIMERS]
  - CPU time (user+system)
  - Process virtual time (user) [PAPI]
  - Hardware performance counters
    (primary/sec. data cache misses, etc.) [PAPI, PCL]

- PAPI (Performance API) provides low-overhead access to counters and timers (U. Tenn. Knoxville) (http://icl.cs.utk.edu/projects/papi/)

TAU Measurement

- Profiling
  - aggregate summaries of performance metrics
  - function-level, block-level, statement-level
  - supports user-defined events
  - measured process timing (as opposed to sampling)
  - statistics (standard deviation)
- Tracing
  - event logs
  - same instrumentation for both profiling and tracing
  - inter-process communication events
  - trace merge and conversion
  - output to Vampir trace format

TAU Analysis

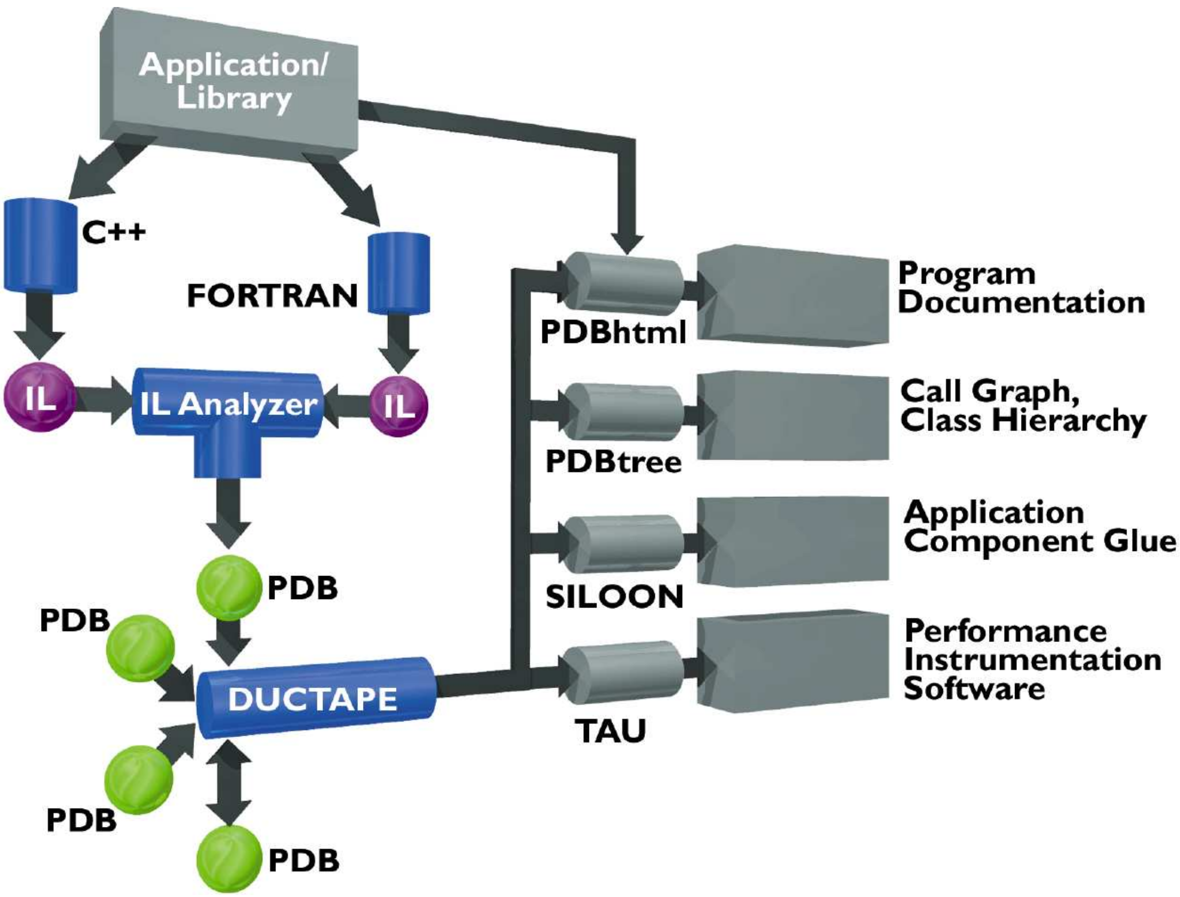
- Profile analysis
  - pprof
    - parallel profiler with text based display
  - racy
    - graphical interface to pprof
- Trace analysis
  - Vampir
    - trace analysis and visualization tool (Pallas GmbH)

TAU Status

Available for download now (ver. TAU 2.8b10)

- Languages
  - C++, C, F90, Java.
  - HPF, pC++, HPC++, ZPL
- Platforms
  - SGI, IBM, SUN, HP, Compaq, Alpha/Pentium Linux clusters, PC Windows, Intel ASCI Red, Cray T3E
- Thread libraries
  - pthread, OpenMP, Java, Windows, SMARTS, Tulip
- Communication libraries
  - MPI, PVM, ACLMPL, Nexus, Tulip
- Compilers
  - KAI's KCC & Guide, PGI, SUN, IBM, SGI, GNU, MS, Fujitsu, Cray
- *550 registered downloads (not users)*

Program Database Toolkit (PDT)



http://www.acl.lanl.gov/tau

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 (EDG, Mutek)
  - portable IL analyzer, database format, and access API
  - open software approach for tool development
- Target and integrate multiple source languages
- C++ version available. F90 version to be released soon.
- **http://www.acl.lanl.gov/pdtoolkit**

New Research Directions

- Multi-level instrumentation
- Micro-instrumentation
- Mapping performance data
- Hybrid execution models
- New measurement options
- Proposed extensions

Multi-level instrumentation

- Combine instrumentation APIs
  - executable (DyninstAPI) + source code
  - virtual machine (JVMPI) + library level (MPI Wrapper)
  - automated source code (PDT) + library level (MPI)
- Better coverage and level of abstraction

Micro-instrumentation

- Crossing "routine" boundaries for instrumentation
- Basic block, statement level probes
- Problems:
  - Optimizations may be affected
  - How do we profile in the presence of code transforming optimizations?
  - Source to source translations (ZPL+TAU)
  - Compiler transformations
  - Instrumentation using mapping tables after optimizations have been applied
  - How should compilers and performance tools "share" mapping information?
  - New mapping models for performance data

Mapping Performance Data

- Traditional mapping scenarios [Irvin/Miller, Adve et.al]
  - one-one (straightforward)
  - one-many (aggregate costs)
  - many-one (amortize/aggregate costs)
  - many-many (aggregate)
- Real life situations have some more information (optimizations)
- How can we use that to refine mapping models?

TAU Mapping of Asynchronous Execution

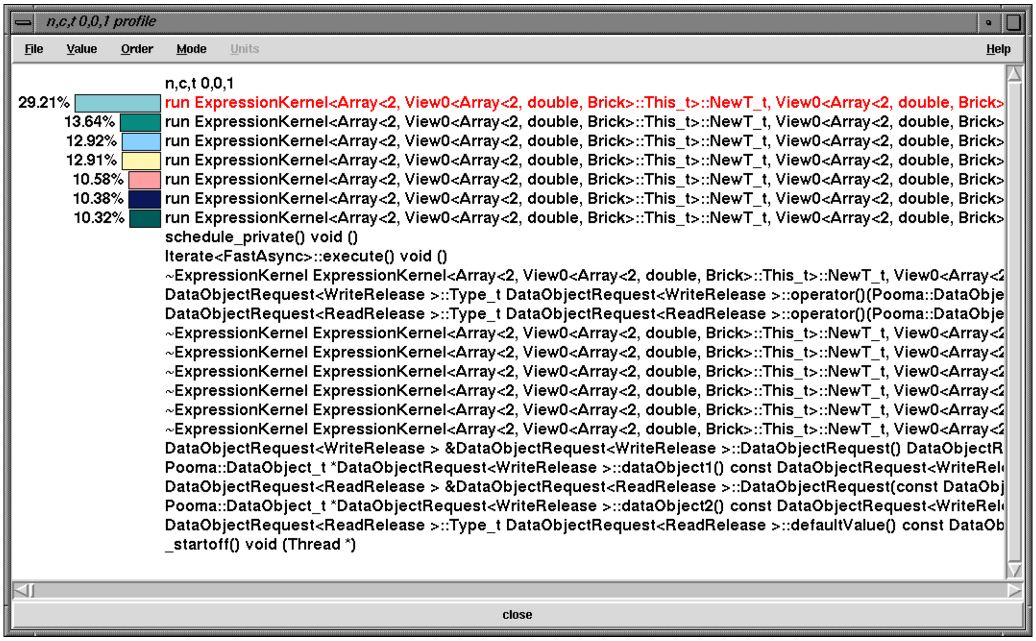
- POOMA II and SMARTS

```
#include "Pooma/Arrays.h"

#include <iostream.h>

// The size of each side of the domain.
const int N = 3*1024;

int
main(
    int                 argc,           // argument count
    char *              argv[]          // argument list
){
    // Initialize Pooma.
    Pooma::initialize(argc, argv);

    // The array we'll be solving for
    Array<2> A(N, N), B(N,N), C(N,N), D(N,N), E(N,N);

    // Must block since we're doing some scalar code (see Tutorial 4).
    Pooma::blockAndEvaluate();

    A = 1.0;
    B = 2.0;
    C = 3.0;
    D = 4.0;
    E = 5.0;

    A = B + C + D;
    C = E - A + 2.0 * B;
    D = A + C;
    C = D + A - B;
    A = 2.0 * D + E ;
    E = 1.5 * B - A ;

    Pooma::blockAndEvaluate();

    cout << "D(1,1) = " << D(1,1) << endl;
    cout << "D(9,9) = " << D(9,9) << endl;

    // Clean up Pooma and report success.
    Pooma::finalize();
    return 0;
}
```

Mapping Asynchronous Executions

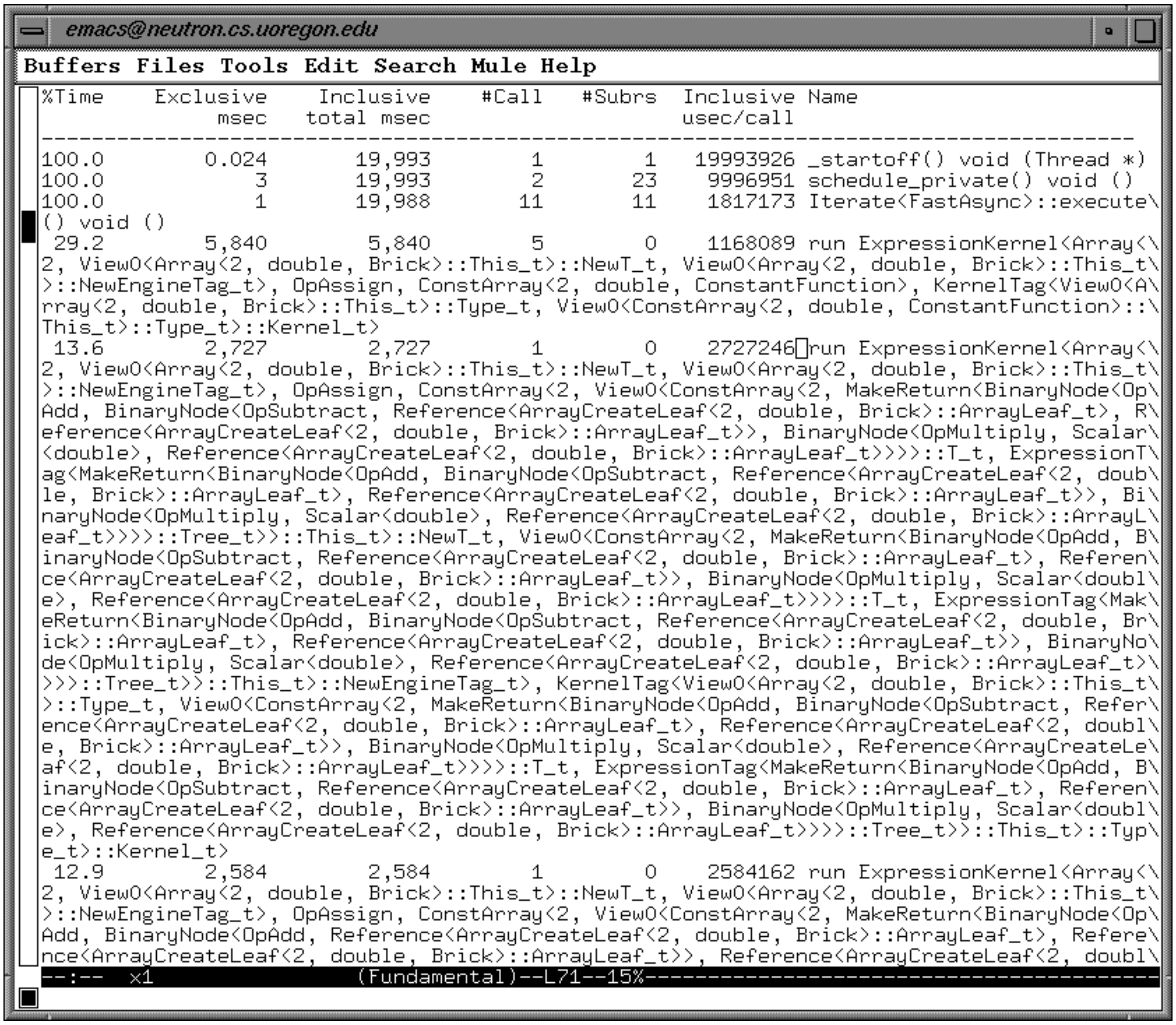
- All Array statements (composed into iterates) map to the ExpressionKernel class (many - one mapping)
- Each Iterate has its own object
- Profiling at the level of iterate objects reveals statement level profile
- Mapping asynchronous performance data to the array statements

POOMA+SMARTS: Without Mappings



- Expression Templates produce long names (embedding the parse tree of the expression in the expression evaluation template)

Without Mappings



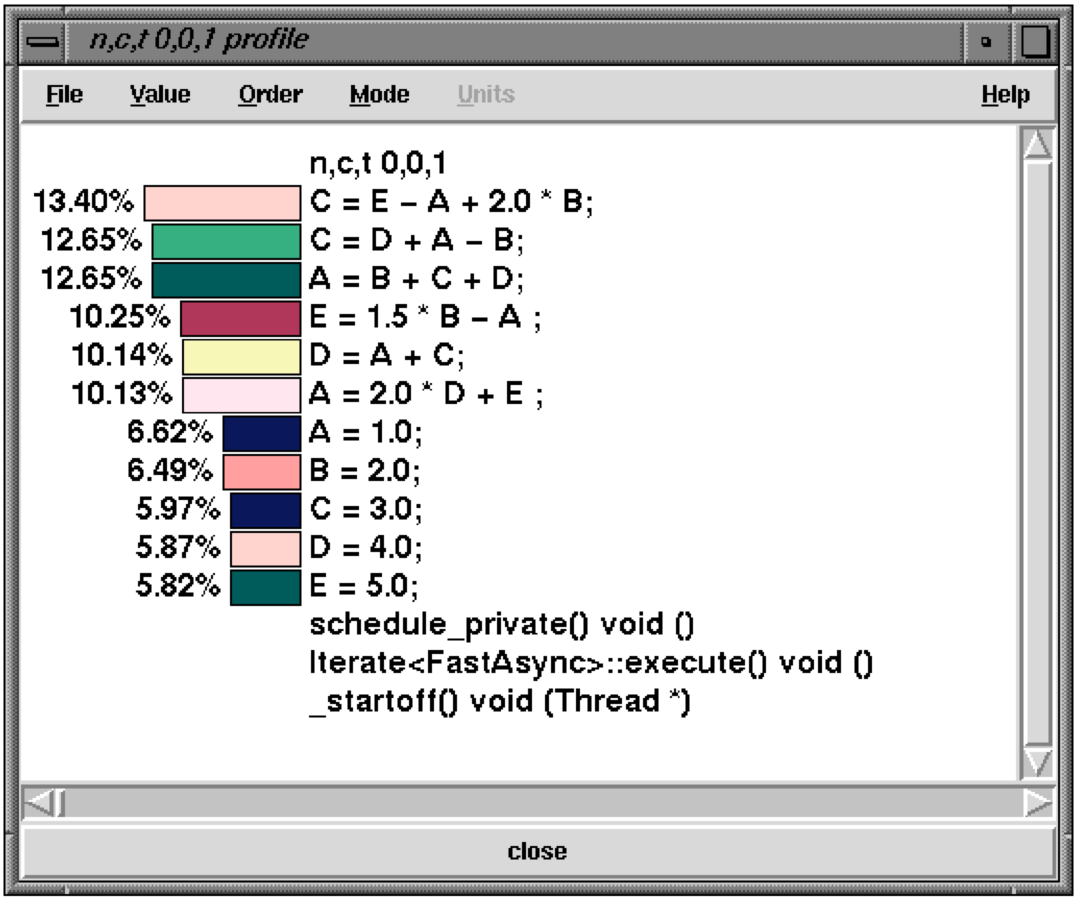
```
emacs@neutron.cs.uoregon.edu
Buffers Files Tools Edit Search Mule Help
%Time    Exclusive    Inclusive     #Call    #Subrs  Inclusive Name
              msec   total msec                      usec/call
-----------------------------------------------------------------------------------
100.0        0.024       19,993         1         1   19993926 _startoff() void (Thread *)
100.0            3       19,993         2        23    9996951 schedule_private() void ()
100.0            1       19,988        11        11    1817173 Iterate<FastAsync>::execute\
() void ()
 29.2        5,840        5,840         5         0    1168089 run ExpressionKernel<Array<\
2, View0<Array<2, double, Brick>::This_t>::NewT_t, View0<Array<2, double, Brick>::This_t\
>::NewEngineTag_t>, OpAssign, ConstArray<2, double, ConstantFunction>, KernelTag<View0<A\
rray<2, double, Brick>::This_t>::Type_t, View0<ConstArray<2, double, ConstantFunction>::\
This_t>::Type_t>::Kernel_t>
 13.6        2,727        2,727         1         0    2727246 run ExpressionKernel<Array<\
2, View0<Array<2, double, Brick>::This_t>::NewT_t, View0<Array<2, double, Brick>::This_t\
>::NewEngineTag_t>, OpAssign, ConstArray<2, View0<ConstArray<2, MakeReturn<BinaryNode<Op\
Add, BinaryNode<OpSubtract, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>, R\
eference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, BinaryNode<OpMultiply, Scalar\
<double>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>>>::T_t, ExpressionT\
ag<MakeReturn<BinaryNode<OpAdd, BinaryNode<OpSubtract, Reference<ArrayCreateLeaf<2, doub\
le, Brick>::ArrayLeaf_t>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, Bi\
naryNode<OpMultiply, Scalar<double>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayL\
eaf_t>>>>::Tree_t>>::This_t>::NewT_t, View0<ConstArray<2, MakeReturn<BinaryNode<OpAdd, B\
inaryNode<OpSubtract, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>, Referen\
ce<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, BinaryNode<OpMultiply, Scalar<doubl\
e>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>>>::T_t, ExpressionTag<Mak\
eReturn<BinaryNode<OpAdd, BinaryNode<OpSubtract, Reference<ArrayCreateLeaf<2, double, Br\
ick>::ArrayLeaf_t>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, BinaryNo\
de<OpMultiply, Scalar<double>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>\
>>>::Tree_t>>::This_t>::NewEngineTag_t>, KernelTag<View0<Array<2, double, Brick>::This_t\
>::Type_t, View0<ConstArray<2, MakeReturn<BinaryNode<OpAdd, BinaryNode<OpSubtract, Refer\
ence<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>, Reference<ArrayCreateLeaf<2, doubl\
e, Brick>::ArrayLeaf_t>>, BinaryNode<OpMultiply, Scalar<double>, Reference<ArrayCreateLe\
af<2, double, Brick>::ArrayLeaf_t>>>>::T_t, ExpressionTag<MakeReturn<BinaryNode<OpAdd, B\
inaryNode<OpSubtract, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>, Referen\
ce<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, BinaryNode<OpMultiply, Scalar<doubl\
e>, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>>>::Tree_t>>::This_t>::Typ\
e_t>::Kernel_t>
 12.9        2,584        2,584         1         0    2584162 run ExpressionKernel<Array<\
2, View0<Array<2, double, Brick>::This_t>::NewT_t, View0<Array<2, double, Brick>::This_t\
>::NewEngineTag_t>, OpAssign, ConstArray<2, View0<ConstArray<2, MakeReturn<BinaryNode<Op\
Add, BinaryNode<OpAdd, Reference<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>, Refere\
nce<ArrayCreateLeaf<2, double, Brick>::ArrayLeaf_t>>, Reference<ArrayCreateLeaf<2, doubl\
--:--  x1              (Fundamental)--L71--15%--------------------------------------
```

- “Array=constant” expressions take 29.2 %

  (lumped together for A=1, B=2, C=3, D=4, E=5)

- “C=E-A+2*B” is incomprehensible (big expression)

Mapping Performance Data using TAU

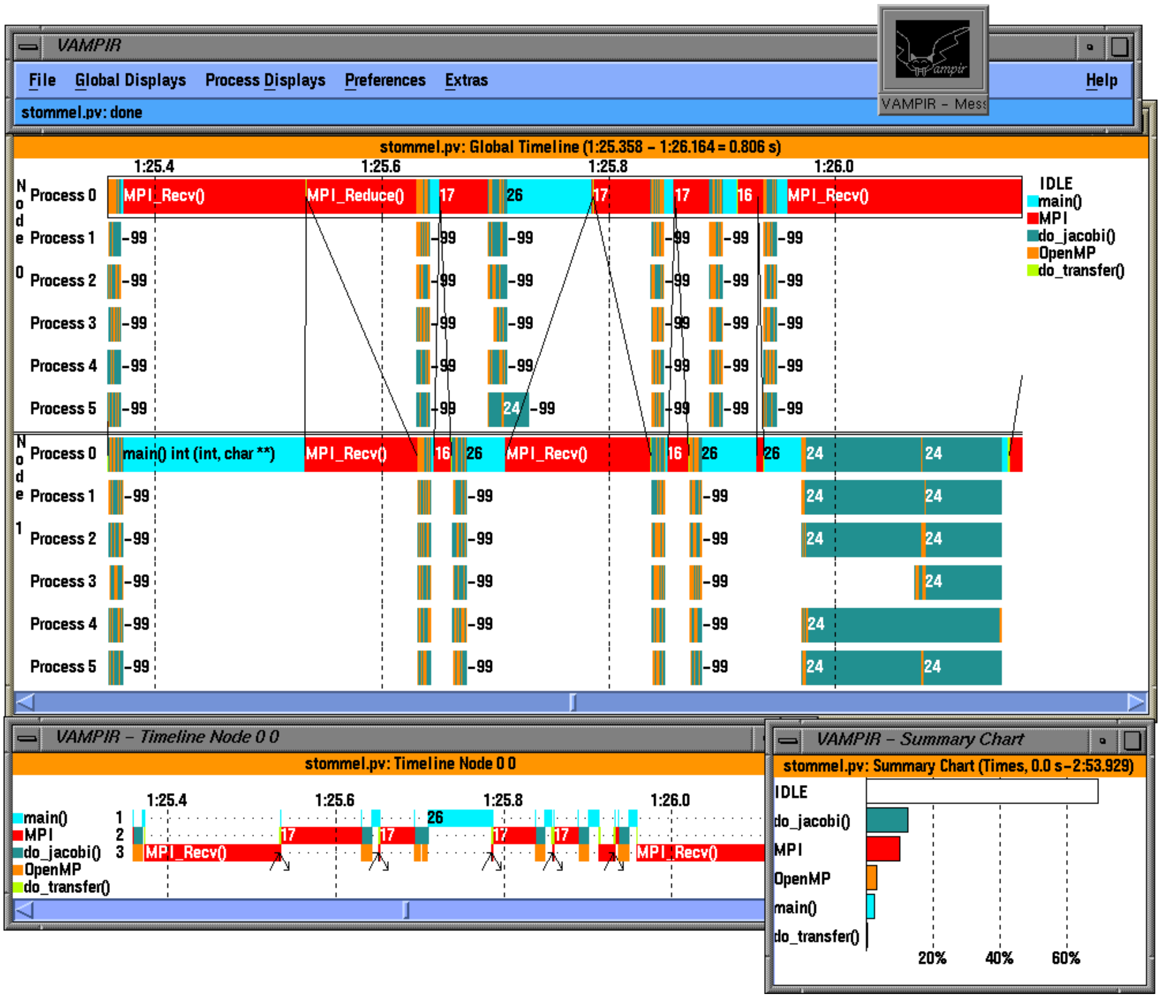


- ❑ Time spent in each statement (A=1, B=2, C=3, D=4...)
- ❑ Works in presence of asynchronous execution
- ❑ Across different “compute” threads
- ❑ Closing the semantic-gap!

Hybrid execution models

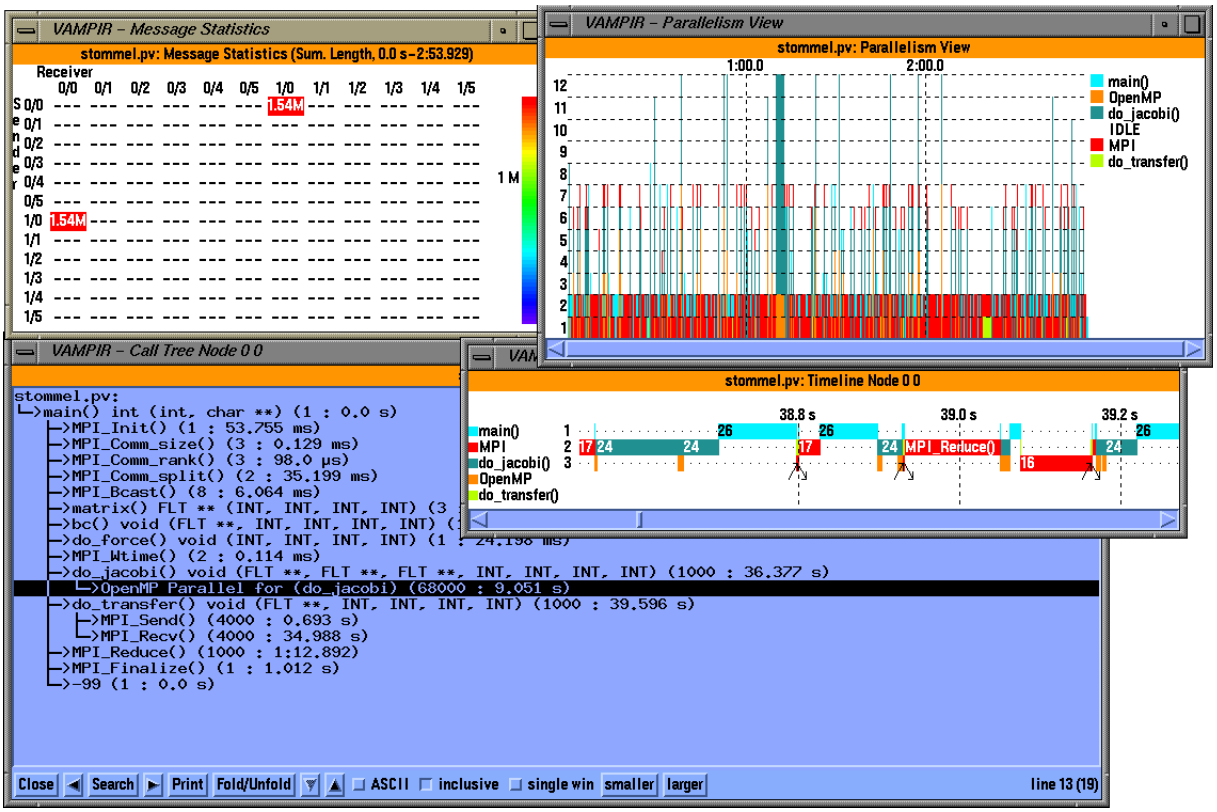
- Mixed model programming merge execution models
- Threads + MPI (pthreads+MPI, OpenMPI, mpiJava)
- Problems:
  - Incomplete information
  - MPI doesn’t know about threads, threads don’t know which node they’re running on
  - TAU allows different modules to “advertise” all information they know and “share” it
  - Sender doesn’t know which thread in the receiver received the message and vice versa
  - Matching sends and receives during post-processing allows for execution model “corrections”
- Problems for message passing and shared memory programs are well understood in isolation
- When models are mixed, we encounter different kinds of problems

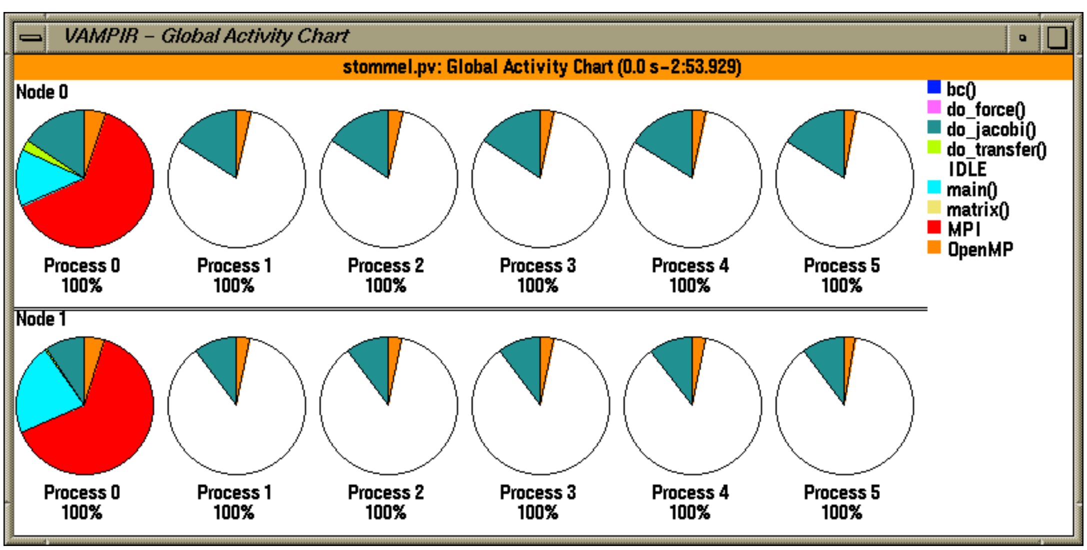
TAU supports OpenMP+MPI



- Vampir [http://www.pallas.de] is used to visualize TAU traces

Integrated Performance Views

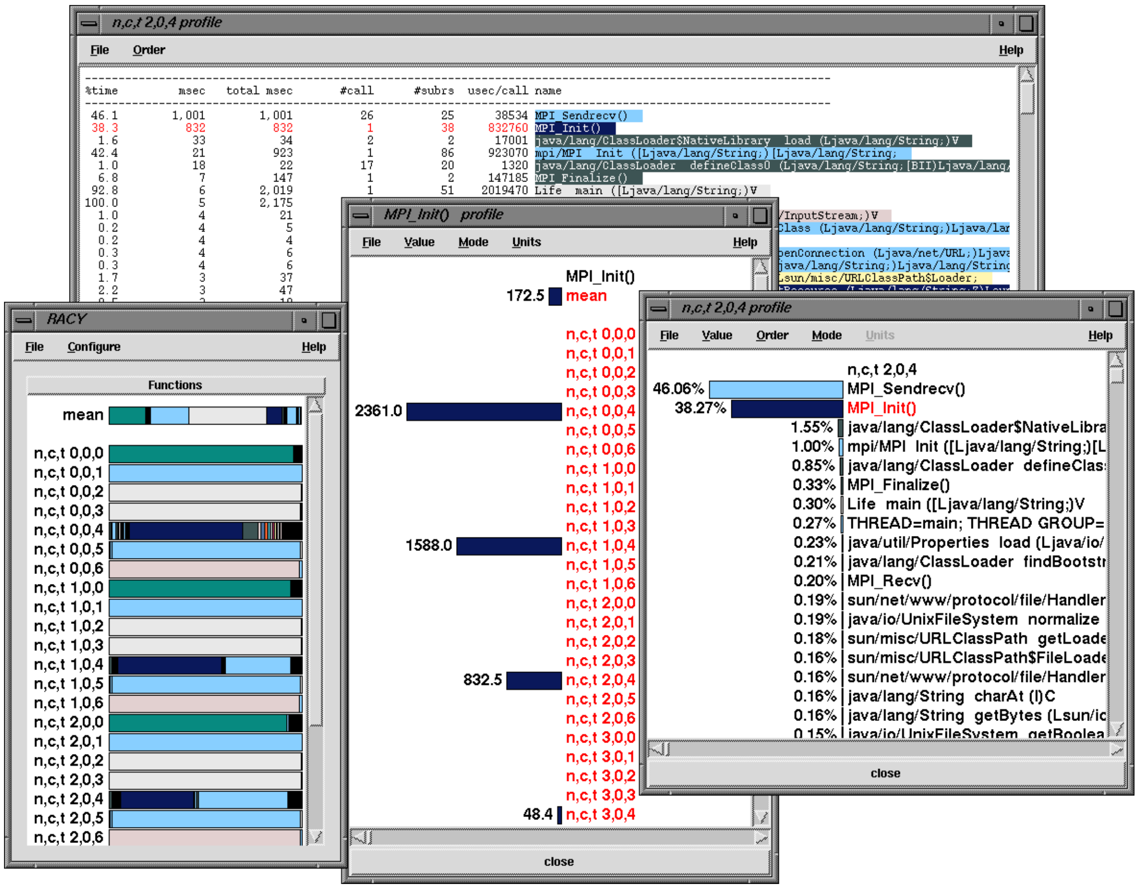




http://www.acl.lanl.gov/tau

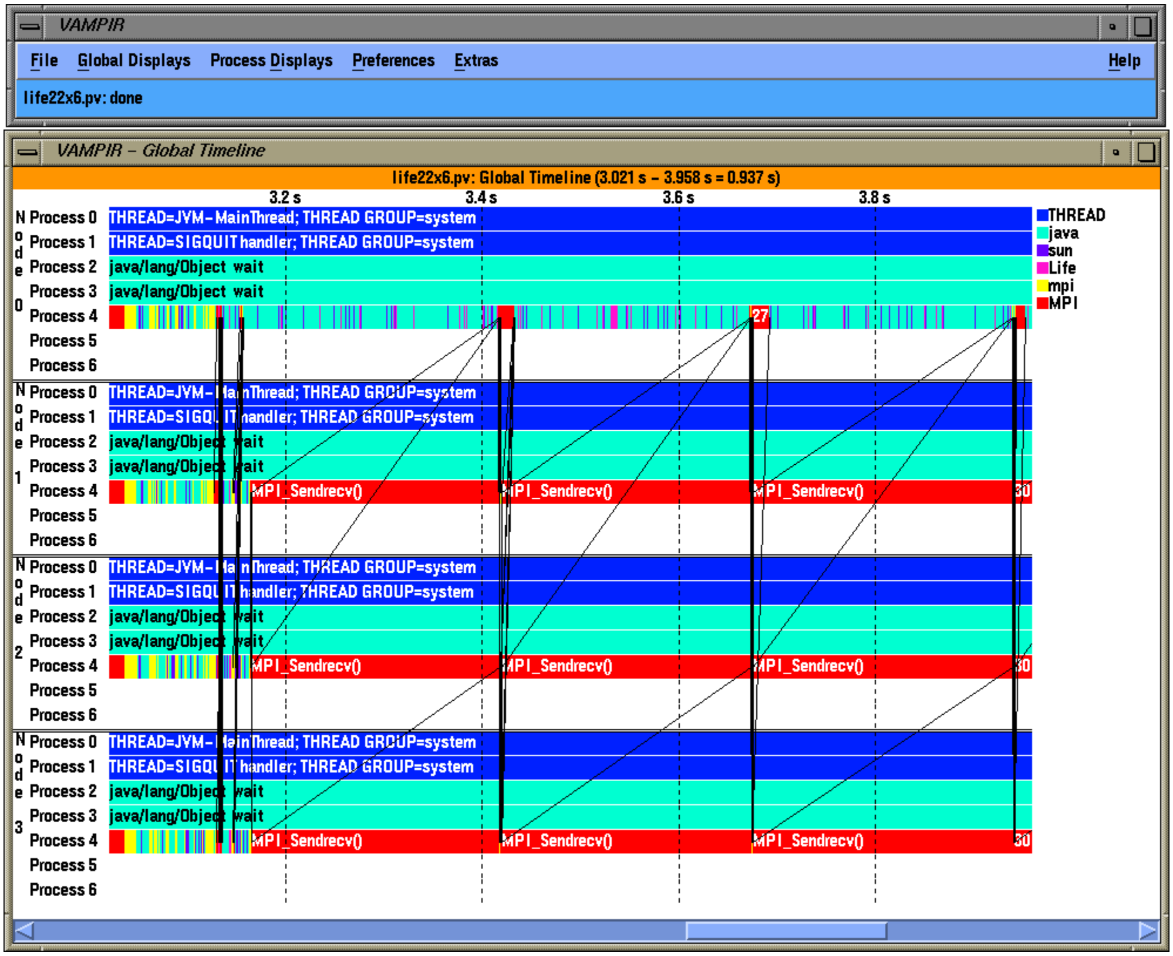
Profiling MPI+Java

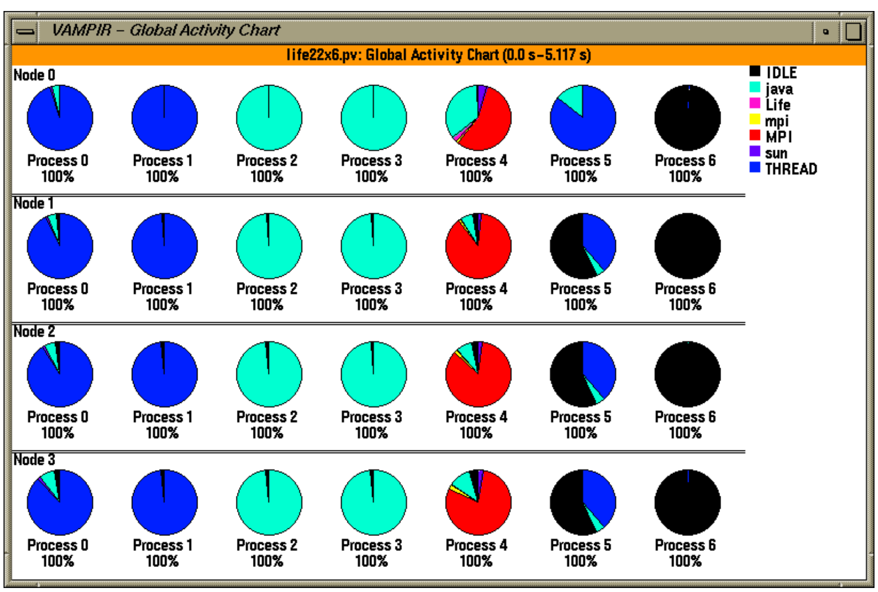
- No changes to the Java source/bytecode/JVM!



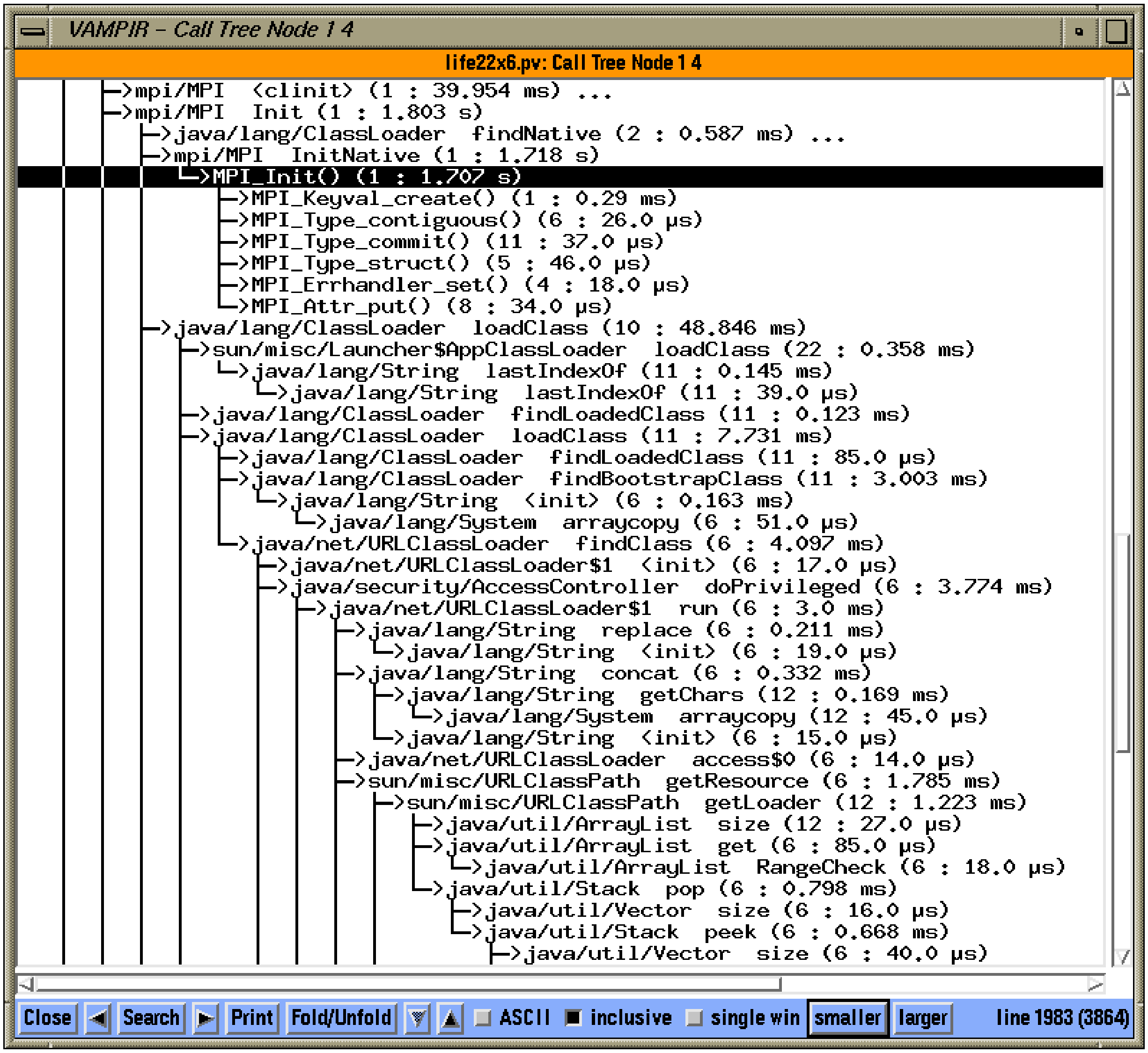
- JVMPI+MPI (mixed-model programming)

Tracing mpiJava





Dynamic CallTree



http://www.acl.lanl.gov/tau

New Measurement Options

- Fast access to wall-clock time using PAPI
    - TAU overhead measured at 830 nanosecs per entry or exit (Profiling with g++ -O2 PIII/550MHz Linux 2.4.0-test4 Kernel)
- CPU Time measurements for multi-threaded applications using Linux
- Thread-safe hardware performance counters [PAPI]
- TAU generic thread layer interfaces with PAPI for supporting thread-safe counters for all thread packages supported by TAU

Future Work & Proposed Extensions

- TAU free probe class server for SPM
- Dyninst support for MPI applications in TAU
- Cheetah runtime system
- UPS (Unified Parallel Software)
- OpenMP hooks for instrumentation
- Distributed monitoring framework
- DPCL support
- Application codes

Conclusions

- Complex parallel computing environments require robust program analysis tools
  - portable, cross-platform, multi-level, integrated
  - able to bridge and reuse existing technology
  - technology savvy
- TAU offers a performance technology framework for complex parallel computing systems
  - flexible instrumentation and measurement
  - extendable profile and trace performance analysis
  - integration with other performance technology

Acknowledgments

Tuning and Analysis Utilities

ASCI
Accelerated Strategic Computing Initiative
Lawrence Livermore

ACTS Toolkit