Debugging

Commonly Used Options for Debugging

-O*0*
: Disables optimizations. Default is -O*2*

-g
: Produces symbolic debug information in object file (implies -O*0* when another optimization option is not explicitly set)

-traceback
: Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at runtime.
Specifying -traceback will increase the size of the executable program, but has no impact on runtime execution speeds.

-check all
: Checks for all runtime failures.
Fortran only.

-check bounds
: Alternate syntax: -CB. Generates code to perform runtime checks on array subscript and character substring expressions.
Fortran only.
Once the program is debugged, omit this option to reduce executable program size and slightly improve runtime performance.

-check uninit
: Checks for uninitialized scalar variables without the *SAVE* attribute.
Fortran only.

-check-uninit
: Enables runtime checking for uninitialized variables. If a variable is read before it is written, a runtime error routine will be called. Runtime checking of undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays.
C/C++ only.

-ftrapuv
: Traps uninitialized variables by setting any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors. This option sets -g.

-debug all
: Enables debug information and control output of enhanced debug information. To use this option, you must also specify the -g option.

-gen-interfaces
-warn interfaces
: Tells the compiler to generate an interface block for each routine in a source file; the interface block is then checked with -warn interfaces.

Options for Handling Floating-Point Exceptions

-fpe{*0*|*1*|*3*}
: Allows some control over floating-point exception (divide by zero, overflow, invalid operation, underflow, denormalized number, positive infinity, negative infinity or a NaN) handling for the main program at runtime.
Fortran only.
-fpe*0*: underflow gives 0.0; abort on other IEEE exceptions
-fpe*3*: produce NaN, signed infinities, and denormal results
Default is -fpe*3* with which all floating-point exceptions are disabled and floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. Note that use of the default -fpe*3* may slow runtime performance.

-fpe-all={*0*|*1*|*3*}
: Allows some control over floating-point exception handling for each routine in a program at runtime. Also sets -assume ieee_fpe_flags. Default is -fpe-all=*3*.
Fortran only.

-assume ieee_fpe_flags
: Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance.
Fortran only.

-ftz
: Flushes denormal results to zero when the application is in the gradual underflow mode. This option has effect only when compiling the main program. It may improve performance if the denormal values are not critical to your application's behavior. Every optimization option O level, except -O*0*, sets -ftz.

Options for Handling Floating-Point Precision

-mp
: Enables improved floating-point consistency during calculations. This option limits floating-point optimizations and maintains declared precision. -mp*1* restricts floating-point precision to be closer to declared precision. It has some impact on speed, but less than the impact of -mp.

-fp-model precise
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point

computations, but they may slow performance.

-fp-model strict
: Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

-fp-speculation=*off*
: Disables speculation of floating-point operations. Default is -fp-speculation=*fast*

-pc{*64*|*80*}
: For Intel EM64 only. Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the -pc[*n*] option. -pc*64* sets internal FPU precision to 53-bit significand. -pc*80* is the default and it sets internal FPU precision to 64-bit significand.

A segmentation fault (often called a *segfault*) can occur if a program you are running attempts to access an invalid memory location. When a segmentation fault occurs, the program will terminate abnormally with an error similar to the following message:

```
SIGSEGV: Segmentation fault - invalid memory reference.
forrtl: severe (174): SIGSEGV, segmentation fault occurred
```

The program may generate a core file, which can help with debugging.

If you use an Intel Classic or OneAPI compiler, and you include the `-g -traceback` options, the runtime system will usually point out the function and line number in your code where a segmentation fault occurred. However, the location of the segmentation fault might not be the root problem—a segfault is often a symptom, rather than the cause of a problem.

Common Segfault Scenarios

Common scenarios that can lead to segmentation faults include running out of stack space and issues resulting from bugs in your code.

Running Out of Stack Space

Stack space is a segment of program memory that is typically used by temporary variables in the program's subroutines and functions. Attempting to access a variable that resides beyond the stack space boundary will cause segmentation faults.

The usual remedy is to increase the stack size and re-run your program. For example, to set the stack size to unlimited, run:

For csh
: `unlimit stacksize`

For bash
: `ulimit -s unlimited`

On some of the NAS front-end nodes, the default stack size is set to 300,000 kilobytes (KB), while others are set with an unlimited stacksize. Use the command `limit` in csh and `ulimit -a` in bash to find the default stacksize set on the front-end you are on. On all NAS compute nodes, PBS sets the stack size to `unlimited`. However, for those compute nodes served by a front-end with a default stacksize set to 300,000 KB, if you use `ssh` to connect from one compute node to another (or several others) in order to run programs, then the stack size on the other node(s) is set to 300,000 KB.

Note: Setting the stack size to `unlimited` on a front-end node might cause problems with Tecplot. For more information, see Tecplot.

Bugs in Your Fortran Code

In Fortran programs, the most common bugs that cause segmentation faults are array bounds violations—attempts to write past the declared bounds of an array. Occasionally, uninitialized data can also cause segmentation faults.

Array Bounds Violations

To find array bounds violations, re-compile your code with the Intel Classic ifort or OneAPI ifx compiler using the `-check` (or `-check all`) option in combination with your other compiler options. When you use the `-check` option, the Fortran runtime library will flag occurrences of array bounds violations (and some other programming errors).

When the runtime library encounters the first array bounds violation, it will halt the program and provide an error message indicating where the problem occurred. You may need to re-run the code multiple times if there is more than one array bounds violation.

Note: Code compiled with the `-check` option may run significantly slower than code compiled with normal optimization (without the `-check` option).

Uninitialized Variables

You can use the `-init=keyword` option (available in the 2015 Intel Fortran compiler and later versions) to check uninitialized variables. The following keywords can be used with the `-init` option:

[no]arrays
: Determines whether the compiler initializes variables that are arrays or scalars. Specifying `arrays` initializes variables that are arrays or scalars. Specifying `noarrays` initializes only variables that are scalars. You must also specify either `init snan` or `init zero` when you specify `init [no]arrays`.

[no]snan
: Determines whether the compiler initializes to signaling NaN all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated.

[no]zero
: Determines whether the compiler initializes to zero all uninitialized variables of intrinsic type REAL, COMPLEX, INTEGER, or LOGICAL that are saved, local, automatic, or allocated.

Additional choices of keyword to set initial values include infinity, minus_infinity (valid only for floating point), tiny, minus_tiny (valid only for floating point), huge, and minus_huge.

Note: The -init compiler option does not catch all possible uninitialized variables. To find more, you can use the NAS-developed uninit tool. For information about using this tool, see the NAS training presentation uninit: Locating Use of Uninitialized Data in Floating Point Computation in Big Applications.

For more information about segmentation faults, see:

- Determining Root Cause of Segmentation Faults SIGSEGV or SIGBUS errors (Intel Developer Zone)
- Segmentation Fault (Wikipedia)

TotalView is a GUI-based debugging tool that provides control over processes and thread execution, as well as visibility into program state and variables, for C, C++ and Fortran applications. It also provides memory debugging to detect errors such as memory leaks, deadlocks, and race conditions. You can use TotalView to debug serial, OpenMP, or MPI codes.

Using TotalView to Debug Your Code

To find out which versions are available as modules, use the `module avail` command.

Before You Begin

Our current licenses allow using TotalView with MemoryScape, ReplayEngine, and CUDA support for up to a total of 256 processes on NAS x86_64 cluster systems and shared memory systems. Use the following command to find out whether there are unused licenses before you start TotalView:

```
pfe% /u/scicon/tools/bin/check_licenses -t
```

Ensure that X11 forwarding is set up on your system. Alternately, you can use the `ssh -X` or `-Y` options to enable X11 forwarding for your SSH session.

Note: If you are using a NAS-supported workstation or compute server, X11 forwarding should already be set up on your system. If the response of the GUI via X11 forwarding is slow, you will need to set up a VNC session.

Complete the following steps:

1. Compile your program using the `-g` option.
2. Start a PBS session.

   On on NAS x86_64 cluster systems:

   ```
   % qsub -I -X -q devel -lselect=2:ncpus=8:model=xxx,walltime=1:00:00
   ```

   where xxx is one of the following: `bro_ele`, `sky_ele`, `cas_ait`, `rom_ait`, or `mil_ait`.

   On Endeavour3/4:

   ```
   % qsub -I -X  -lselect=1:ncpus=28:mem=185GB:model=cas_end,walltime=1:00:00 \
   -q queue_name@pbspl4
   ```

   Note: The command line above is too long to be formatted as one line, so it is broken with a backslash (\).

3. Test the X11 forwarding with `xclock`:

   ```
   % xclock
   ```

Debugging with TotalView

Load the module:

```
% module load totalview/2023.4.16
```

The method you use to run TotalView depends on the application you want to debug.

For Serial Applications

Launch TotalView by running the `totalview` command and specifying the application:

```
% totalview ./a.out
```

Or, if your application requires arguments:

```
% totalview ./a.out -a arg_1 arg_2
```

For MPI Applications Built with HPE MPT

The old `-tv` option of HPE MPT's `mpiexec_mpt` is no longer supported (since mpt.2.23). Use the following method instead:

1. Load the latest MPT module:

   ```
   % module load comp-intel
   % module load mpi-hpe/mpt
   ```

2. Launch your program as follows:

   ```
   % totalview mpiexec_mpt.real -a -np 16 ./a.out
   ```

If the MPI launcher fails to launch the executable and recommends setting MPI_SHEPHERD=true, then set that environment variable and try running TotalView again.

For more information, see the TotalView documentation.

The GNU Debugger (GDB) is available on HECC systems in the /usr/bin directory. GDB can be used to debug programs written in C, C++, Fortran, and Modula-a.

GDB can perform four main tasks:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Be sure to compile your code with -g for symbolic debugging.

GDB is typically used in the following ways:

- Start the debugger by itself:

  % gdb
  (gdb)

- Start the debugger and specify the executable:

  % gdb *your_executable*
  (gdb)

- Start the debugger, and specify the executable and core file:

  % gdb *your_executable core-file*
  (gdb)

- Attach gdb to a running process:

  %gdb *your_executable pid*
  (gdb)

At the prompt (gdb), enter commands such as break (for setting a breakpoint), run (to start running your executable, and step (for stepping into next line). Read **man gdb** to learn more on using gdb.

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Aitken and Electra Intel processor types: Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome and Milan nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the -g option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

```
module load vtune/2021.9
```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the vtune command line that is appropriate for your code, as listed below. Use the-collect (or -c) option to run the hotspots collection and the -result-dir (or -r) option to specify a directory.

Note: The vtune command replaces amplxe-cl, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs a.out
```

Data collected by VTune for the a.out application are stored in the r000hs directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in ther000hs directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir r000hs a.out
```

Note: If your vtune run fails with MPT ERROR and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default tostdout. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, r000hs.r587i0n0 and r000hs.r587i0n1, where each directory contains 24 subdirectories, (data.[*0-23*]).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hs  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), `vtune-gui`. Since the PFEs do not have sufficient memory to run `vtune-gui`, run it on a compute node as follows:

1. On a PFE, run `echo $DISPLAY` to find its setting (for example, `pfe22:102.0`).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session (`qsub -I -X`) or a reservable front end (`pbs_rfe`).
5. On the compute node, start `vtune-gui` with name of the result directory, for example, `r000hs`.

```
compute_node% module load vtune/2021.9
compute_node% vtune-gui r000hs
```

Generating a Report

You can also use the `vtune` command with the `-report` option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to `stdout`, or to a specific output text file, such as `output.txt`:

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```

- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```

- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```

- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```

- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Gprof is a compiler-assisted performance profiler for C, Fortran, and Pascal applications running on Unix systems. You can use Gprof to help identify hotspots in your application where code optimization efforts may be most useful.

Gprof uses a hybrid of sampling and instrumentation, and provides the following information:

- The number of calls to each function and the amount of time spent there.
- Information about the caller-callee relationship.

Note: Gprof only measures the user code; it does not provide information on time spent in the kernel (such as system calls or I/O wait time).

The profiling data will be collected in a file called gmon.out, which will be generated at the end of a successful, uninterrupted run.

Gprof is available in the /usr/bin directory on all NAS compute systems. To use this tool, follow the instructions in the sections below.

Compiling and Linking to Enable Profiling with Gprof

To enable profiling with Gprof, add one of the options shown below when you compile your code:

- With Intel compilers, add the -p option (alternatively you can add -pg, which is deprecated, but still works).
- With PGI compilers, add the -pg option.
- With GNU compilers, add the -pg option. You might also have to use the -O0 option, if you do not get meaningful profiling results when using higher levels of optimization.

Collecting Profiling Data

To collect profiling data, simply run your gprof-enabled executable the same way you would run a non-gprof-enabled executable. The data will be collected in a file called gmon.out.

OpenMP Applications

If you are using an OpenMP application, gprof does not generate per-thread profiling data. Only one gmon.out file is produced.

Note: If a file named gmon.out already exists in the directory, it will be overwritten.

MPI Applications

For MPI applications, if you use the Intel MPI library, no additional steps are required before you run your code. However, if you use the HPE MPT library, you also need to set the MPI_SHEPHERD variable as follows before you run the code; otherwise, the timing information will not be shown:

```
export MPI_SHEPHERD=true (bash)
setenv MPI_SHEPHERD true (csh)
```

Each MPI process will generate a profile with the same filename, gmon.out. These files will overwrite one another when they are written to a global filesystem. Therefore, to avoid this behavior and produce a profile with a distinct filename for each process, do:

```
export GMON_OUT_PREFIX=gmon.out (bash)
setenv GMON_OUT_PREFIX gmon.out (csh)
```

For example:

```
#PBS ...

module load comp-intel
module load mpi-hpe/mpt
export MPI_SHEPHERD=true
export GMON_OUT_PREFIX=gmon.out

mpiexec a.out
```

This operation will generate a file called gmon.out.*pid*, where *pid* is the process ID of an MPI process. With *N* ranks, you should get *N* such files, with filenames differing only in their *.pid* extensions. You can then analyze an individual gmon.out.*pid* file or several at the same time.

Generating ASCII Gprof Output

You can use the gprof command to convert binary data in gmon.out into a human-readable format.

```
gprof [options] executable_name [gmon.out] [ > analysis.output ]
```

There are two types of output: *flat profile* and *call graph.*

The flat profile shows how much time your program spent in each function, and how many times that function was called. This profile helps to identify hotspots in your application. Hotspots are shown at the top of the flat profile.

The call graph shows, for each function, which functions called it; which other functions it called; and how many times the calls occurred. The call graph also provides an estimate of how much time was spent in the subroutines of each function, which can suggest places where you might try to eliminate function calls that use a lot of time.

Commonly Used gprof Command Options

Some common options are described here. Read **man gprof** for more information.

- -p prints a flat profile. For example:

  ```
  gprof -p a.out gmon.out.1001 gmon.out.1002
  ```

- -q prints a call graph. For example:

  ```
  gprof -q a.out gmon.out.*
  ```

- -s sums up the information from multiple profiling data files and produces a file called `gmon.sum` for analysis. For example:

  ```
  gprof -s a.out gmon.out.*
  gprof a.out gmon.sum > analysis.output
  ```

- -b omits verbose texts that explain the meaning of all of the fields in the tables.