Understanding and Troubleshooting HPCTOOLKIT Problems

The HPCTOOLKIT Team

24 March 2009

`hpcviewer` runs glacially slow, what is the workaround?

There are three likely reasons why `hpcviewer` might run terribly slow. You may be running `hpcviewer` on a remote system with low bandwidth, high latency or an otherwise unsatisfactory network connection to your desktop. If any of these conditions are true, `hpcviewer`'s otherwise snappy GUI can become sluggish if not downright unresponsive. The solution is to install `hpcviewer` on your local system, copy the database onto your local system, and run `hpcviewer` locally. We almost always run `hpcviewer` on our local workstations or laptops for this reason.

Another reason could be that the `experiment.xml` file in your database is very large. If this is tens of megabytes or more, the total database size might be the problem.

This problem can also occur if the database contains too many columns of metrics. This can happen if you use `hpcprof` to build a database for several threads with several metrics each, resulting in too many metrics total. You can check the number of columns in your database by running

```
grep -e "<Metric" experiment.xml | wc -l
```

If that command yields a number greater than 30 or so, `hpcviewer` is likely slow because you are working with too many columns of metrics. In this case, run `hpcprof` to build a database with fewer threads.

Why isn't my source code found by `hpcviewer`?

When using `hpcprof` to create performance databases, one must specify the path to the source directories using the `-I dir` flag. If there are multiple source directories, then multiple `-I` flags are required:

```
-I dir1 -I dir2 ... -I dirN
```

The specified list of directory paths, `dir1` through `dirN`, can be either relative or absolute.

In addition, `hpcprof` has a recursive directory search feature that can greatly reduce the number of `-I` flag instances. To conduct a source-file search of a directory *and all of it's decendants*, simply append an escaped '*' after the last slash, e.g., `/mypath/\*` (or `/mypath/'*'`). This greatly simplifies the `hpcprof` command line for a typical project. For example, suppose a

project has all of it's source in a top-level directory called `src`. Inside `src` are several libraries (like `src/lib1`,`src/lib2`, `src/lib-math`, etc). Then, all of the source is visible to `hpcprof` by using the single flag

```
-I src/'*'.
```

NOTE: The `'*'` can be used *only* at the end of a directory path.

For any functions whose source code is not found in the specified directories (e.g., system libraries), `hpcviewer` will generate a synopsis that shows the presence of the function and its line extents (if known), but no source code.

I tried the approach above, but I am still missing source files.

This is a common problem. The cause for this difficulty is that the pathnames associated with performance metrics are extracted from an application binary. The path names encoded in the binary can be

- relative to the directory where the source was compiled, not where you are now as you measure and analyze the code, or
- absolute paths that have encodings for the mount point the directory that existed when the file was compiled, but might be different now (e.g., they were compiled on a different node of a cluster that had a different mount point for the shared file system).

Diagnosing and fixing this problem requires knowing exactly what path names are referenced in the binary and/or perhaps the performance data. Fortunately, this is information is supplied by `hpcprof`. If a source file is sucessfully located, then a

```
msg:   cp:...
```

line appears in the output of `hpcprof`. Unlocated files are deemed 'lost' and there is an output line of the form

```
WARNING:  lost:
```

in the output.

For example, suppose we have an application `app1` whose main source is in in a directory `/projs/Apps/app1-src`. The `app1` application is built inside the `app1-src` subdirectory, and it uses source files from a subdirectory `app1-src/special` as well as some source common to all applications, located in `/projs/Apps/common`. When `app1` is built, the `common` source is accessed by relative path `../common`. The `app1` executable is installed on our path.

Now, we switch to our home directory `/h/user/T1` to collect some profile data for `app1`. When we run `hpcstruct` (without the `-I` flag) as follows:

```
hpcstruct -S app1.hpcstruct */*.hpcrun
```

This results in the output

```
msg: Line map : /opt/apps/intel/compilers/10.1/lib/libimf.so
msg: STRUCTURE: /usr/local/bin/app1
msg: Copying source files reached by PATH option to /h/user/T1/hpctoolkit-app1-database
WARNING: lost: app1.c
WARNING: lost: special/xfn1.c
WARNING: lost: ../common/mathx.c
WARNING: lost: ~unknown-file~
WARNING: lost: irc_msg_support.c
```

The `WARNING: lost:` obtains for `~unknown-file~` and `irc_msg_support.c` because these are compiler system files — source is unavailable. The other lost files, however, can be found by using the proper `-I` flag:

```
hpcstruct -I /projs/Apps/'*' -S app1.hpcstruct */*.hpcrun
```

The resulting output:

```
msg: Line map : /opt/apps/intel/compilers/10.1/lib/libimf.so
msg: STRUCTURE: /usr/local/bin/app1
msg: Copying source files reached by PATH option to /h/user/T1/hpctoolkit-app1-database
msg:   cp:/projs/Apps/app1-src/app1.c -> ./projs/Apps/app1-src/app1.c
msg:   cp:/projs/Apps/app1-src/special/xfn1.c -> ./projs/Apps/app1-src/special/xfn1.c
msg:   cp:/projs/Apps/common/mathx.c -> ./projs/Apps/common/mathx.c
WARNING: lost: ~unknown-file~
WARNING: lost: irc_msg_support.c
```

Much better!

Best Practice: First, carefully inspect the output of `hpcprof` to determine which files are lost. Next, determine the absolute path for each distinct top-level source directory (or build directory, if it is separate fromt the source directory). Finally, for each of these (absolute) directory paths, specify a `-I` option with the recursive search option (`'*'` at the end of the path).

Why don't the line numbers for loops and/or procedures exactly correspond to what I see in my source code?

To use a cliché, "garbage in, garbage out". HPCTOOLKIT depends on information recorded in the symbol table by the compiler. Line numbers for procedures and loops are inferred by looking at the symbol table information recorded for machine instructions identified as being inside the procedure or loop.

For procedures, often no machine instructions are associated with a procedure's declarations. Thus, the first line in the procedure that has an associated machine instruction is the first line of executable code.

Inlined functions may occasionally lead to confusing data for a procedure. Machine instructions mapped to source lines from the inline function appear in the context of other functions. While `hpcprof`'s methods for handling inline functions are good, some codes can confuse the system.

For loops, the process of identifying what source lines are in a loop is similar to the procedure process: what source lines map to machine instructions inside a loop defined by a backward

branch to a loop head. Sometimes compilers sometimes dont properly record the line number mapping.

When the compiler line mapping information is wrong, there is little you can do about it other than to ignore its imperfections, or hand-edit the XML program structure file produced by `hpcstruct`. This technique is used only when truly desparate.

A particular scope in my code (*e.g.*, a loop) contains one call to a function, yet `hpcviewer` shows several. Is something wrong with HPCToolkit?

In a word: no. In the course of code optimization, compilers often replicate code blocks. For instance, as it generates code, a compiler may peel iterations from a loop or split the iteration space of a loop into two or more loops. In such cases, one call in the source code may be transformed into multiple distinct calls that reside at different code addresses in the executable.

When analyzing applications at the binary level, it is difficult to determine whether two distinct calls to the same function that appear in the machine code were derived from the same call in the source code. Even if both calls map to the same source line, it may be wrong to coalesce them; the source code might contain multiple calls to the same function on the same line. By design, HPCTOOLKIT does not attempt to coalesce distinct calls to the same function because it might be incorrect to do so; instead, it independently reports each call site that appears in the machine code. If the compiler duplicated calls as it replicated code during optimization, multiple call sites may be reported by `hpcviewer` when only one appeared in the source code.