DEEP Tutorial: Parallel Performance Profiling at the Source Code Level.

David McNamara
Veridian PSR
September 2000

DEEP Tutorial: Topics

- Who is Veridian PSR?
- What is DEEP?
- What can DEEP do for my programs?
- DEEP program analysis demonstration.
  - Gathering the data.
  - Starting DEEP.
  - Finding Bottlenecks.
  - Improving the code.
- DEEP Debugger.
- DEEP future plans.

Veridian -- Pacific-Sierra Research High Performance Computing Group

- PSR founded 1971, part of Veridian since 1998.
- High Performance Computing group has been in Santa Monica, CA since 1979.
- Products: VAST, DEEP
- Code optimization and porting, seminars on parallel programming, consulting.
- Past and present customers include NEC, Cray, IBM, Fujitsu, Hitachi, Convex, Alliant, Intel, DARPA, Absoft, Apogee, ...

DEvelopment Environment for Parallel Programs: DEEP

- Motivation: Understanding parallel program performance and behavior is **hard**.
- Goal: Provide an integrated, cross-platform, parallel development environment *at the user's source code level*.
- Originally developed under DARPA SBIR contract.
- Status: First available in 1999.

DEEP: Target Parallel Systems

- One environment for all popular parallel programming choices.
  - Fortran and C (and mixed). C++ soon.
- Shared Memory Parallel.
  - Automatic parallelization
  - OpenMP Programming
- Distributed Memory Parallel.
  - Data Parallel Programming (HPF and Data Parallel C)
  - MPI Programming
- Scalar Performance
  - Cache misses, branch performance, ...
- Vector Performance
  - Mflops rate, vector length, ...

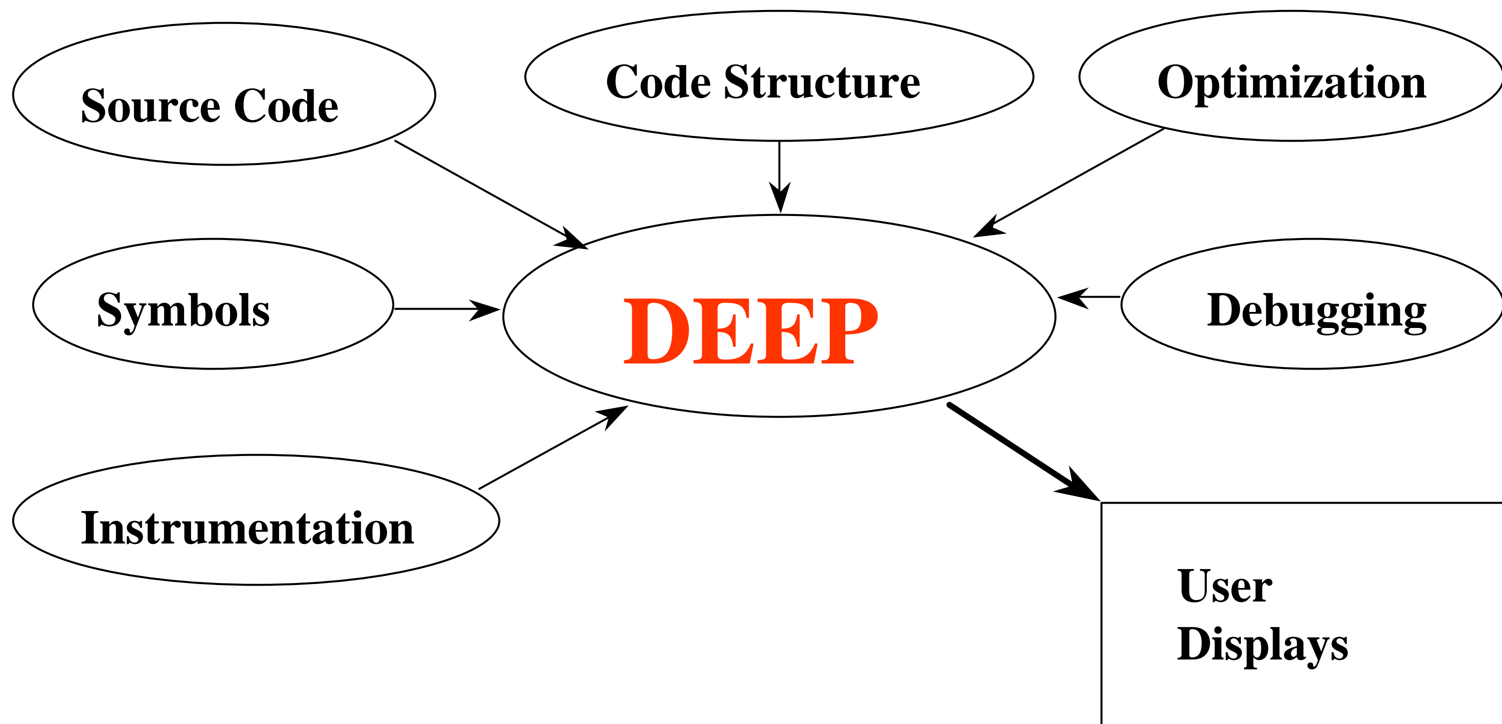
DEEP: Overview

- Supports both Unix and Windows interfaces.
  - **Unix**/Motif (Alpha, SUN/Solaris, IBM/AIX, SGI/IRIX, ...)
  - **Linux**/Motif (x86, Alpha)
  - **Windows**/NT, 95, 98 (x86)
  - Can run DEEP on non-target system (cross analyze).
- Coordinated set of tools.
  - Move quickly between tools, point and click.
  - Single master graphical user interface.
  - Framework allows organization of lots of information
    - better than many separate windows.

DEEP: Features

- Profiling
  - Where is the time spent? Which are the most important loops? How well are these loops being optimized?
  - Where are most of the messages passed? Is the processor load balanced? How much waiting is being done at synchronization points?
- Program Structure Browsing
  - How are the procedures connected to each other?
  - Where are global (parallel) variables referenced and set?
- Debugging (Under development)
  - What is the value of this distributed array?
  - What is the current status of the parallel processes?
  - What run-time data types do I have defined?

DEEP: Data gathering



DEEP: Profiling at All Levels

Distributed Memory Parallel System (load balance, communication traffic).

Message passing library, data parallel languages. Top level loops.

Shared Memory Node (synchronization bottlenecks).

OpenMP, automatic parallelization. Outer loops.

Cache Memory Levels (cache miss rates).

L3, L2, L1. Loop nests.

Vector Unit (MOP rates, % utilization)

Hand or auto vectorization. Inner loops.

Scalar Unit (instruction issue rates)

Inner loops.

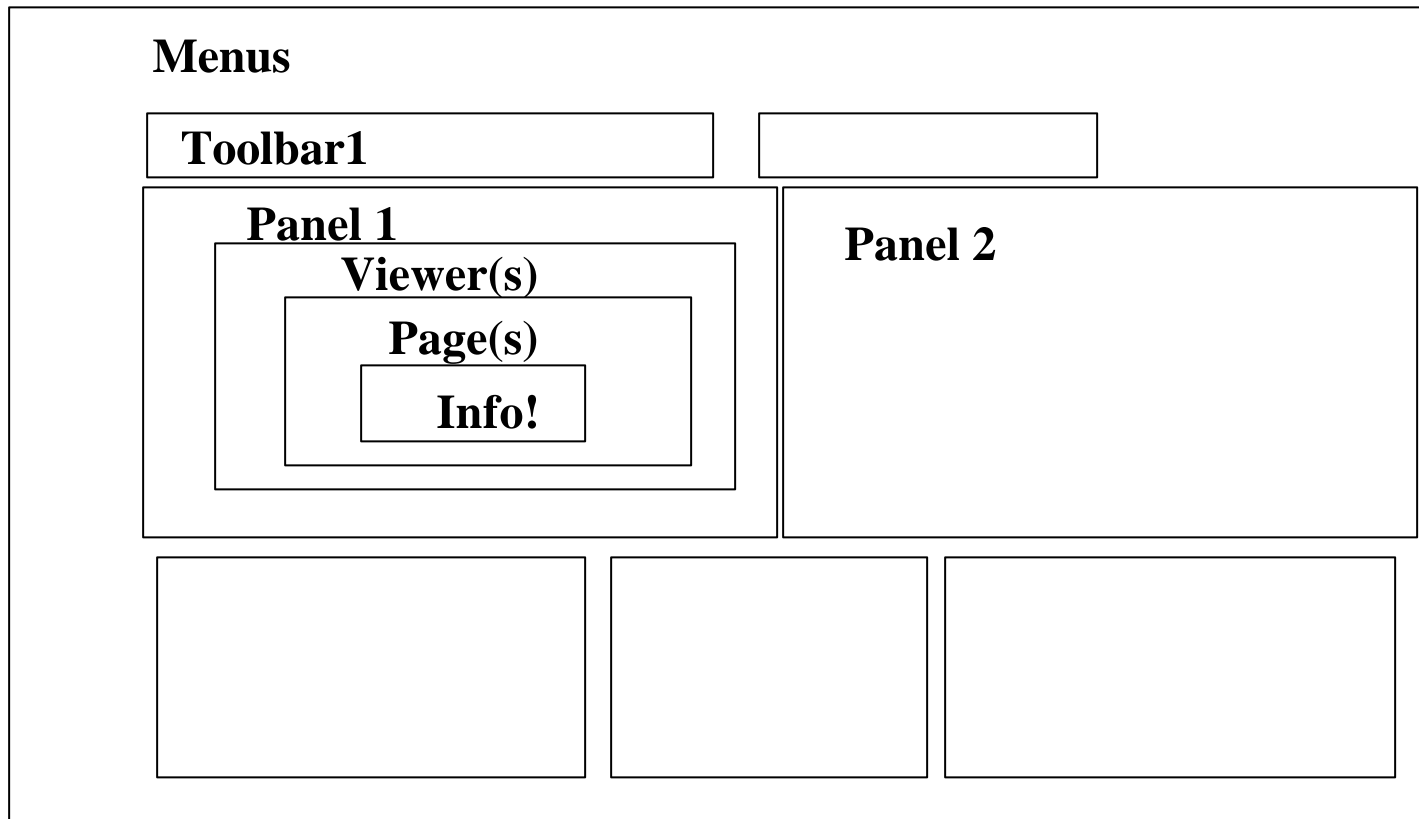
DEEP Program Analysis: Drill Down

- Start with whole program data.
- Identify "bottleneck" procedures.
  - charts, tables
- Move to details for procedure.
  - code abstraction, loop performance
- Identify run-time problems.
  - "hot" loops, color-coded
- Move to source code.
  - click on important loops
- Remove bottlenecks.
- Recompile, run again and look at performance changes. Iterate if needed.

DEEP: Drill Down

- Or, drill up?

DEEP: Framework



DEEP: MPI Support

- MPI message load balance is recorded and displayed.
- All MPI call sites in the application are profiled.
- Wait time in MPI calls is presented, so inefficient synchronizations can be spotted.
- MPI usage is summarized for each routine in the program.
- MPI operations are highlighted in the call tree and the code abstract, so you can move quickly to that spot in the code.

DEEP: PerfAPI Support

- PerfAPI (PAPI)
  - PAPI is a cross-platform library that allows access to hardware performance counters.
  - Cache misses for various levels, instruction counts, branch prediction counts, instruction stalls etc.
  - over 50 events (not all supported by all platforms).
- DEEP PAPI support
  - User can select counters to be profiled.
  - Example: show L2 cache misses and branch misprediction.
  - Counts are available at loop and routine levels.
  - Automatically graphed and included in tables.

DEEP: Gathering Data

- Run profiler on input source files.
  - This creates "static information files".
    - Compiler optimization information, program structure, symbol table.
  - Profiler also instruments the source code.
    - Adds data gathering directly to the source code.
- Run the instrumented program.
  - This creates a "run file" for each distributed processor.
  - Run files are accumulated information for a particular processor (not a huge trace file).
- Run DEEP to view complete picture.

DEEP: Instrumentation

- DEEP instruments the source code directly.
- This allows gathering data on loops, loop nests, call sites, and other language structures lower than the function level.
- Measured instrumentation overhead can vary greatly depending on the program (1% to 200%).
  - If loops turn out to have very small iteration counts, then overhead of instrumentation can swamp computation time.
- If instrumentation impact is high, then use sampling mode.
  - Samples first 1000 instances, then a few instances (about one out of 100) from then on.

Using DEEP for Program Analysis

- Compile the program with the DEEP profiling driver.
  - MPI: mpiprof myprog.f -or- mpiprof myprog.c
  - PAPI: dprof myprog.f -or- dprof myprog.c
- Run the program.
  - Do this however you normally execute your program.
    - MPI: mpirun –np 4 myprog < inputfile1.dat
    - PAPI: a.out < myinputfile.dat
- Start DEEP.
  - deep
- Provide path of source code and DEEP information.
  - File dialog box: select any source file.
- Select Run (if more than one). Click on row.
- View information. Determine Bottlenecks. Repeat.

DEEP Files

- All files (compile-time and run-time) are written to subdirectory "/deep".
  - If you compile and run in different directories, you need to combine the "/deep"s into one "/deep" prior to starting deep.
- The DEEP files are:
  - *.sif – static information files, one for each function.
  - *.sym – symbol table information. This is a binary file.
  - run.*.*.dif – runtime profile information.
    - Written by the Instrumenter profiling library at the end of execution of the application.
    - One .dif file is written for each process for each run of the program.
    - Name: *run.run_no.process_no.dif*, where run_no is the number of the run (starting with 0) and process_no is the number of the logical process (starting with zero).

DEEP: Instrumentation Options

- Use the –Wv,-z switch: mpiprof -Wv,-zfimoz myfile.c
- -z [a] [d] [f] [i] [m] [o] [z]

  -za : shortcut for -zfimo options for MPI, or –zfiop for PAPI.
- **-zd** enable all loop profiling (do not use with -zo)
- **-zf** enable external function profiling
- **-zi** enable I/O call profiling
- **-zm** enable message passing profiling for MPI versions
- **-zp** enable PAPI profiling for PAPI versions
- **-zo** enable outermost loop profiling (not with -zd)
- **-zz** enable sampling

DEEP/PAPI: Selecting Counters

- Hardware can only count a few items at a time.
- By default, DEEP/PAPI selects two PAPI events
  - Floating Point Operations and Total Cycles.
- Change before run time (after compile time) by setting a series of environment variables:
  - `setenv PROF_PAPI_1 FP_INS`
  - `setenv PROF_PAPI_2 L2_TCM`
- The general form is:
  - `setenv PROF_PAPI_x FLAG`
- where 'x' is a number from 1 to the maximum number of PAPI events that can be counted
- DEEP automatically adjusts its charts and tables to reflect your choice of PAPI counters.

DEEP/PAPI Counter Options

PAPI Flag	PAPI Item Description
L1_DCM	Level 1 data cache misses
L1_ICM	Level 1 instruction cache misses
L2_DCM	Level 2 data cache misses
L2_ICM	Level 2 instruction cache misses
L3_DCM	Level 3 data cache misses
L3_ICM	Level 3 instruction cache misses
L1_TCM	Level 1 total cache misses
L2_TCM	**Level 2 total cache misses**
L3_TCM	Level 3 total cache misses
CA_SNP	Snoops
CA_SHR	access to shared cache line (SMP)
CA_CLN	access to clean cache line (SMP)
CA_INV	Cache Line Invalidation (SMP)
CA_ITV	Cache Line Intervention (SMP)
L3_LDM	Level 3 load misses
L3_STM	Level 3 store misses
BRU_IDL	Cycles branch units are idle
FXU_IDL	Cycles integer units are idle
FPU_IDL	Cycles floating point units are idle
LSU_IDL	Cycles load/store units are idle
TLB_DM	**Data translation lookaside buffer misses**
TLB_IM	Inst xlation lookaside buffer misses
TLB_TL	Total xlation lookaside buffer misses
L1_LDM	Level 1 load misses
L1_STM	Level 1 store misses
L2_LDM	Level 2 load misses
L2_STM	Level 2 store misses
BTAC_M	BTAC miss
PRF_DM	Prefetch data instruction caused a miss
TLB_SD	Xlation lookaside buf shootdowns (SMP)
CSR_FAL	Failed store conditional instructions
CSR_SUC	Succ store conditional instructions
CSR_TOT	Total store conditional instructions

PAPI Flag	PAPI Item Description
MEM_SCY	Cyc Stalled Waiting for Memory Access
MEM_RCY	Cyc Stalled Waiting for Memory Read
MEM_WCY	Cyc Stalled Waiting for Memory Write
STL_ICY	Cyc with No Instruction Issue
FUL_ICY	**Cyc with Maximum Instruction Issue**
CCY	Cyc with No Instruction Completion
FUL_CCY	Cyc with Maximum Instruction Completion
HW_INT	Hardware interrupts
BR_UCN	Uncond branch instructions executed
BR_CN	Cond branch instructions executed
BR_TKN	Cond branch instructions taken
BR_NTK	Cond branch instructions not taken
BR_MSP	Cond branch instructions mispredicted
BR_PRC	Cond branch instr correctly predicted
FMA_INS	FMA instructions completed
TOT_IIS	Total instructions issued
TOT_INS	Total instructions executed
INT_INS	Integer instructions executed
FP_INS	**Floating point instructions executed**
LD_INS	Load instructions executed
SR_INS	Store instructions executed
BR_INS	Total branch instructions executed
VEC_INS	**Vector/SIMD instructions executed**
FLOPS	Floating Point instructions per second
RES_STL	Any resource stalls
FP_STAL	FP units are stalled
TOT_CYC	Total cycles
IPS	Instructions executed per second
LST_INS	**Total load/store inst. executed**
SYC_INS	Sync. inst. executed

View Menu

- Program
  - Charts
    - Total Messages
    - CPU Balance
    - Message Balance
    - MPI Calls
    - MPI Wallclock Time
    - Wallclock Time
    - Custom Chart
  - Views
    - Calling Tree
    - Loop Time
    - Advisor
  - Tables
    - Compile-time
    - MPI Call Sites
    - MPI Summary
    - Loops
    - Inclusive Run-time
    - Run-time
    - Global Symbols
- Procedure
  - Source Code Browser
  - Code Abstract
  - Loop Table
  - Call Table
  - MPI Call Sites
  - Symbol Table

DEEP: Screen Shot

DEEP

File Edit Mode View Options Help

x_solve..f | copy_faces..f

```
c-----------------------------------------------
      end do

      call mpi_irecv(in_buffer(sr(0)), b_size
     >     dp_type, successor(1), WEST,
     >     comm_rhs, requests(0), error)
      call mpi_irecv(in_buffer(sr(1)), b_size
     >     dp_type, predecessor(1), EAST,
     >     comm_rhs, requests(1), error)
      call mpi_irecv(in_buffer(sr(2)), b_size
     >     dp_type, successor(2), SOUTH,
     >     comm_rhs, requests(2), error)
      call mpi_irecv(in_buffer(sr(3)), b_size
```

Source Code

Pie Chart: Total messages | CPU Balance | Message Balance

- 16% matmul_sub
- 16% matvec_sub
- 10% z_solve_cell
- 9% x_solve_cell
- 9% binvcrhs
- 9% y_solve_cell
- 5% copy_faces
- 3% lhsx
- 3% lhsy
- 20% other

Charts | Views | Tables

matmul_sub | copy_faces

Operation	File/Line
Code: (stmts=2)	copy_faces.f/151
EndLp	copy_faces.f/153
EndLp	copy_faces.f/154
EndLp	copy_faces.f/155
Endif	copy_faces.f/156
EndLp	copy_faces.f/161
EndLp	copy_faces.f/166
MPI: MPI_IRECV	copy_faces.f/168
MPI: MPI_IRECV	copy_faces.f/171

Code Abstraction | Loop Performance | Call Performance

Processes | Performance Advisor

DEEP Performance Advisor

matmul_sub

The ratio of CPU time to wall clock time is low. Check for overloaded system, or excessive synchronizations or I/O. (0.415)

Loop/J x_solve.f/557

Loop uses a large percentage of time. Make sure the loop is optimized. (34.5)

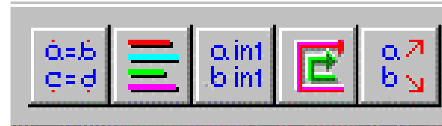
Loop has a very small average iteration count. Consider unrolling the loop entirely. (5)

Advisor | Symbol Viewer

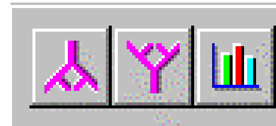
G:\DEEP\MPI_NasP\BT\\copy_faces.f | 168

DEEP Toolbars

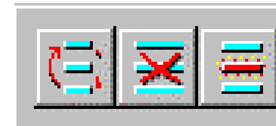
- Procedure Toolbar.
  - Source code.
  - Code abstract.
  - Symbol table.
  - Loop performance.
  - Call performance.



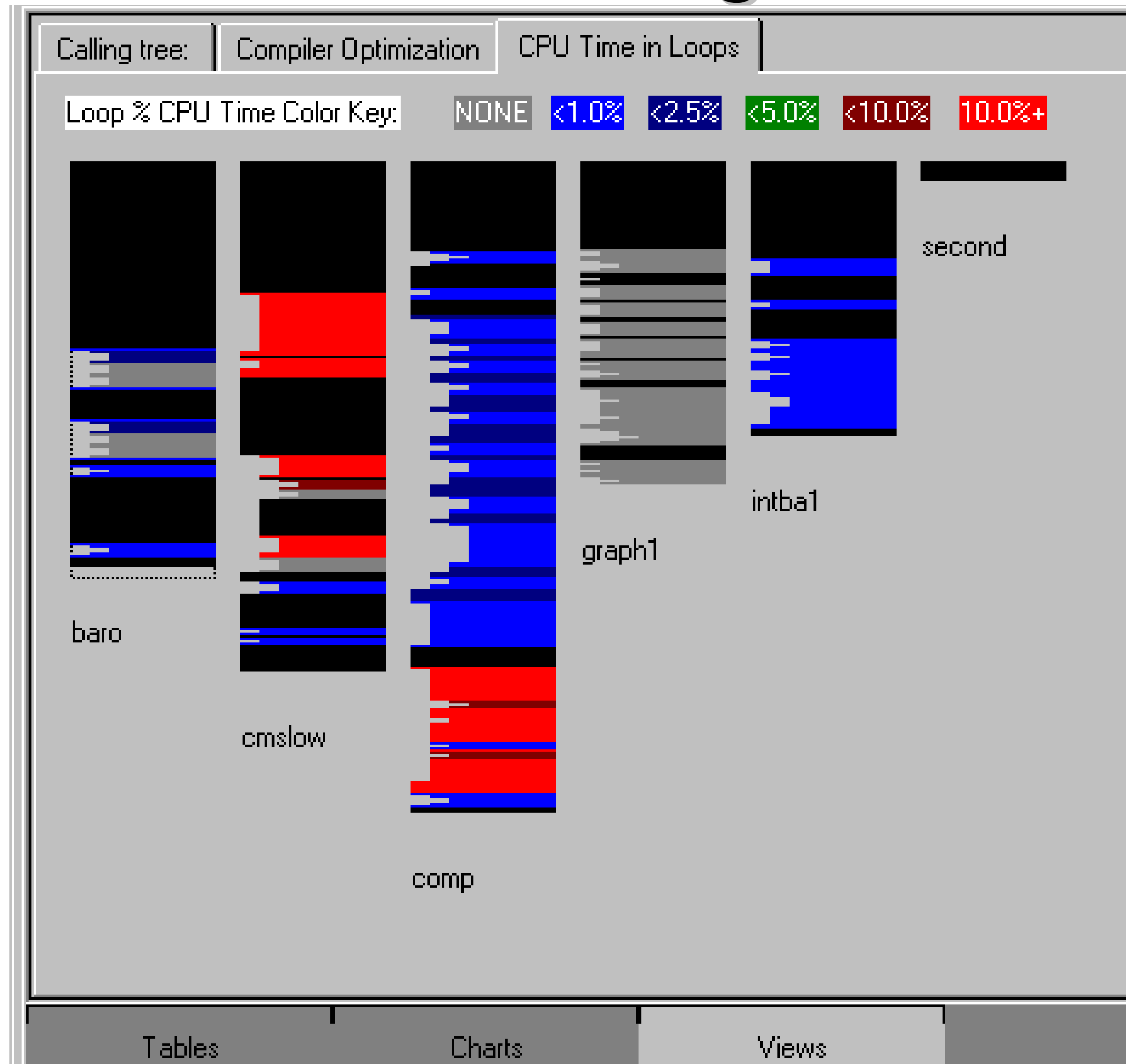
- Program Toolbar.
  - Calling tree.
  - Called tree.
  - Charts.



- List Manipulation Toolbar.
  - Sort.
  - Prune.
  - Highlight.



DEEP: Whole Program View

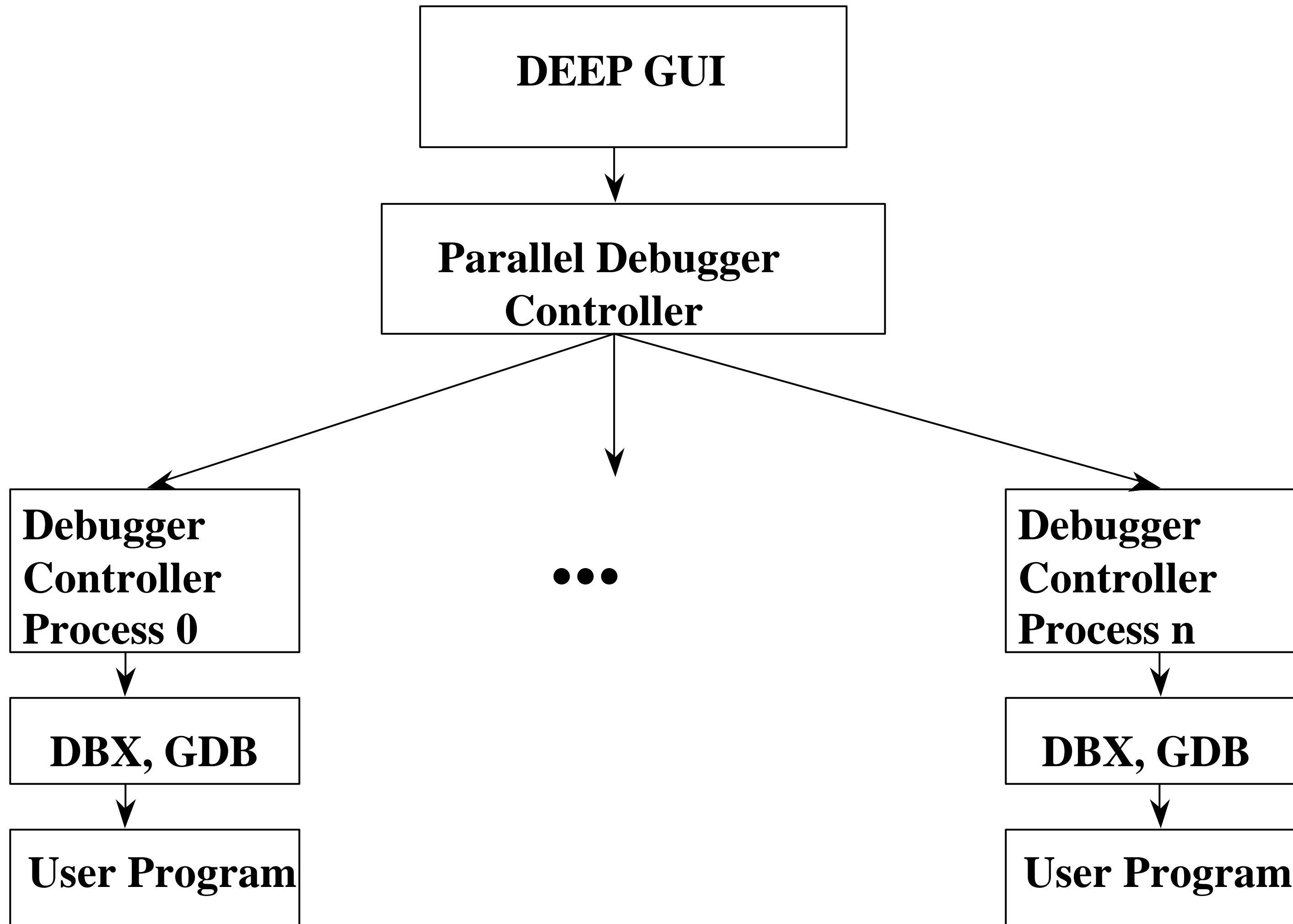


DEEP: The big picture

DEEP: Debugger

- Parallel Source Code Debugger currently under development to complement the existing program analysis features.
- Control existing scalar debuggers running on each parallel process.
  - rely on dbx, gdb, etc. for scalar debugger.
- Sophisticated GUI.
  - multiple viewers available.
- Examine values of parallel variables at breakpoints or through “watches”.
- Keep track of current process status for all processes.

DEEP:Debugger structure



DEEP Debugger Screen

DEEP

File Edit Mode View Options Help

0 1 2 3 4 5

is.c

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

    int             i, iteration, itemp;

    double          timecounter, maxtime;

/*  Initialize MPI */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    MPI_Comm_size( MPI_COMM_WORLD, &comm_size );

/*  Initialize the verification arrays if a valid class */
    for( i=0; i<TEST_ARRAY_SIZE; i++ )
        switch( CLASS )
        {
            case 'S':
                test_index_array[i] = S_test_index_array[i];
                test_rank_array[i]  = S_test_rank_array[i];
                break;
            case 'A':
                test_index_array[i] = A_test_index_array[i];
                test_rank_array[i]  = A_test_rank_array[i];
                break;
            case 'W':
```

Source Code

From Debugger

```
Result = 3
fdb?

13 more messages expected
Command : [lpid : 3] print comm_size
Results : [lpid : 3]
Result = 6
fdb?

12 more messages expected
Command : [lpid : 2] continue
Results : [lpid : 2] Continue program: is.exe
Breakpoint #2: main() at line 881 in D:\CPP\Deep\Tests\IS\is.c.
881                 switch( CLASS )
fdb?
11 more messages expected
Command : [lpid : 4] print my_rank
Results : [lpid : 4]
Result = 4
fdb?

10 more messages expected
Command : [lpid : 1] continue
Results : [lpid : 1] Continue program: is.exe
Breakpoint #3: main() at line 881 in D:\CPP\Deep\Tests\IS\is.c.
881                 switch( CLASS )
fdb?
9 more messages expected
Command : [lpid : 5] print comm_size
Results : [lpid : 5]
```

Groups | Program Output

Watch 1

Expression	Value
my_rank	<0> 0, <1> 1, <2> 2, <3> 3, <4> 4, <5> 5
comm_size	6

Watches

Processes

Logical ID	Machine ID	Status	Groups
0	Appgeo/25427	Paused at is.c:881	MPI_COMM_WORLD
1	Appgeo/25428	Paused at is.c:881	MPI_COMM_WORLD
2	Appgeo/25429	Paused at is.c:881	MPI_COMM_WORLD
3	Appgeo/25430	Paused at is.c:881	MPI_COMM_WORLD
4	Appgeo/25431	Paused at is.c:881	MPI_COMM_WORLD
5	Appgeo/25432	Paused at is.c:881	MPI_COMM_WORLD

Processes

From Debugger 0 Paused

DEEP: Future Directions

- "Project" feature: full IDE capability
  - Specify source files, DEEP launches compiles, links, etc.
- Improved support for large numbers (<100) of processors.
  - Search tools to look for outlying cases...
  - Matrix rather than column displays...
- Allow inserting of Library calls from menu.
  - Provide correct number and name of arguments.
  - Allow insertion of arbitrary set of routines (from file).
  - Check syntax and semantics of calls. MPI, BLAS, VSIPL, etc.
- "Real-time" profiling, through debugger interface.

DEEP: More Directions

- Distributed Objects
  - Corba/DCOM...
- Enhanced structure debugging for parallel structures.
- Event Trace Tool
  - Map events to original source code line, zoom in/out.
- ccNUMA performance tuning
  - relate local/global memory perf. back to source code.
- Support for counter multiplexing with PAPI.

Advantages of DEEP

- Can use DEEP for all major parallel programming paradigms.
- Quickly zoom in on the bottlenecks and problems. Saves time, especially on larger applications.
- More fun than putting in your own print statements. Look at interesting displays.
- All tools in one place, don't have to learn several interfaces. Tools all work together.
- Can be customized for special requirements.

Summary

- For further information:

David McNamara

Veridian -- Pacific-Sierra Research

2901 28th Street

Santa Monica CA 90405

(310)-314-2338

brian@psrv.com