DEEP: A Development Environment for Parallel Programs

Brian Q. Brode and Chris R.Warber
Pacific-Sierra Research Corporation
2901 28th Street, Santa Monica, CA 90405
info@psrv.com

Abstract

The use of the DEEP development environment to analyze parallel program performance is described. The full integrated environment contains tools for the creation, analysis and debugging of parallel programs. All information is related back to the original parallel source code. This paper describes the program analysis portion of DEEP, with examples of its use on data parallel programs (HPF and Data Parallel C) and shared memory parallel (SMP) programs.

1. Introduction

The DEEP system provides an integrated parallel program development environment that binds debugging and performance tools back to the original parallel source code. DEEP includes many useful tools in a highly interactive integrated GUI interface. DEEP provides a simple and intuitive way to understand and investigate parallel program structure, performance, and behavior. DEEP supports both data parallel and shared memory parallel programming.

1.1 Data Parallel Programming.

Data parallel programming allows the user to concentrate on the higher level aspects of the decomposition of the program data across the available processors, while the compilation system performs all the low-level bookkeeping and provides for all of the communication needed between processors. Data parallel programming languages include High Performance Fortran[1], which is an emerging standard first proposed in 1993, and the Data Parallel C Extensions (DPC)[2], accepted as a technical report in 1994 by the X3J11 ANSI C Committee.

Data parallel languages have two distinguishing features: syntax for describing the distribution of data across processors, and a method for making clear the parallel nature of calculations. HPF has a set of directives that allow specification of data distribution and DPC has "shape" declarations for this purpose. Both HPF and DPC have array syntax, which allows entire arrays of data to be manipulated with single statements.

Distributed memory parallel computer systems can range from networks of workstations or even PCs to large-scale massively parallel computer systems designed for efficient inter-processor communication. DEEP supports any computing platform that provides the MPI or PVM message-passing interfaces. DEEP works with both HPF and DPC programs[3], and can even support mixed language Fortran/C applications.

1.2 Shared Memory Programming

For SMP systems, DEEP supports Fortran and C programs that are being automatically parallelized[4], and programs that are being parallelized by hand with the recently specified OpenMP[5] set of directives. DEEP for SMP targets systems that support threads, such as the POSIX threads standard, and the system works on both UNIX systems and Windows/NT.

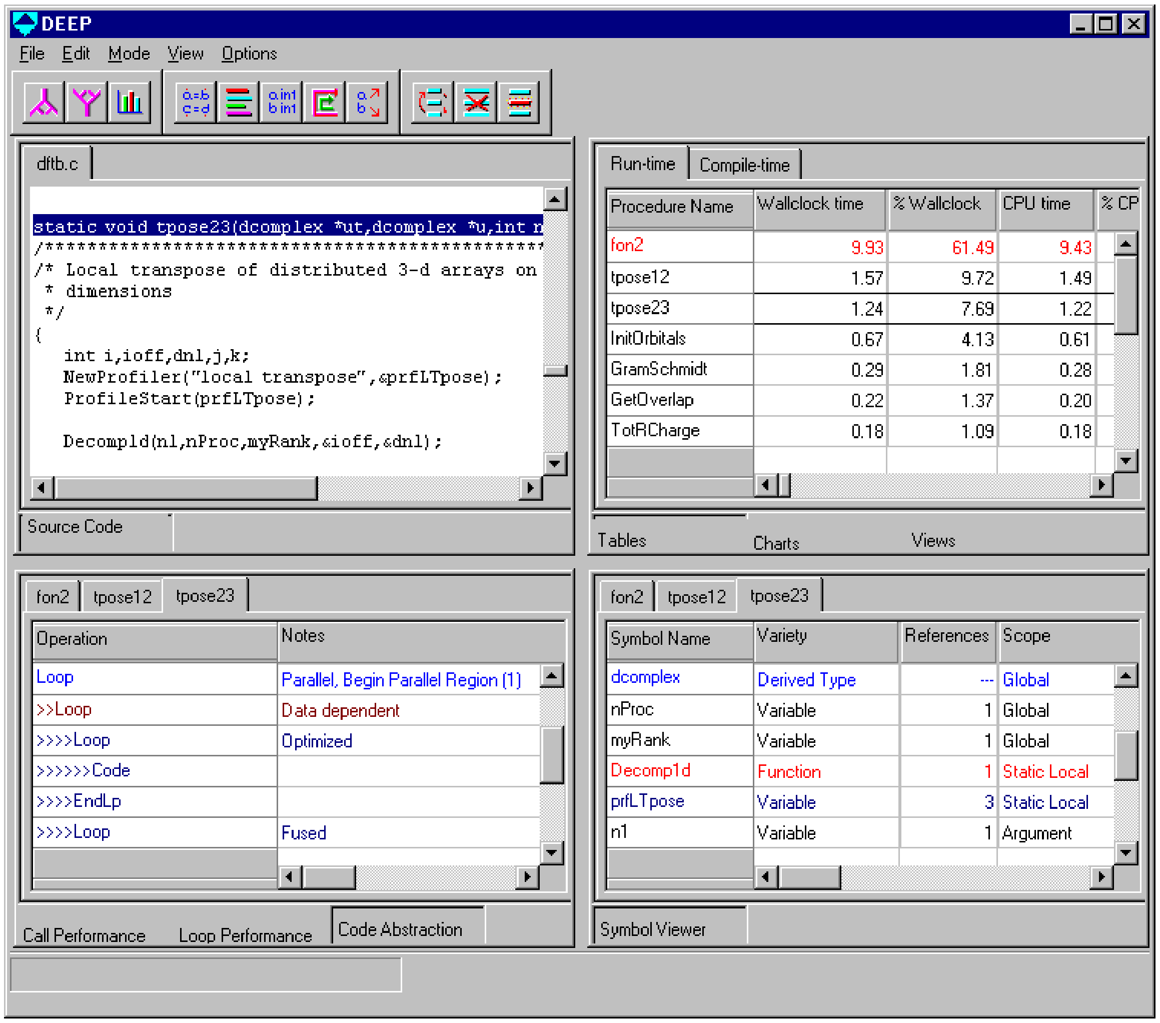
2. The DEEP Framework

The DEEP system generates an abundance of information about a parallel program. Presenting this type of information in a coordinated package can be a significant challenge to user interface design. DEEP is organized around a single GUI framework that organizes and displays the wealth of information provided by the system. The DEEP framework is organized into user-configurable panels that contain “viewers”. Each viewer in turn contains pages that hold the information.

Panels can be reconfigured by grabbing and moving. Normally, a user will have three to five panels open. When the size of one panel is changed, the other panels automatically readjust themselves accordingly. We find this arrangement much easier to deal with than many independent windows popping up all over the screen. Also, a panel can be expanded to fill the whole DEEP window, then restored to its previous configuration; this is easy to do through a right-click pop-up menu.

Within these panels are viewers, a viewer being a software tool for examining some aspect of a parallel program. Viewers can be docked in any panel or popped into their own window. Viewers in a panel or pages in a viewer are selected with a click of a button. Figure 1 shows a DEEP display with three toolbars (along the top) and four panels. Tabs along the bottom of the panels select viewers inside the panel; tabs at the top of each panel select between pages in a viewer.

Three toolbars can be seen near the top of Figure 1. Toolbars contain buttons that invoke various functions. The buttons contain help tips, which is useful if the icon does not clearly bring to mind the function to be performed. The toolbars can be dragged to docking areas at the top, bottom, left and right of the panels.



```
static void tpose23(dcomplex *ut,dcomplex *u,int n
/*************************************************
/* Local transpose of distributed 3-d arrays on
 * dimensions
 */
{
   int i,ioff,dnl,j,k;
   NewProfiler("local transpose",&prfLTpose);
   ProfileStart(prfLTpose);

   Decomp1d(nl,nProc,myRank,&ioff,&dnl);
```

Procedure Name	Wallclock time	% Wallclock	CPU time	% CP
fon2	9.93	61.49	9.43	
tpose12	1.57	9.72	1.49	
tpose23	1.24	7.69	1.22	
InitOrbitals	0.67	4.13	0.61	
GramSchmidt	0.29	1.81	0.28	
GetOverlap	0.22	1.37	0.20	
TotRCharge	0.18	1.09	0.18	

Operation	Notes
Loop	Parallel, Begin Parallel Region (1)
>>Loop	Data dependent
>>>>Loop	Optimized
>>>>>>Code	
>>>>EndLp	
>>>>Loop	Fused

Symbol Name	Variety	References	Scope
dcomplex	Derived Type	---	Global
nProc	Variable	1	Global
myRank	Variable	1	Global
Decomp1d	Function	1	Static Local
prfLTpose	Variable	3	Static Local
n1	Variable	1	Argument

Figure 1: DEEP Framework

3. DEEP Program Views

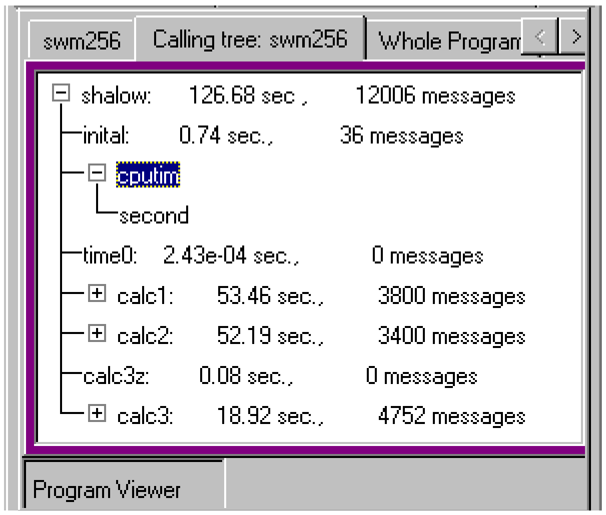
DEEP program analysis tools allow use of both compile-time information (gathered by the compilers) and run-time information (gathered by a run-time profiling library) to investigate a parallel program in more detail. In analyzing a program with the DEEP system you would normally start with the tools that look at the whole program, identify procedures of interest, and then “drill down” with tools that look at the internal structure of individual procedures. We will examine the whole-program tools first.

3.1 Program Information Table

DEEP allows you to inspect the program as a whole in various forms. For inspection of raw data gathered from compiling and executing the program, the *program information tables* are the place to start; they list each of the procedures in the program with various statistics. There is a compile-time table that shows information gathered from compile-time analysis and optimization of the program, such as the number of parallel loops. The run-time information table lists CPU time, wallclock time, number of messages passed, number of calls, number of loops executed, average iteration count, etc. The tables can be sorted on any field, and procedures of extra interest can be highlighted by user-selected criteria. Procedures that are not of interest can be pruned. Selecting a procedure in the table (by clicking with the mouse) brings up detailed information about the procedure in other panels. The buttons in the tool bars at the top of the display can be used to launch other views of the program as well. The example in the upper right quadrant of Figure 1 shows three of the time-related fields in the program table; by default, the table is sorted by wallclock time and all procedures that use more than 10% of the wallclock time are highlighted.

3.2 Call Tree Display

The *call tree display* allows browsing of the call relationships of the procedures. Any procedure can be set as the root, and both calling trees (down to leaf procedures) and called trees (back to the main procedure) can be browsed. If runtime data is available, the trees are annotated with inclusive time and message counts. This performance annotation helps direct the browsing of the tree to branches that used the most resources. The tree can be expanded or contracted at each level by clicking on the + and - icons (see Figure 3). When you get to a point in the tree where you would like to have more information, you can use the tool bars to move directly to detailed displays, or the source code, of the selected procedure.



```
⊟ shalow:    126.68 sec.,    12006 messages
├─inital:    0.74 sec.,      36 messages
├─⊟ cputim
│   └─second
├─time0:     2.43e-04 sec.,  0 messages
├─⊞ calc1:   53.46 sec.,     3800 messages
├─⊞ calc2:   52.19 sec.,     3400 messages
├─calc3z:    0.08 sec.,      0 messages
└─⊞ calc3:   18.92 sec.,     4752 messages
```

Figure 2: Call Tree View

3.3 Whole Program View

With the *whole program view*, the entire program can be examined on one screen, with color-coded lines of pixels representing performance of source lines in each subprogram. Each procedure is represented by a rectangle which is proportional to the number of source code lines. The lines of pixels represent the individual lines of code, and are indented to correspond to control structures in the original source (loops, conditional blocks, etc.). Clicking on any pixel in a rectangle will bring up the corresponding source line in the source code editor, and also bring up that area in the code abstract viewer for that procedure. This allows you to move quickly from the high-level program information to specific information about an area of the program.

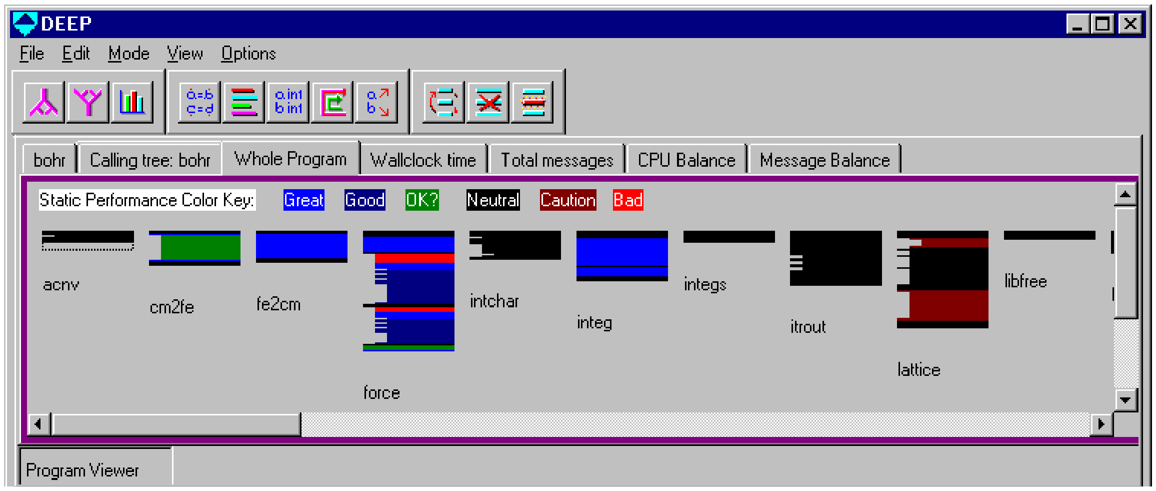


Figure 3: Whole Program View

In Figure 3, DEEP uses static performance information from a data parallel compiler to color-code lines in each procedure, based on how well they were compiled for parallel execution. Source lines that result in inefficient code (messages need to be passed to other processors) are highlighted at the red end of the spectrum; source lines of loops that are partitioned cleanly (with minimal message passing) across the parallel processors are highlighted at the blue end of the spectrum (see the key at the top of the window in Figure 4). Other colorings of the whole program display are possible, including ones based on dynamic performance information.

3.4 Summary Charts

Several summary charts are prepared, including the wall clock time and the breakdown of total messages passed. These let the user quickly see where the resources are being spent in the parallel program. These charts are presented as colored pie charts or bar graphs.

3. 5 Load Balance Displays

The *message load balance display* and the *CPU load balance display* are intended to give information on how the computational load is distributed. Is one processor sending most of the messages or doing most of the processing? If so, then you may want to reconsider how you have distributed the data. For instance, a block-cyclic approach may provide better load balance than a pure block distribution on a data parallel program. For SMP programs, low utilization of the last few threads may indicate that the program can get by with less threads.

An example of a message balance display for a data parallel program run on four processes in seen in Figure 4. The display shows the number of sends and receives for each logical process.

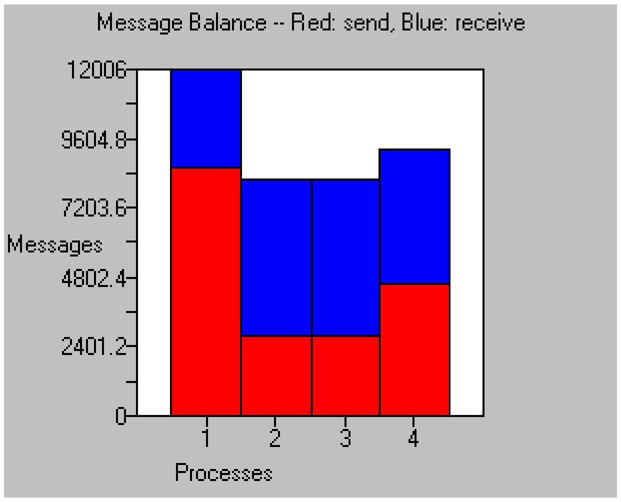


Figure 4 – Message Load Balance

4. DEEP Procedure Views

When a procedure of interest is determined by examining the program-level views, you can move to the more detailed procedure-level displays.

4.1 Code Abstract Viewer

The *code abstract viewer* allows the programmer to examine routines with an overview of important control structures. This view is annotated with optimization notes from the data parallel or SMP compiler. From this display, you can move to the corresponding line in the source code editor with a click of the mouse. The display is color-coded to provide performance feedback at a glance. The lower left quadrant of Figure 1 shows part of the abstract for a routine.

4.2 Symbol Viewer

You can track the uses and definitions of variables throughout the parallel program with the *symbol viewer*. In the lower right quadrant of Figure 1 various kinds of symbols can be seen. Symbol information including scope and attributes is available. The file of definition is displayed, and if clicked on will bring up this file in the editor.

Clicking on the “References” column brings up a new page that lists all references of a variable, and allows the user to move through these references with the click of a button. As each reference is selected, the corresponding line of code is displayed in the source code editor.

4.3 Performance Viewers

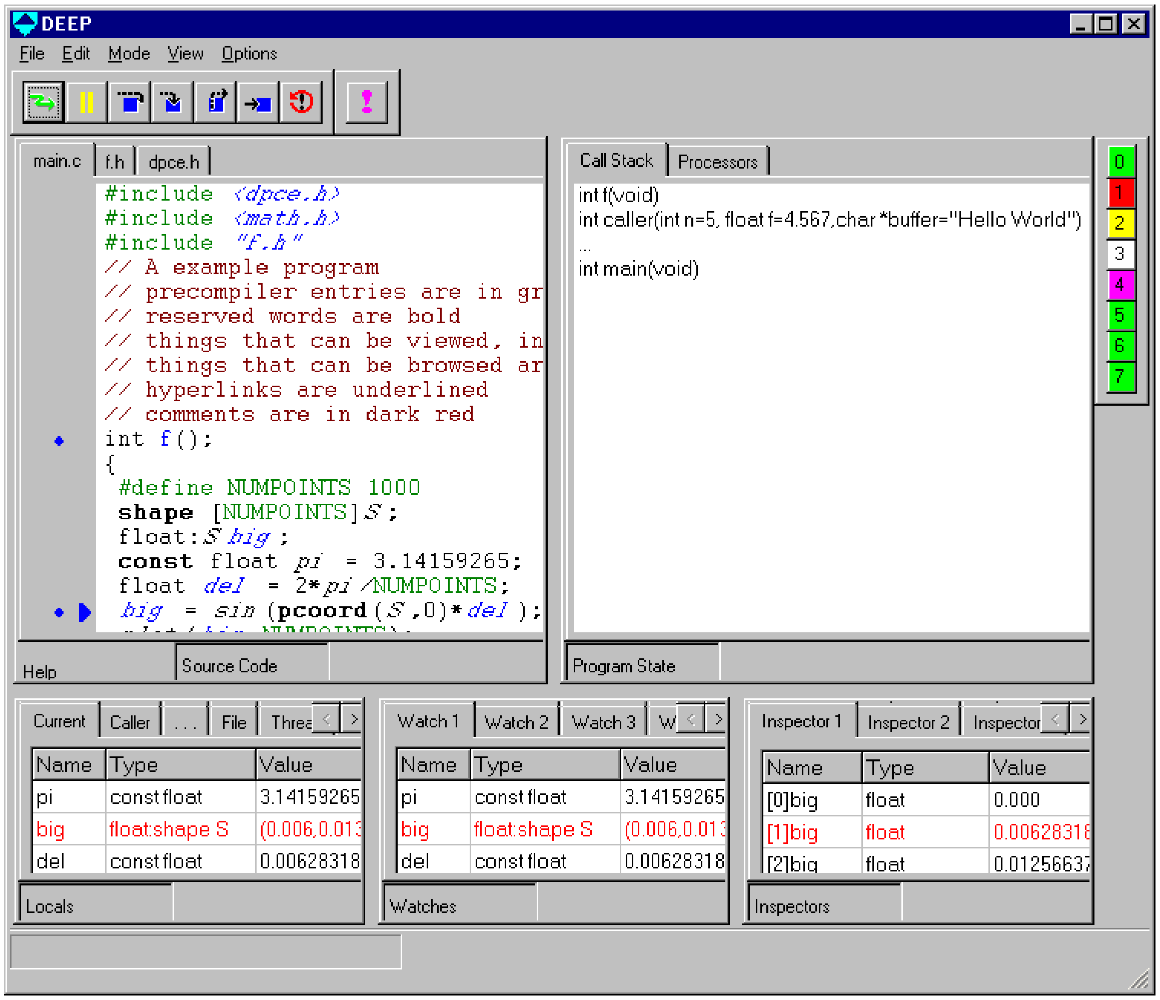
The *loop performance viewer* and *call performance viewer* present tabular information on the details of a procedure’s performance. This performance information is gathered during run-time at the individual loop and call level.

5. DEEP Debugging Tools

DEEP also provides run-time interactive GUI-based debugging of parallel programs at the source code level. The user can switch back and forth between the analysis and debugging modes. Tools include:

- *Source Code Viewer*. Shows the execution location in the source code. Features language-sensitive syntax highlighting, and interactive symbol browsing and variable value inspection.
- *Breakpoints*. Both conditional and unconditional breakpoints are supported. Breakpoints can be set with a single click.
- *Watch points*. Watch points suspend execution when the value of a variable changes.
- *Trace points*. When a trace point is reached in the source code, user selected expressions are displayed in the program log viewer.
- *Performance Zones*. You can establish performance zones in the source; the system can display the CPU time, wall clock time, messages sent or received, and I/O done by each process in each zone.
- *Inspection and watch viewers*. DEEP provides watch viewers for simple variables, and inspection viewers for structured variables; variables can be selected from the Source Code Viewer.
- *Graphical array viewers*. Multidimensional distributed arrays can be visualized using two and three dimensional graphical tools.
- *Processor Status*. The current state of all the processors is shown by its color in the processor status viewer. In addition, detailed information about individual processes can also be displayed.

A demonstration screen snapshot of the debugger interface can be seen in Figure 5.



```
#include <dpce.h>
#include <math.h>
#include "f.h"
// A example program
// precompiler entries are in gr
// reserved words are bold
// things that can be viewed, in
// things that can be browsed ar
// hyperlinks are underlined
// comments are in dark red
int f();
{
 #define NUMPOINTS 1000
 shape [NUMPOINTS]S;
 float:S big ;
 const float pi  = 3.14159265;
 float del  = 2*pi /NUMPOINTS;
 big  = sin (pcoord(S ,0)*del );
```

```
int f(void)
int caller(int n=5, float f=4.567,char *buffer="Hello World")
...
int main(void)
```

Name	Type	Value
pi	const float	3.14159265
big	float:shape S	(0.006,0.013
del	const float	0.00628318

Name	Type	Value
pi	const float	3.14159265
big	float:shape S	(0.006,0.013
del	const float	0.00628318

Name	Type	Value
[0]big	float	0.000
[1]big	float	0.00628318
[2]big	float	0.0125663

Figure 5 – Debugging Snapshot

6. Future Work

DEEP is planned to include tools that allow the user to see what is happening in the system as the parallel program executes (parallel program monitoring tools). The user will be able to visualize the current status of processors, message traffic, and calculations. These tools are currently in development.

Much data parallel programming is done directly in MPI or similar message passing system and we plan to extend DEEP to support this programming model as well. And since DEEP supports both Data Parallel and SMP models, extending it to support hydrid/cluster/NUMA and other such systems seems to be a natural progression.

7. Acknowledgments

This work received support from DARPA.

DEEP is a trademark of Pacific-Sierra Research Corporation.

References

[1] *High Performance Fortran Language Specification*, HPF Forum, Version 2.0, 10/19/96
[2] *Data Parallel C Extensions*, Numerical C Extensions Group of X3J11, DPCE subcommittee, Version 1.6, X3J11/94-080, 12/31/94
[3] *VAST-DPC User's Guide*, Version 1.2, March 1997, Pacific-Sierra Research Corporation.
[4] *VAST-2/Parallel User's Guide*, Version 2.0, February 1998, Pacific-Sierra Research Corporation.
[5] *OpenMP Fortran Application Program Interface*, Version 1.0, October 1997