Using the lgdb Comparative Debugging Feature

S–0042–22

RECORD OF REVISION

S–0042–22 Published December 2013 Supports Cray Debugger Support Toolkit release 2.2.0 running on Cray XE, Cray XK, and Cray XC30 systems.

S–0042–20 Published March 2013 Supports Cray Debugger Support Toolkit release 2.1.2 running on Cray XE, Cray XK, and Cray XC30 systems.

Abstract

Using the lgdb Comparative Debugging Feature S–0042–22

This paper describes the *comparative debugging* functionality first released in version 2.0 of `lgdb`, Cray's command line debugger. Comparative debugging technology enables programmers to debug a faulty program against a working version, by comparing data structures between the two executing programs. A demonstration utilizing the comparative debugging feature of `lgdb` to find an error within a faulty version of the High-Performance Linpack benchmark (HPL) is provided.

Contents

Introduction [1]

The `lgdb` command line parallel debugger can be used to debug applications compiled with CCE, PGI, and GNU Fortran, C, and C++ compilers. Basic operation is documented in the `lgdb`(1) man page. Version 2.0 of `lgdb` also introduced the first release of Cray's *comparative debugging* technology. Comparative debugging enables programmers to compare corresponding data structures between two executing applications. If the values of the corresponding data structures diverge, an error may exist and the user is notified. This capability is useful for locating errors that are introduced when applications are modified through code, compiler, or library changes, or when running an application on a different scale produces incorrect results.

Although this document offers an introduction to the concepts and constructs of comparative debugging within `lgdb`, Cray recommends accessing the comparative debugger technology through the new Cray Comparative Debugger (CCDB) with graphical user interface (GUI) that enhances the debugging capabilities of `lgdb`. For further information on CCDB, see the `ccdb`(1) man page.

Note: Throughout this document, some examples are left-justified to better fit the page. Left justification has no special significance.

1.1 The Comparative Debugging Cycle

Comparative debugging assumes there are two versions of an application to be compared, a reference version that is considered correct and a development version being debugged. The typical comparative debugging cycle involves following the use of key variables in the two applications, comparing their values, and tracing them back to their points of definition to refine the area within the development version where results first diverge. Although every debugging session takes its own unique path, the initial pass through of comparative debugging with `lgdb` includes the following steps.

Procedure 1. Initial pass of comparative debugging with `lgdb`

1. Locate Entry Point into Code.

   Where in the code does it make sense to begin comparing data structures? Which data structures must be compared? The user must have an in depth understanding of the source code in order to select and locate key data structures, determine comparison points, follow the path of execution, and understand the implications of the results.

2. Prepare executable files.

   Both applications will be launched for execution by `lgdb`, and must be compiled using the debugging option (`-g` or `-G`*n*) of the relevant compiler to include additional debugger information required by `lgdb`.

   Example 1. Compile code with debugging enabled

   In this example, two executable files, `version1` and `version2`, are created when the source code files `source1.f90` and `source2.f90` are compiled with debugging enabled.

   ```
   % ftn -g -o version1 source1.f90
   % ftn -g -o version2 source2.f90
   ```

3. To begin using `lgdb`, load the `cray-lgdb` module and then initiate the debugger.

   ```
   % module load cray-lgdb
   % lgdb
   ```

4. Specify resource requirements and launch applications.

   Applications are launched and processor resource requirements are defined by using the `launch` command within `lgdb`. The syntax of the command is:

`launch` [--args "*app_args*" | -a "*app_args*"] [--aprun-args "*aprun_args*" | -g "*aprun_args*"][--aprun-input "*input_file*" | -i "*input_file*"] [--env="*name=value*",--env="*name=value*",...] [--workdir="*work_path*" | -w="*work_path*"] $*proc_set path_to_executable*

The `launch` command requires the following parameters:

$*proc_set* Defines a debugger variable and associates it with the number of processing elements (PEs) in the application. For sequential applications, $*proc_set* is a single debugger scalar variable. For parallel applications, $*proc_set* is a debugger array variable, the size of which determines the number of PEs for the application.

The `launch` command transparently passes the number of PEs to `aprun`, through the `-n` option, to launch applications on batch systems.

path_to_executable

Specifies the path to the application executable. This is passed directly to `aprun`.

The `launch` command accepts the following options. Option arguments must be enclosed within quotation marks, such as `"args"`.

`--args` "*app_args*" | `-a` "*app_args*"

Passes *app_args* to the application executable.

`--aprun-args` "*aprun_args*" | `-g` "*aprun_args*"

Passes *aprun_args* to the `aprun` command.

`--aprun-input` "*input_file*" | `-i` "*input_file*"

Redirects the `stdin` of the `aprun` command to be *input_file*. This is useful for applications requiring input from `stdin`.

`--env=`*"name=value"*, `--env=`*"name=value"*,...

Sets the environment variable (defined by *name*) to *value*, for this `aprun` session instance. Note that `--env=` can be used more than once to set multiple environment variables.

`--workdir=`"*work_path*" | `-w=`"*work_path*"

Changes the current working directory, relative to its present setting where `lgdb` was invoked, to *work_path*. This is useful for applications that write files to the current working directory. If the `--workdir=` option is specified without a path, the current working directory will be changed to the location of the application's executable file. By default, if `--workdir=` is not specified, *work_path* is defined as the directory from where `lgdb` was invoked.

Example 2. Launching applications using `lgdb`

In this example, two PEs for each application, `version1` and `version2`, are launched and associated with the process sets `$working` and `$broken`, respectively.

```
dbg> launch $working{2} version1
dbg> launch $broken{2} version2
```

5. Define key data structures.

   In parallel programming, data is typically decomposed and distributed across numerous application PEs. To perform comparisons of distributed data structures, each individual piece must be obtained from the PEs and reconstructed. In `lgdb`, a *decomposition scheme* is created in script mode and specifies the reconstruction of distributed variables into the global representation of the data by defining four required characteristics: dimensionality, distribution, process grid, and dimension order. Enter the following command to initiate script mode and create a decomposition scheme, $*scheme_name*. Script subcommands are read until the `end` subcommand is issued, returning `lgdb` to interactive mode. Following are explanations of the decomposition script subcommands.

   ```
   dbg all> decomposition $scheme_name
   ```

   `dimension` Specifies the size and dimensionality of the global reconstruction. Each characteristic must have the same dimensionality as defined by the `dimension` subcommand.

`distribute` Specifies the distribution type for each dimension of the reconstruction. Distribution options are:

- `block` — Equal-sized *chunks* of data are assigned to each PE.
- `cyclic` — Elements in the dimension are dealt out in round robin fashion.
- *a numeric value* — Representing the blocking factor used to partition the dimension in a *block-cyclic* distribution.
- asterisk (`*`) — Indicating that this dimension is not distributed and, therefore, each PE in the global reconstruction contains all of the data in that dimension.

`proc_grid` Defines the process grid for the reconstruction by specifying the number of PEs contained in each dimension. If a dimension is not distributed, the value for that dimension must be defined as an asterisk (`*`).

`dim_order` Defines the order in which the application PEs are assigned in each dimension of the process grid for the global reconstruction. Each local chunk of data obtained from each PE must be placed into the global reconstruction. To do this, each PE is assigned a logical position in the process grid for its chunk of data. When considering *n*- dimensional distributions, any of the *n* dimensions can be assigned sequential numbered PEs, and any of the other higher order dimensions can be incremented after the dimension containing sequential PEs is filled.

`dim_order` is defined by assigning a sequential number from `1` to *n* to each of the defined distributed dimensions indicating fastest to slowest varying dimension, respectively. If a dimension is not distributed, the order must be defined as an asterisk (*). The fastest varying dimension is the dimension assigned sequential PEs up to its corresponding grid size. The second fastest varying dimension is incremented after the fastest varying dimension is completely filled and PEs are again assigned to the fastest varying dimension. This process continues until all PEs have been assigned to all of the *n*- dimensions.

Example 3. Two-dimensional data decomposition scheme

This example creates a decomposition scheme for an 8 x 8 array:

```
dgb all> decomposition $data_a
> dimension 8,8
> distribute block,*
> proc_grid 4,*
> dim_order 1,*
> end
dgb all>
```

The first dimension of the array is distributed in a block manner, and the second dimension is not distributed; therefore, each application PE contains all eight elements. The `proc_grid` definition indicates that the data is to be distributed over four PEs in the first dimension and not distributed in the second dimension. Thus, the local chunk of data for each PE is a 2 x 8 array of data, or two rows of the data array. The `dim_order` definition specifies that the first dimension is the fastest, and in this case, the only varying dimension because the second dimension is not distributed.

The decomposition construct provides a method to reconstruct distributed data into a global view that can be compared across applications. Instead of writing thousands of individual assertion statements to conduct comparisons of data variables across application PEs, users can create a decomposition scheme to globally reconstruct the data automatically.

6. Employ assertions to compare data structures.

   Assertions, the key construct used in `lgdb`, define the names of two data structures that are to be compared. There are two types of assertions available in `lgdb`, *imperative* and *declarative*.

   Imperative assertions allow a user to interactively compare data structures between the executing applications when they are suspended at user-define breakpoints. The user can create breakpoints within the two applications before they are simultaneously executed. When a breakpoint is reached and the applications are suspended, the user issues a `compare` command to compare the contents of key data structures at that time.

   Example 4. Use an imperative assertion to compare data structures

   In this example, the variable `Value` in the reference application "working" is compared with the variable `Value` in the development application "broken".

   ```
   dgb all> compare $working::Value = $broken::Value
   dgb all>
   ```

   The process of debugging using only imperative assertions would involve numerous iterations of defining breakpoints, resuming or restarting the applications, and comparing the contents of key data structures. If the user wants to compare the results of computations within a loop, the `compare` command must be manually invoked for each iteration of the loop when a breakpoint is reached. This is obviously not the most efficient method.

   Declarative assertions allow a user to state a set of spatial and temporal conditions that must be satisfied for the data structures within the development version to be considered correct. In `lgdb`, declarative assertions are defined by the `assert` subcommand within an *assertion script*, and state that a data structure (the spatial condition) at a specific line (the temporal condition) in the development application should contain the same value as the corresponding data structure, at a specific line, in the reference application. An assertion script can contain as many assertions as needed.

   The `build` command initiates assertion script mode, subcommands are accepted until the `end` subcommand is entered to return `lgdb` to interactive mode, after which the `start` command is used to initiate execution of the assertion script. The script will continue to successful completion or until the assertion interpreter halts due to assertion failures or application errors.

Example 5. Use a declarative assertion to compare data structures

The assertion script in this example instructs `lgdb` to compare the value of the variable `stor1` at line 234 of `source1.f90` for the application associated with the process set `$working` with the variable `stor1` at line 187 of `source2.f90` for the application associated with the process set `$broken`.

```
dbg all> build $test
> assert $working::stor1@"source1.f90":234 = $broken::stor1@"source2.f90":187
> end
dbg all>
```

`lgdb` will create breakpoints in both applications at the respective line numbers, and will compare the specified variables when the assertion script is executed. If the comparison does not detect an error, the applications are automatically resumed; otherwise, execution of the applications is halted and the difference is reported.

7. Evaluate results and repeat debugging process, as necessary.

   Results from the assertion script provide clues to the user as to other areas of the application code that should be investigated. Tracing the path of data structure calculations to find where results diverge will likely require multiple iterations of the comparative debugging cycle.

   With this preliminary release of the comparative debugging feature, it is necessary to quit `lgdb` and then restart it, in order to release the applications and associated variables, making it possible to relaunch the applications and begin another debugging cycle. This will be resolved in a future release.

Comparative Debugging Demonstration [2]

This demonstration illustrates the use of comparative debugging capabilities of `lgdb` to detect and analyze data variances between two applications, a reference version and a development version, that differ in results. The High-Performance Linpack (HPL) benchmark, part of the HPC Challenge Benchmark set, is the test application. All necessary files can be found in the `demo` directory of the `lgdb` release package. Follow the directions in the `README` file to properly set up and build the demo. For further information about the HPL benchmark, go to: http://icl.cs.utk.edu/hpcc/index.html.

2.1 Staging the Demonstration

Two binaries are compiled for the HPL demonstration, `hpcc_broken` and `hpcc_working`. `hpcc_broken` is built from HPL source into which a bug was deliberately introduced, while `hpcc_working` is built from the original HPL source code. Both executables are launched using the `aprun` command requesting four PEs each; each PE maps to one MPI (Message Passing Interface) rank. Upon completion of the run, an output file is generated containing results of the run.

Note: The scale of this demo is small for practical considerations. The techniques used are applicable when running on thousands of processors.

Run `hpcc_broken`:

```
% aprun -n 4 ./hpcc_broken
```

The generated output file, `hpccoutf.txt`, contains a failure message. The following is a partial listing from that log:

```
- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be               1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4        1000    80     2     2               0.05             1.306e+01
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 283705609311.4508057 ...... FAILED
||Ax-b||_oo  . . . . . . . . . . . . . . . . . =         132.675817
||A||_oo . . . . . . . . . . . . . . . . . . . =         262.773468
||A||_1  . . . . . . . . . . . . . . . . . . . =         263.865287
||x||_oo . . . . . . . . . . . . . . . . . . . =          16.028046
||x||_1  . . . . . . . . . . . . . . . . . . . =        3689.284539
||b||_oo . . . . . . . . . . . . . . . . . . . =           0.499776
================================================================================

Finished      1 tests with the following results:
              0 tests completed and passed residual checks,
              1 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
```

Run `hpcc_working`:

```
% aprun -n 4 ./hpcc_working
```

The output of `hpcc_working` is appended to `hpccoutf.txt` and does not contain a failure message. The following is a partial listing from the log file:

```
- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be              1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4        1000    80     2     2               0.05              1.337e+01
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0054597 ...... PASSED
================================================================================

Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
```

2.2 The Comparative Debugging Process — Initial Pass

The HPL benchmark is a good choice for a debugging demonstration as its size and complexity provides sufficient challenges to make the debugging process interesting. As you will see, after the initial pass through the debugging steps described earlier, several iterations of defining key data structures, employing assertions and evaluating results (step 5 through step 7) are needed to follow the clues back to the origin of the bug.

Important: In many of the examples within this demonstration, some command lines are split across two lines for publishing purposes only. `lgdb` does not interpret commands split across multiple lines.

2.2.1 Locate Entry Point into Code

To debug this problem, a logical entry point into the HPL code must first be determined. The `FAILED` message in the `hpcc_broken` output is being generated by the following section of code from the source file `HPL_pdtest.c`:

```
429     HPL_fprintf( TEST->outfp, "%s%16.7f%s%s\n",
430       "||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= ", resid1,
431       " ...... ", ( resid1 < TEST->thrsh ? "PASSED" : "FAILED" ) );
```

This checks to see if the variable `resid1` is less than the value of `TEST->thrsh`. If so, `PASSED` is printed to the output file, otherwise `FAILED` is printed. Something must be different with the calculation of `resid1`, on line 418 of `HPL_pdtest.c`, in the broken version of the code:

```
418     resid1 = resid0 / ( TEST->epsil * ( AnormI * XnormI + BnormI ) * (double)(N) );
```

Therefore, the focus is on the variables going into the calculation of `resid1`.

2.2.2 Specify Resource Requirements and Launch Applications

After loading the `cray-lgdb` module and invoking `lgdb`, the first task is to launch both the broken and working versions of the HPL application using the `launch` command. As described earlier, `launch` associates an instance of an application with an internal process set representation. Therefore, in the following output, launching four PEs of the `hpcc_broken` binary associates them with the process set `$broken`.

Note: Commands shown are available in the script files found in the `hpcc_scripts` directory. Scripts can be used inside `lgdb` using the `source` command.

```
dbg all> launch $broken{4} hpcc_broken
Starting alps application, please wait...
Creating MRNet communication network...
Waiting for debug servers to attach to MRNet communications network...
Timeout in 60 seconds. Please wait for the attach to complete.
Number of dbgsrvs connected: [1];  Timeout Counter: [0]
Number of dbgsrvs connected: [1];  Timeout Counter: [1]
Number of dbgsrvs connected: [4];  Timeout Counter: [0]
Finalizing setup...
Launch complete.
[0..3]Initial breakpoint, main at /lus/.../.../src/hpcc.c:18
dgb all>
```

Similarly, launching four PEs of the `hpcc_working` binary associates them with the process set `$working`. Additionally, the error tolerance level is set for the assertion scripts when comparing floating point values.

```
dbg all> launch $working{4} hpcc_working
Starting alps application, please wait...
Creating MRNet communication network...
Waiting for debug servers to attach to MRNet communications network...
Timeout in 60 seconds. Please wait for the attach to complete.
Number of dbgsrvs connected: [1];  Timeout Counter: [0]
Number of dbgsrvs connected: [1];  Timeout Counter: [1]
Number of dbgsrvs connected: [4];  Timeout Counter: [0]
Finalizing setup...
Launch complete.
[0..3]Initial breakpoint, main at /lus/.../.../src/hpcc.c:18
dbg all> set error 1.0e-14 1.0e-13 absolute
dbg all>
```

2.2.3 Define Key Data Structures

Both applications are now launched and held immediately before execution is passed to their `main()` routines. The next task is to create a decomposition scheme that will make PE comparisons of the scalar data easier. In this case, the decomposition is named `$chk1` and is defined with a total size of four data variables distributed in a block fashion over a grid of four PEs. This means that when `$chk1` is used in conjunction with a scalar variable in either of the two invoked process sets, it expects a single scalar data variable is present in each PE, because there are a total of four data variables distributed over four PEs.

```
dbg all> decomposition $chk1
> dimension 4
> distribute block
> proc_grid 4
> dim_order 1
> end
dbg all>
```

2.2.4 Employ Assertions to Compare Data Structures

Recall from Locate Entry Point into Code on page 17 that the following line of code produces different results in the two versions of the application.

```
418     resid1 = resid0 / ( TEST->epsil * ( AnormI * XnormI + BnormI ) * (double)(N) );
```

Therefore, an assertion script is built and executed to compare the variables that go into the `resid1` calculation.

```
dbg all> build $resid1
> assert $chk1{$broken::resid0@"HPL_pdtest.c":418} = $chk1{$working::resid0@"HPL_pdtest.c":418}
> assert $chk1{$broken::TEST->epsil@"HPL_pdtest.c":418} =
$chk1{$working::TEST->epsil@"HPL_pdtest.c":418}
> assert $chk1{$broken::AnormI@"HPL_pdtest.c":418} = $chk1{$working::AnormI@"HPL_pdtest.c":418}
> assert $chk1{$broken::XnormI@"HPL_pdtest.c":418} = $chk1{$working::XnormI@"HPL_pdtest.c":418}
> assert $chk1{$broken::BnormI@"HPL_pdtest.c":418} = $chk1{$working::BnormI@"HPL_pdtest.c":418}
> assert $chk1{$broken::N@"HPL_pdtest.c":418} = $chk1{$working::N@"HPL_pdtest.c":418}
> end
Assertion script $resid1 compiled.
dbg all> start $resid1
***Starting execution of applications
dbg all>
*** Difference found in AssertID:1
*** Difference found in AssertID:4

*** The interpreter has halted.
Assertion script $resid1 complete.
Successful Assertion Set Iterations: 0
Total Passed Assertions: 4
Total Warned Assertions: 0
Total Failed Assertions: 2

                                        Assertion summary:
```

```
AssertID 1:  Pass: 0 Warn: 0 Fail: 1
AssertID 2:  Pass: 1 Warn: 0 Fail: 0
AssertID 3:  Pass: 1 Warn: 0 Fail: 0
AssertID 4:  Pass: 0 Warn: 0 Fail: 1
AssertID 5:  Pass: 1 Warn: 0 Fail: 0
AssertID 6:  Pass: 1 Warn: 0 Fail: 0
**********************************

Current location:
working[0..3]: Application halted in HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:418
broken[0..3]: Application halted in HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:418
dbg all>
```

A deviation in the data is found causing the assertion interpreter to halt execution.

Note: The amount of output above is typical after an assertion run. For brevity after future runs, nonessential information will be removed.

2.2.5 Evaluate Results

After running the assertion script, `$resid1`, it is determined that variables `resid0` and `XnormI` deviate between the two applications. Therefore, it is safe to ignore the other variables that went into the calculation of `resid1` and focus on `resid0` and `XnormI`.

2.3 Comparative Debugging — 2nd Pass

Because `XnormI` deviates, an assertion script must be built to compare every variable that goes into its calculation. `XnormI` is defined in the source file `HPL_pdtest.c` as follows:

```
357     rdata->XnormI =
358       XnormI = HPL_pdlange( GRID, HPL_NORM_1, 1, N, NB, mat.X, 1 );
```

Variables `GRID`, `N`, and `NB` are straightforward to compare, but the matrix `mat.X` is a bit more complicated to compare and is done separately in the assertion script `$XnormI_mat.X`.

Note: With this preliminary release of the comparative debugging feature, it is necessary to quit `lgdb` and then restart it, in order to release the applications and associated variables, thus making it possible to relaunch the applications and run another test. For brevity, rather than include the following step in every iteration, they will simply be noted as, "Restart and Relaunch."

```
dbg all> quit
% lgdb
dbg all> launch $broken{4} hpcc_broken
dbg all> launch $working{4} hpcc_working
dbg all> set error 1.0e-14 1.0e-13 absolute
```

```
dgb all> Restart and Relaunch
dbg all> decomposition $chk2
> dimension 4
> distribute block
> proc_grid 4
> dim_order 1
> end
dbg all> build $XnormI
> assert $chk2{$broken::*GRID@"HPL_pdtest.c":357} = $chk2{$working::*GRID@"HPL_pdtest.c":357}
> assert $chk2{$broken::N@"HPL_pdtest.c":357} = $chk2{$working::N@"HPL_pdtest.c":357}
> assert $chk2{$broken::NB@"HPL_pdtest.c":357} = $chk2{$working::NB@"HPL_pdtest.c":357}
> assert $chk2{$broken::*GRID@"HPL_pdtest.c":359} = $chk2{$working::*GRID@"HPL_pdtest.c":359}
> assert $chk2{$broken::N@"HPL_pdtest.c":359} = $chk2{$working::N@"HPL_pdtest.c":359}
> assert $chk2{$broken::NB@"HPL_pdtest.c":359} = $chk2{$working::NB@"HPL_pdtest.c":359}
> end
Assertion script $XnormI compiled.
dbg all> start $XnormI
***Starting execution of application

*** The interpreter has halted. ***
Assertion script $XnormI complete.
Successful Assertion Set Iterations: 1
Total Passed Assertions: 6
Total Warned Assertions: 0
Total Failed Assertions: 0
```

There are no deviations before or after the call to `XnormI`; therefore, all of these variables can safely be ignored.

`mat.X` is the 1 by `nq` solution vector `x`. As shown in the following section of code, this points to a region inside of `mat.A` to avoid unneeded reallocation of memory.

```
187     mat.A  = (double *)HPL_PTR( vptr,
188       ((size_t)(ALGO->align) * sizeof(double) ) );
189     mat.X  = Mptr( mat.A, 0, mat.nq, mat.ld );
```

Use `lgdb` to break at line 357 (prior to the calculation of `XnormI`) and print the value of `nq`.

```
dgb all> Restart and Relaunch
dbg all> break HPL_pdtest.c:357
broken[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 357.
working[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 357.
dbg all> continue
working[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:357
broken[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:357
dbg all> print nq
broken[1,3]: 480
broken[0,2]: 520
working[1,3]: 480
working[0,2]: 520
dbg all>
```

Thinking about this in terms of the global problem, one might expect the global solution vector `x` to be 1 by 1000; however, a reconstruct of what each PE is pointing at, indicates that there is enough space for a 1 by 2000 vector. Note that `mat.X` points into the local `A` matrix; however, to compare only the bits on which `HPL_pdlange` operates (as on line 358 of `HPL_pdtest.c`), the PEs used to calculate the norm value must be determined.

The code for the function `HPL_pdlange`, shows that `HPL_NORM_1` only operates for PEs with `mp` greater than 0. The next step is to set a break at `HPL_pdtest.c:357`, continue to the breakpoint, set a breakpoint at `HPL_pdlange.c:164` (the start of the `HPL_NORM_1` calculation) and then issue a `print` on `mp`, to find the following for both `$working` and `$broken`:

```
dbg all> Restart and Relaunch
dbg all> break HPL_pdtest.c:357
broken[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 357.
working[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 357.
dbg all> continue
broken[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:357
working[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:357
dbg all> break HPL_pdlange.c:164
broken[0..3]: Breakpoint 2: file ./lus/.../src/pauxil/HPL_pdlange.c, line 164.
working[0..3]: Breakpoint 2: file /lus/.../src/pauxil/HPL_pdlange.c, line 164.
dbg all> continue
broken[0..3]: Breakpoint 2, HPL_pdlange at /lus/.../src/pauxil/HPL_pdlange.c:164
working[0..3]: Breakpoint 2, HPL_pdlange at /lus/.../src/pauxil/HPL_pdlange.c:164
dbg all> print mp
broken[2..3]: 0
broken[0..1]: 1
working[2..3]: 0
working[0..1]: 1
dbg all>
```

This means that PE 0 and PE 1 hold the actual information for `mat.X`, and only these two PEs must be compared. To do this, the dereferenced `mat.X` pointer must be cast to the proper dimension so that `lgdb` is able to grab the amount of data expected, because C language does not provide a way to determine this directly from the pointer alone.

```
dbg all> Restart and Relaunch
dbg all>  build  $XnormI_matX
> assert $broken{0}::(double[520])*mat.X@"HPL_pdtest.c":357 =
$working{0}::(double[520])*mat.X@"HPL_pdtest.c":357
> assert $broken{0}::(double[520])*mat.X@"HPL_pdtest.c":359 =
$working{0}::(double[520])*mat.X@"HPL_pdtest.c":359
> assert $broken{1}::(double[480])*mat.X@"HPL_pdtest.c":357 =
$working{1}::(double[480])*mat.X@"HPL_pdtest.c":357
> assert $broken{1}::(double[480])*mat.X@"HPL_pdtest.c":359 =
$working{1}::(double[480])*mat.X@"HPL_pdtest.c":359
> end
Assertion script $XnormI_matX compiled.
dbg all> start $XnormI_matX
***Starting execution of application
*** Difference found in AssertID:1
*** Difference found in AssertID:3
```

After running `$XnormI_matX`, it is found that `mat.X` is different **before** the call to `XnormI`; therefore, the original source of deviation must occur earlier.

2.4 Comparative Debugging — 3rd Pass

In addition to `XnormI`, `$resid0` was also found to be a deviating variable in our original calculation of `resid1`; therefore, every variable that goes into the function that calculates its value must be checked.

```
407     rdata->RnormI =
408       resid0 = HPL_pdlange( GRID, HPL_NORM_I, N, 1, NB, Bptr, mat.ld );
```

`Bptr` is a bit more complicated to compare, and is done separately in the assertion script `$resid0_Bptr`.

```
dbg all> Restart and Relaunch
dbg all> decomposition $chk3
> dimension 4
> distribute block
> proc_grid 4
> dim_order 1
> end
dbg all> build $resid0
> assert $chk3{$broken::*GRID@"HPL_pdtest.c":407} = $chk3{$working::*GRID@"HPL_pdtest.c":407}
> assert $chk3{$broken::N@"HPL_pdtest.c":407} = $chk3{$working::N@"HPL_pdtest.c":407}
> assert $chk3{$broken::NB@"HPL_pdtest.c":407} = $chk3{$working::NB@"HPL_pdtest.c":407}
> assert $chk3{$broken::mat.ld@"HPL_pdtest.c":407} = $chk3{$working::mat.ld@"HPL_pdtest.c":407}
> assert $chk3{$broken::*GRID@"HPL_pdtest.c":409} = $chk3{$working::*GRID@"HPL_pdtest.c":409}
> assert $chk3{$broken::N@"HPL_pdtest.c":409} = $chk3{$working::N@"HPL_pdtest.c":409}
> assert $chk3{$broken::NB@"HPL_pdtest.c":409} = $chk3{$working::NB@"HPL_pdtest.c":409}
> assert $chk3{$broken::mat.ld@"HPL_pdtest.c":409} = $chk3{$working::mat.ld@"HPL_pdtest.c":409}
> end
Assertion script $resid0 compiled.
dbg all> start $resid0
***Starting execution of application
*** The interpreter has halted. ***
Script $resid0 complete.
Successful Assertion Set Iterations: 1
Total Passed Assertions: 8
Total Warned Assertions: 0
Total Failed Assertions: 0
```

There are no deviations before or after the call to `$resid0`; therefore, it is safe to ignore all of these variables and move on to check `Bptr`.

```
367     Bptr = Mptr( mat.A, 0, nq, mat.ld );
```

Bptr is the global N by 1 b matrix, and also points to a region inside mat.A to avoid unnecessary reallocation of memory. The next step is to insert a break point at HPL_pdtest.c:407, continue to the breakpoint, set another breakpoint at HPL_pdlange.c:200 (found at the start of the HPL_NORM_I calculation) and then issue a print command for mp and nq.

```
dbg all> break HPL_pdtest.c:407
break HPL_pdtest.c:407
broken[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 407.
working[0..3]: Breakpoint 1: file /lus/.../src/ptest/HPL_pdtest.c, line 407.
dbg all> continue
working[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:407
broken[0..3]: Breakpoint 1, HPL_pdtest at /lus/.../src/ptest/HPL_pdtest.c:407
dbg all> break HPL_pdlange.c:200
broken[0..3]: Breakpoint 2: file /lus/.../src/pauxil/HPL_pdlange.c, line 200.
working[0..3]: Breakpoint 2: file /lus/.../src/pauxil/HPL_pdlange.c, line 200.
dbg all> continue
broken[0..3]: Breakpoint 2, HPL_pdlange at /lus/.../src/pauxil/HPL_pdlange.c:200
working[0..3]: Breakpoint 2, HPL_pdlange at /lus/.../src/pauxil/HPL_pdlange.c:200
dbg all> print mp
broken[2..3]: 480
broken[0..1]: 520
working[2..3]: 480
working[0..1]: 520
dbg all> print nq
broken[1,3]: 0
broken[0,2]: 1
working[1,3]: 0
working[0,2]: 1
```

This means that PEs 0 and 2 hold the information for Bptr, and only these two PEs need to be compared. PE 0 contains 520 elements of b, and PE 2 contains 480 elements of b. As with mat.X, the dereferenced Bptr pointer must be cast to the proper dimension so that lgdb is able to grab the amount of data expected, because C language does not provide a way to determine this directly from a pointer alone.

```
dbg all> Restart and Relaunch
dbg all> build $resid0_Bptr
> assert $broken{0}::(double[520])*Bptr@"HPL_pdtest.c":407 =
$working{0}::(double[520])*Bptr@"HPL_pdtest.c":407
> assert $broken{0}::(double[520])*Bptr@"HPL_pdtest.c":409 =
$working{0}::(double[520])*Bptr@"HPL_pdtest.c":409
> assert $broken{2}::(double[480])*Bptr@"HPL_pdtest.c":407 =
$working{2}::(double[480])*Bptr@"HPL_pdtest.c":407
> assert $broken{2}::(double[480])*Bptr@"HPL_pdtest.c":409 =
$working{2}::(double[480])*Bptr@"HPL_pdtest.c":409
> end
Assertion script $resid0_Bptr compiled.
dbg all> start $resid0_Bptr
***Starting execution of application
*** Difference found in AssertID:1
*** Difference found in AssertID:3
```

After running $resid0_Bptr it is found that Bptr is different **before** the call for resid0 and, therefore, the original source of deviation must occur earlier.

2.5 Comparative Debugging — 4th Pass

At this point, it is known that both `mat.X` and `Bptr` deviate at some point in the code; however, `mat.X` deviates at an earlier point (`HPL_pdtest.c:357`) than `Bptr` (`HPL_pdtest.c:407`). Note that, this does not imply that `Bptr` is not also deviating at the point `mat.X` was checked, but it does suggest that `mat.X` is deviating at an earlier point. Comparative debugging ignores the control flow as much as possible, and it is best practice to always try to work backwards in time as quick as possible to discover the deviation.

By examining the code, it is found that `mat.X` is originally pointed to at line 188. It appears that line 189 generates the entire `A` matrix, into which `mat.X` is pointing. The value of `mat.X` should be checked immediately after it is generated.

```
186     mat.A  = (double *)HPL_PTR( vptr,
187       ((size_t)(ALGO->align) * sizeof(double) ) );
188     mat.X  = Mptr( mat.A, 0, mat.nq, mat.ld );
189     HPL_pdmatgen( GRID, N, N+1, NB, mat.A, mat.ld, HPL_ISEED );
```

The following codes shows that the `mat` struct is being passed into the `HPL_pdgesv` function at line 200.

```
198     HPL_ptimer_boot(); (void) HPL_barrier( GRID->all_comm );
199     HPL_ptimer( 0 );
200     HPL_pdgesv( GRID, ALGO, &mat );
201     HPL_ptimer( 0 );
```

It is not known whether `mat.X` is going to be used inside `HPL_pdgesv`, but it should be checked before and after this function, just to be safe. There does not appear to be any other locations where `mat.X` is used before line 357.

```
dbg all> Restart and Relaunch
dbg all> build $pdtest_matX
> assert $broken{0}::(double[520])*mat.X@"HPL_pdtest.c":198 =
$working{0}::(double[520])*mat.X@"HPL_pdtest.c":198
>assert $broken{0}::(double[520])*mat.X@"HPL_pdtest.c":200 =
$working{0}::(double[520])*mat.X@"HPL_pdtest.c":200
> assert $broken{0}::(double[520])*mat.X@"HPL_pdtest.c":201 =
$working{0}::(double[520])*mat.X@"HPL_pdtest.c":201
> assert $broken{1}::(double[480])*mat.X@"HPL_pdtest.c":198 =
$working{1}::(double[480])*mat.X@"HPL_pdtest.c":198
> assert $broken{1}::(double[480])*mat.X@"HPL_pdtest.c":200 =
$working{1}::(double[480])*mat.X@"HPL_pdtest.c":200
> assert $broken{1}::(double[480])*mat.X@"HPL_pdtest.c":201 =
$working{1}::(double[480])*mat.X@"HPL_pdtest.c":201
> end
Assertion script$pdtest_matX compiled.
dbg all> start $pdtest_matX
***Starting execution of application
*** Difference found in AssertID:3
*** Difference found in AssertID:6
```

After running `$pdtest_matX`, it is found that lines 198 and 200 do not deviate; however, a deviation of `mat.X` is detected at line 201. Therefore, `mat.X` is deviating somewhere inside `HPL_pdgesv`, and this function must be examined more closely.

2.6 Comparative Debugging — 5th Pass

Although it is known that the call to HPL_pdgesv is causing deviation on mat.X, an important first check is to determine whether the arguments going into the function (GRID, ALGO, and mat) are matching.

```
dbg all> Restart and Relaunch
dbg all> decomposition $chk4
> dimension 4
> distribute block
> proc_grid 4
> dim_order 1
> end
dbg all> build $pdgesv_args
> assert $chk4{$broken::*GRID@"HPL_pdtest.c":200} = $chk4{$working::*GRID@"HPL_pdtest.c":200}
> assert $chk4{$broken::*ALGO@"HPL_pdtest.c":200} = $chk4{$working::*ALGO@"HPL_pdtest.c":200}
> assert $chk4{$broken::mat@"HPL_pdtest.c":200} = $chk4{$working::mat@"HPL_pdtest.c":200}
> end
Assertion script $pdgesv_args compiled.
dbg all> start $pdgesv_args
***Starting execution of application
*** The interpreter has halted. ***
Assertion script $pdgesv_args complete.
Successful Assertion Set Iterations: 1
Total Passed Assertions: 3
Total Warned Assertions: 0
Total Failed Assertions: 0
```

No differences are detected. Next, HPL_pdgesv is examined.

```
 97     if( A->n <= 0 ) return;
 98
 99     A->info = 0;
100
101     if( ( ALGO->depth == 0 ) || ( GRID->npcol == 1 ) )
102     {
103       HPL_pdgesv0(  GRID, ALGO, A );
104     }
105     else
106     {
107       HPL_pdgesvK2( GRID, ALGO, A );
108     }
109     /*
110      * Solve upper triangular system
111      */
112     if( A->info == 0 ) HPL_pdtrsv( GRID, A );
113     /*
114      * End of HPL_pdgesv
115      */
```

This is a wrapper for three function calls. The next step is to create assertions for `mat.X` at each of these. Note that this function transforms the symbolic `mat` name into `A`.

```
dbg all> Restart and Relaunch
dbg all> build $pdgesv_matX
> assert $broken{0}::(double[520])*A.X@"HPL_pdgesv.c":97 =
$working{0}::(double[520])*A.X@"HPL_pdgesv.c":97
> assert $broken{0}::(double[520])*A.X@"HPL_pdgesv.c":103 =
$working{0}::(double[520])*A.X@"HPL_pdgesv.c":103
> assert $broken{0}::(double[520])*A.X@"HPL_pdgesv.c":107 =
$working{0}::(double[520])*A.X@"HPL_pdgesv.c":107
> assert $broken{0}::(double[520])*A.X@"HPL_pdgesv.c":112 =
$working{0}::(double[520])*A.X@"HPL_pdgesv.c":112
> assert $broken{0}::(double[520])*A.X@"HPL_pdgesv.c":115 =
$working{0}::(double[520])*A.X@"HPL_pdgesv.c":115
> assert $broken{1}::(double[480])*A.X@"HPL_pdgesv.c":97 =
$working{1}::(double[480])*A.X@"HPL_pdgesv.c":97
> assert $broken{1}::(double[480])*A.X@"HPL_pdgesv.c":103 =
$working{1}::(double[480])*A.X@"HPL_pdgesv.c":103
> assert $broken{1}::(double[480])*A.X@"HPL_pdgesv.c":107 =
$working{1}::(double[480])*A.X@"HPL_pdgesv.c":107
> assert $broken{1}::(double[480])*A.X@"HPL_pdgesv.c":112 =
$working{1}::(double[480])*A.X@"HPL_pdgesv.c":112
> assert $broken{1}::(double[480])*A.X@"HPL_pdgesv.c":115 =
$working{1}::(double[480])*A.X@"HPL_pdgesv.c":115
> end
Assertion script $pdgesv_matX compiled.
dbg all> start $pdgesv_matX
***Starting execution of application
*** Difference found in AssertID:5
*** Difference found in AssertID:10
*** The interpreter has halted. ***
Assertion script $pdgesv_matX complete.
Successful Assertion Set Iterations: 0
Total Passed Assertions: 6
Total Warned Assertions: 0
Total Failed Assertions: 2

Assertion summary:
AssertID 1:  Pass: 1 Warn: 0 Fail: 0
AssertID 2:  Pass: 0 Warn: 0 Fail: 0
AssertID 3:  Pass: 1 Warn: 0 Fail: 0
AssertID 4:  Pass: 1 Warn: 0 Fail: 0
AssertID 5:  Pass: 0 Warn: 0 Fail: 1
AssertID 6:  Pass: 1 Warn: 0 Fail: 0
AssertID 7:  Pass: 0 Warn: 0 Fail: 0
AssertID 8:  Pass: 1 Warn: 0 Fail: 0
AssertID 9:  Pass: 1 Warn: 0 Fail: 0
AssertID 10:  Pass: 0 Warn: 0 Fail: 1
**********************************
```

The assertion at line 103 is never hit; therefore, it is not a part of the valid control flow for the way this code is compiled. All assertions except for line 115 match. This means that the deviation for `A.X` occurs in the `HPL_pdtrsv` function that solves the upper triangular system. At this point, the other input, `A.A`, should be checked to ensure that this is not deviating at an earlier point inside this function.

It is known that the total dimension of A is N by N+1; however, in the code's comments it states that every process holds onto an ld by nq chunk of A. For process 0, A is 520 by 521, and for process 1, A is 520 by 481. Assertions for A.A can be created in the same fashion as was done for A.X to check the A matrix at different points in the control flow. Because line 103 is never hit, this assertion can be omitted for our A matrix assertion script.

```
dbg all> Restart and Relaunch
dbg all> build $pdgesv_matA
> assert $broken{0}::(double[520][520])*A.A@"HPL_pdgesv.c":97 =
$working{0}::(double[520][520])*A.A@"HPL_pdgesv.c":97
> assert $broken{0}::(double[520][520])*A.A@"HPL_pdgesv.c":107 =
$working{0}::(double[520][520])*A.A@"HPL_pdgesv.c":107
> assert $broken{0}::(double[520][520])*A.A@"HPL_pdgesv.c":112 =
$working{0}::(double[520][520])*A.A@"HPL_pdgesv.c":112
> assert $broken{0}::(double[520][520])*A.A@"HPL_pdgesv.c":115 =
$working{0}::(double[520][520])*A.A@"HPL_pdgesv.c":115
> assert $broken{1}::(double[520][480])*A.A@"HPL_pdgesv.c":97 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesv.c":97
> assert $broken{1}::(double[520][480])*A.A@"HPL_pdgesv.c":107 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesv.c":107
> assert $broken{1}::(double[520][480])*A.A@"HPL_pdgesv.c":112 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesv.c":112
> assert $broken{1}::(double[520][480])*A.A@"HPL_pdgesv.c":115 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesv.c":115
> end
Assertion script $pdgesv_matA compiled.
dbg all> start $pdgesv_matA
***Starting execution of application
*** Difference found in AssertID:3
*** Difference found in AssertID:7
```

It is found that matrix A is deviating at line 112. This is an important result as it deviates before the X matrix and indicates that the N+1 matrix is deviating inside the call to HPL_pdgesvK2.

2.7 Comparative Debugging — 6th Pass

The call to HPL_pdgesvK2 is causing deviation to A.A only, and not to the inputs GRID, ALGO, or A. Assertions must be created at different points in the code to check A.A. At this point, this is a *guess and check* process. Assertions can be added or removed, as needed, to refine the search.

Initially the value of A.A is checked before and after panel initialization (lines 121 and 134); before and after lookahead initialization (lines 140 and 164); before and after the main loop (lines 164 and 202); and before and after cleanup (lines 202 and 210). The assertion script can be built to compare both PE 0 and PE 1, but for brevity, in this example focus is on PE 0.

```
dbg all> Restart and Relaunch
dbg all> build $pdgesvK2
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":121 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":121
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":134 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":134
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":140 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":140
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":164 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":164
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":202 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":202
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":210 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":210
> end
Assertion script $pdgesvK2 compiled.
dgb all> start $pdgesvK2
***Starting execution of application
*** Difference found in AssertID:5
```

A deviation of A.A is detected at line 202, which means the deviation occurs somewhere inside the main loop. Next an assertion script is built that looks explicitly at the main loop, picking lines 174, 183, 185, 192, and 198 for comparison locations.

```
dbg all> Restart and Relaunch
dbg all> build $pdgesvK2_main_loop
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":174 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":174
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":183 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":183
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":185 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":185
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":192 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":192
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":198 =
$working{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":198
> end
Assertion script $pdgesvK2_main_loop compiled.
dbg all> start $pdgesvK2_main_loop
***Starting execution of application
*** Difference found in AssertID:3
```

A deviation of A.A is detected at line 185. This means the deviation occurs between lines 174 and 185.

```
dbg all> build $pdgesvK2_main_loop2
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":176 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesvK2.c":176
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":177 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesvK2.c":177
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":178 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesvK2.c":178
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":179 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesvK2.c":179
> assert $broken{0}::(double[520][480])*A.A@"HPL_pdgesvK2.c":183 =
$working{1}::(double[520][480])*A.A@"HPL_pdgesvK2.c":183
> end
Assertion script $pdgesvK2_main_loop2 compiled.
dbg all> start $pdgesvK2_main_loop2
***Starting execution of application
*** Difference found in AssertID:1
```

A deviation of A.A is detected at line 177, which means the deviation occurs inside HPL_pdupdate. Note that this is a function pointer that gets set inside HPL_pdgesvK2. Its value can be determined by printing HPL_pdupdate.

```
dbg all> print HPL_pdupdate
broken[0,2..3]: No symbol "HPL_pdupdate" in current context
broken[1]: {void (*)()} 0x431c60 <HPL_pdupdateTT>
working[0,2..3]: No symbol "HPL_pdupdate" in current context
working[1]: {void (*)()} 0x431c60 <HPL_pdupdateTT>
```

This shows that HPL_pdupdate points to the function HPL_pdupdateTT.

2.8 Comparative Debugging — 7th Pass

HPL_update passes a HPL_T_panel pointer, which contains our A matrix, to HPL_updateTT. This type is defined in hpl_panel.h. The member pmat contains the local array information where the A matrix that is deviating is found. To check the A matrix, use the variable PANEL->pmat->A. The control flow gets very complicated inside this function due to the use of numerous compiler directives. An assertion can be placed inside the main blocks to determine what is called and what is not.

```
dbg all> Restart and Relaunch
dbg all> build $pdupdateTT
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":119 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":119
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":143 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":143
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":145 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":145
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":264 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":264
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":431 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":431
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":436 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":436
> end
Assertion script $pdupdateTT compiled.
dbg all> start $pdupdateTT
***Starting execution of application
*** Difference found in AssertID:5
*** The interpreter has halted. ***

Assertion summary:
AssertID 1:  Pass: 1 Warn: 0 Fail: 0
AssertID 2:  Pass: 1 Warn: 0 Fail: 0
AssertID 3:  Pass: 0 Warn: 0 Fail: 0
AssertID 4:  Pass: 1 Warn: 0 Fail: 0
AssertID 5:  Pass: 0 Warn: 0 Fail: 1
AssertID 6:  Pass: 0 Warn: 0 Fail: 0
```

Assertions on lines 119, 143, and 264 pass, but the assertion on line 431 failed. This narrows the scope to between lines 264 and 431.

```
dbg all> build $else_block
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":300 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":300
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":328 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":328
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":352 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":352
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":360 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":360
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":386 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":386
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":410 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":410
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":431 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":431
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":436 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":436
> end
Assertion script $else_block compiled.
dbg all> start $else_block
***Starting execution of application
*** Difference found in AssertID:8
*** The interpreter has halted. ***

Assertion summary:
AssertID 1:  Pass: 1 Warn: 0 Fail: 0
AssertID 2:  Pass: 0 Warn: 0 Fail: 0
AssertID 3:  Pass: 0 Warn: 0 Fail: 0
AssertID 4:  Pass: 1 Warn: 0 Fail: 0
AssertID 5:  Pass: 0 Warn: 0 Fail: 0
AssertID 6:  Pass: 0 Warn: 0 Fail: 0
AssertID 7:  Pass: 0 Warn: 0 Fail: 0
AssertID 8:  Pass: 0 Warn: 0 Fail: 1
```

Assertions on lines 300 and 360 pass; however, the assertion on line 386 fails. At this point the value of the directive HPL_CALL_VSIPL is not known. gdb will automatically assign invalid line numbers to the next valid line in the source code; therefore, it is necessary to first check higher line numbers to ensure gdb does not assign a lower number to a higher number without notification.

```
367     #ifdef HPL_CALL_VSIPL
368     /*
369      * Create the matrix subviews
370      */
371     Uv1 = vsip_msubview_d( Uv0, nq0,          0,             nn, jb );
372     Av1 = vsip_msubview_d( Av0, PANEL->ii+jb, PANEL->jj+nq0, mp, nn );
373
374     vsip_gemp_d( -HPL_rone, Lv1, VSIP_MAT_NTRANS, Uv1, VSIP_MAT_TRANS,
375       HPL_rone, Av1 );
376     /*
377      * Destroy the matrix subviews
378      */
379     (void) vsip_mdestroy_d( Av1 );
380     (void) vsip_mdestroy_d( Uv1 );
381     #else
382     HPL_dgemm( HplColumnMajor, HplNoTrans, HplTrans, mp, nn,
383       jb, -HPL_rone, L2ptr, ldl2, Uptr, LDU, HPL_rone,
384       Mptr( Aptr, jb, 0, lda ), lda );
385     #endif
386     HPL_dlatcpy( jb, nn, Uptr, LDU, Aptr, lda );
```

Start out by checking line 382 followed by the known failure at line 386.

```
dbg all> build $inner_if_block
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":382 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":382
> assert $broken{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":386 =
$working{1}::(double[520][480])*PANEL->pmat->A@"HPL_pdupdateTT.c":386
> end
Assertion script $inner_if_block compiled.
dbg all> start $inner_if_block
***Starting execution of application
*** Difference found in AssertID:2
*** The interpreter has halted. ***

Assertion summary:
AssertID 1:  Pass: 1 Warn: 0 Fail: 0
AssertID 2:  Pass: 0 Warn: 0 Fail: 1
```

The assertion on line 382 was hit and passed, but the assertion on line 386 fails. This indicates that HPL_CALL_VSIPL was not defined and the function HPL_dgemm was called. It is also known that the A matrix began deviating on the return from this call.

2.9 Comparative Debugging — 8th Pass

Now compare all scalar inputs to the HPL_dgemm function call.

```
dbg all> Restart and Relaunch
dbg all> build $dgemm
> assert $broken{1}::ORDER@"HPL_dgemm.c":467 = $working{1}::ORDER@"HPL_dgemm.c":467
> assert $broken{1}::TRANSA@"HPL_dgemm.c":467 = $working{1}::TRANSA@"HPL_dgemm.c":467
> assert $broken{1}::TRANSB@"HPL_dgemm.c":467 = $working{1}::TRANSB@"HPL_dgemm.c":467
> assert $broken{1}::M@"HPL_dgemm.c":467 = $working{1}::M@"HPL_dgemm.c":467
> assert $broken{1}::N@"HPL_dgemm.c":467 = $working{1}::N@"HPL_dgemm.c":467
> assert $broken{1}::K@"HPL_dgemm.c":467 = $working{1}::K@"HPL_dgemm.c":467
> assert $broken{1}::ALPHA@"HPL_dgemm.c":467 = $working{1}::ALPHA@"HPL_dgemm.c":467
> assert $broken{1}::LDA@"HPL_dgemm.c":467 = $working{1}::LDA@"HPL_dgemm.c":467
> assert $broken{1}::LDB@"HPL_dgemm.c":467 = $working{1}::LDB@"HPL_dgemm.c":467
> assert $broken{1}::BETA@"HPL_dgemm.c":467 = $working{1}::BETA@"HPL_dgemm.c":467
> assert $broken{1}::LDC@"HPL_dgemm.c":467 = $working{1}::LDC@"HPL_dgemm.c":467
> end
Assertion script $dgemm compiled.
dbg all> start $dgemm
***Starting execution of application
*** Difference found in AssertID:7
*** Difference found in AssertID:10
*** The interpreter has halted. ***
Assertion script $dgemm complete.
Successful Assertion Set Iterations: 1
Total Passed Assertions: 20
Total Warned Assertions: 0
Total Failed Assertions: 2
Assertion summary:
AssertID 1:  Pass: 2 Warn: 0 Fail: 0
AssertID 2:  Pass: 2 Warn: 0 Fail: 0
AssertID 3:  Pass: 2 Warn: 0 Fail: 0
AssertID 4:  Pass: 2 Warn: 0 Fail: 0
AssertID 5:  Pass: 2 Warn: 0 Fail: 0
AssertID 6:  Pass: 2 Warn: 0 Fail: 0
AssertID 7:  Pass: 1 Warn: 0 Fail: 1
AssertID 8:  Pass: 2 Warn: 0 Fail: 0
AssertID 9:  Pass: 2 Warn: 0 Fail: 0
AssertID 10:  Pass: 1 Warn: 0 Fail: 1
AssertID 11:  Pass: 2 Warn: 0 Fail: 0
```

Note that there was one successful assertion set iteration, which means that function HPL_dgemm was called, without failure at some point in the control flow, before it was called at line 382 of HPL_pdupdateTT.c. A difference between ALPHA and BETA that correspond to assertIDs 7 and 10, respectively. When this took place can be determined by issuing the backtrace (or bt) command after the script interpreter halts.

```
dbg all> bt
  broken[0,2-3]: *** program is running
  broken[1]:  0  0x0000000000042a488 in HPL_dgemm at ..src/blas/HPL_dgemm.c:467
  broken[1]:  1  0x0000000000432561 in HPL_pdupdateTT at ..src/pgesv/HPL_pdupdateTT.c:382
  broken[1]:  2  0x000000000044f69e in HPL_pdgesvK2 at ..src/pgesv/HPL_pdgesvK2.c:178
  broken[1]:  3  0x0000000000432706 in HPL_pdgesv at ..src/pgesv/HPL_pdgesv.c:107
  broken[1]:  4  0x000000000040fbce in HPL_pdtest at ..src/ptest/HPL_pdtest.c:200
  broken[1]:  5  0x000000000040a1ad in HPL_main at ..src/ptest/HPL_pddriver.c:228
  broken[1]:  6  0x0000000000402434 in main at ..src/hpcc.c:309
  working[0,2-3]: *** program is running
  working[1]:  0  0x0000000000042a488 in HPL_dgemm at ..src/blas/HPL_dgemm.c:467
  working[1]:  1  0x0000000000432561 in HPL_pdupdateTT at ..src/pgesv/HPL_pdupdateTT.c:382
  working[1]:  2  0x000000000044f69e in HPL_pdgesvK2 at ..src/pgesv/HPL_pdgesvK2.c:178
  working[1]:  3  0x0000000000432706 in HPL_pdgesv at ..src/pgesv/HPL_pdgesv.c:107
  working[1]:  4  0x000000000040fbce in HPL_pdtest at ..src/ptest/HPL_pdtest.c:200
  working[1]:  5  0x000000000040a1ad in HPL_main at ..src/ptest/HPL_pddriver.c:228
  working[1]:  6  0x0000000000402434 in main at ..src/hpcc.c:309
```

This verifies that the call to HPL_dgemm was made at line 382 of HPL_pdupdateTT.c, as expected. The values of ALPHA and BETA can be printed to see what they are currently set to in both processes.

```
dbg all> print ALPHA
broken[0,2..3]: No symbol "ALPHA" in current context
broken[1]: 1
working[0,2..3]: No symbol "ALPHA" in current context
working[1]: -1
dbg all> print BETA
broken[0,2..3]: No symbol "BETA" in current context
broken[1]: -1
working[0,2..3]: No symbol "BETA" in current context
working[1]: 1
```

Note that there is a sign difference for both. The creator of the broken code mistakenly reversed the sign for both ALPHA and BETA, which led to a deviation. If the mistake is corrected, the code recompiled and script hpcc_script_1.rc is run, the codes no longer deviate; the problem has been resolved.

Conclusion [3]

A major bottleneck in the development of high-performance applications is caused by the complexity of running applications across tens of thousands of processing cores. Although progress has been made in debuggers for parallel programs with improvements in the user interface to present application data, it is still cumbersome to isolate the source of a program bug. Comparative debugging is a methodology for debugging applications that undergo evolutionary changes such as enhancements, optimizations, porting, or running at a larger scale. Comparative debugging enables programmers to compare key data structures between two executing applications, making it possible to pinpoint the area within the application where incorrect results are first produced.

This paper demonstrated Cray's initial support of comparative debugging using `lgdb` 2.0 to debug an error within a large and complex application. Although the command-line interface is cumbersome, the basic functionality exists. In the future, Cray plans to release its comparative debugger with a GUI, simplifying and enhancing the debugging process.