Cluster:Gprof

From Earlham Cluster Department

Contents

Using gprof

Overview

gprof is a statistical profiler, that is it derives the information it provides by recording the address in the program counter at regular intervals over the course of a run and then reconstructing the call tree and displaying the results. gprof is part of the GNU toolchain and depends on gcc and ld to create a binary with particular characteristics.

gprof is good for finding the 20% of your program where 80% of the time is being spent. It's hard to directly pinpoint particular lines with gprof, or why the code is spending so much time there (I/O, long instruction stream, etc.), but you will know exactly where to concentrate during a more detailed analysis.

Using gprof is a three step process, first the binary is compiled and linked with gcc using options that signal the runtime to collect statistical information. Then the program is run one or more times, each time creating a gmon.out file with run-time information. gprof is then run against the gmon.out file(s) producing one or more reports of the runtime behavior.

gprof is part of the GNU toolchain. More information about gprof and the GNU project is available at http://gnu.org.

Basic Recipe - Serial Code

These commands should work on any "modern" Linux distro. In all the example command lines the $ is not part of the command, it's the indicator that these are to be typed at a shell prompt.

1) Compile your program with the `-O0 -pg -g` options. Disabling optimization (`-O0`) helps to insure that the gprof output can be reasonably understood in the context of your program's source code.

```
$ gcc -O0 -pg -g ... -o <binary-file-name>
```

2) If you link your program separately from compiling it be sure to include the `-pg` option there as well:

```
$ ld -pg ... -o <binary-file-name>
```

3) Run your program, which will create gmon.out. Your program must terminate with `exit(0)` or `return(0)` for the gmon.out file to be created. 4) Run gprof using the gmon.out file from the previous step as input:

```
$ gprof --line --flat-profile  <binary-file-name> gmon.out
```

A brief summary of the command line options:

- `--line` - enables line-by-line profiling
- `--flat-profile` - print a flat profile, this is the easiest of the gprof outputs to understand
- `--annotated-source` - print annotated source code
- `--graph` - causes the call graph of the program to be augmented with data from the text space of the object file

Line-by-line profiling tends to magnify statistical errors if there isn't enough run-time data. If you use this option be sure you are collecting enough data by having sufficiently large input to your program. gprof supports many options, read the man page for more information about them.

Advanced Recipe - Serial Code

If the nature or size of the input changes the runtime characteristics of your code in substantive ways, you can run it many times using a range of different inputs, saving each gmon.out file as it is created, and then have gprof analyze the whole lot of them in one fell swoop. In this mode gprof will combine the data in the run output files into one set of reports showing the sum of the runtime behaviors.

1) Compile and link your program as above.
2) Run your program say 10 times, using different input and renaming the gmon.out files between each iteration (this for loop syntax assumes the Bash shell under Linux):

```
$ for x in {1..10}; do ./matmul-blocked --infile matmul-$x.dat; mv gmon.out gmon.out.$x; done
```

3) Run gprof using the gmon.out.* files from the previous step as input:

```
$ gprof --line --annotated-source --flat-profile --graph <binary> gmon.out.*
```

This technique can also be useful with repeated runs with the same input stream, or a variety of input streams repeated many times. In both cases more data collected yields a more accurate analysis.

Basic Recipe - Parallel MPI Code

gprof can be used to analyze parallel as well as serial programs. In order to keep the MPI processes from clobbering each others gmon.out files we can use a feature which allows us to add a prefix to the output file. By including the statement below in the rc file for your shell, e.g. `.bashrc` or `.cshrc` the runtime environment will create unique output file names for each MPI process.

```
$ export GMON_OUT_PREFIX='gmon.out-'`echo $OMPI_COMM_WORLD_RANK`  # BASH syntax
```

Note that only OpenMPI supports the `OMPI_COMM_WORLD_RANK` environment variable. Other MPI bindings may support something similar.

1) Modify the appropriate .<shell-rc-file> to include the environment variable definition illustrated above.
2) Compile and link your program as described above.
3) Run your program.

When the run is complete you will have a group of gmon.out.MPI_RANK files which you can analyze using the technique described in the Advanced Recipe above. Depending on the parallel architecture of your code you may want to analyze rank 0 separately from the other ranks, for example if you have client/server structure where very different code paths are used by the server and the clients.

More Information

The man page is worth reading, it's not particularly long and includes a detailed description of how gprof works.

gcov, also part of the GNU toolchain, analyzes runtime data for code coverage. Using gcov and gprof together can yield additional insight into the software under examination.

There are many statistical and statement level profilers available. You should almost always start with whatever tool(s) are available with the development toolchain you are using; GNU, Intel, Portland Group, etc. all provide profilers for use with their kits.

Module Metadata

- software-type profiler
- software-package gprof
- software-platform Linux-2.6.9-34-x86_64-GNU/Linux-RHEL_AS4, Linux-generic
- module-version 1.0