Gprof2tree

Huub van Dam, CCLRC Daresbury Laboratory, January 2006

Contents

Quick reference

This section provides a quick reminder to the expert user as to the syntax and meaning of the supported options.

```
gprof2tree [--parents] [--children] [--[no]sort]
[--once|--always]
[--exclude "<file1>[:<file2>[:<file3>]]"]
[--stopat "<file4>[:<file5>[:<file6>]]"]
[--to "<routine name1>[ <routine name2>[ <routine name3>]]"]
<subroutine name>
```

`--parents`	Print the call moving upwards. I.e. print the routines that call `<subroutine name>`.
`--children`	Print the call moving downwards. I.e. print the routines that routine `<subroutine name>` calls. This is the default.
`--sort`	Alphabetically sort the routine names in the call tree. This is useful if call trees of slighty different runs have to be compared as it reduces the differences introduced by changes of CPU usage.
`--nosort`	Print the routines in the call tree in the same order as found in the `gprof` call tree information. Usually this means that the routines with the highest CPU usage come first. This is the default.
`--once`	Print the sub-tree for every subroutine only once, i.e. the first time it is encountered. In subsequent encounters only the call is printed. This helps to greatly reduce the length of the printed call trees and is therefore the default.
`--always`	Always print the tree of every subroutine encountered limited only by the `--stopat` and `--exclude` options in effect.
`--exclude`	Removes all the routines that are named in the specified files from the call tree. This is useful in larger codes where there can a lot of routines that are of no particular interest.
`--stopat`	Removes the call tree information of the routines that are named in the specified files. The routine names themselves are still printed in the call trees but not the routines they call. This is particularly useful in larger codes which may have deep call trees. In which one may want to know where a routine is called but not how it goes about doing its job.
`--to`	Prints only the paths through the code that lead from the desired subroutine to the routines named in the list. All other paths are suppressed.
`--level`	Print the depth level of a routine in the call tree on every line.

User guide

What gprof2tree is for

Gprof2tree aims to provide a tool for analyzing code structure using widely (and freely) available software. Relying on data as produced by the gprof program sections of the call tree can be extracted in plain text. Several mechanisms are provided to select the sections of interest, and several printing options are provided for example to aid in comparing different pathways through a code.

What gprof2tree can and cannot do

Because the input data is provided by gprof the call trees reflect only the pathways through a code that are actually executed. For large codes this helps focusing on the parts relevant to a run, i.e. the ones that are actually executed. However it also means that the nothing can be learned about routines that are not executed. Therefore gprof2tree does not help answering questions like finding all invocations of a particular routine anywhere in a code. Only the ones that are actually executed in a run can be found.
Gprof2tree allows the call tree to start from any routine in the tree. The tree can be traversed downwards from a parent routine to its children and its children's children and so on. Alternatively the tree can be traversed upwards from a child routine to its parents and its parent's parents and so on.
To reduce the size of the printed call tree there are a number of options to terminate branches at selected points or to print selected branches only.

Gprof2tree's requirements and recommended companion software

As gprof2tree relies on the data produced by gprof the obvious requirement is the availability of a compiler that produces an executable that will write a gmon.out file and the availability of gprof itself. In practice this means that gprof2tree can be used in conjunction with most compilers. The only compilers I have encountered that will not generate code for writing a gmon.out file are the once from the Portland Group suite of compilers (i.e. pgcc, pgf77, pgf90).
Gprof2tree is implemented in the Python scripting language. Python is available for many platforms from http://www.python.org .
To compare call trees, for example ones produced from different runs of the same code to find differences in the executed pathways, a graphical diff program is recommended. A good and freely available diff program tkdiff is provided with tkCVS (http://www.twobarleycorns.net/tkcvs.html, http://sourceforge.net/projects/tkcvs/). It is

advised to activate an option to ignore differences in white-space. TkCVS does require Tcl/Tk (http://www.tcl.tk) to run.

Call tree generation

Generating a basic call tree

To explain the basics of this call tree generation tool it makes sense to look at a simple example program. In later sections we will consider real applications to demonstrate the strengths of the more advanced options but this would be too cumbersome as an introduction. The sample program we will consider is a Fortran77 program given in Box 1. Note though that gprof2tree will work for

```
      program simple
      call sub_a
      end
      subroutine sub_a
      call sub_b
      end
      subroutine sub_b
      call sub_c
      end
      subroutine sub_c
      write(*,*)"Hello from sub_c"
      end
```

Box 1: The source code of the program simple kept in the file simple.f

programs written in any other language that can be compiled to generate a gmon.out file. To compile the program and produce a gprof output that serves as the basis for gprof2tree to work on execute the commands in Box 2.

```
      g77 -c -pg simple.f
      g77 -pg simple.o -o simple
      simple
      gprof simple gmon.out > gprof.out
      gprof2tree MAIN__ < gprof.out > simple_calltree.out
```

Box 2: The commands needed to produce a file gprof.out that will be the main input to gprof2tree.

In this example I used the GNU Fortran77 compiler. In practice many other compilers could be used and in most cases the `-pg` flag can be used to instruct the compiler to instrument the code to produce the `gmon.out` file. Note that the `-pg` flag typically has to be provided to the linker (the second invocation of `g77`) as well to link in the code that actually writes the `gmon.out` file.
Running the executable simple produces the file `gmon.out`. This file contains binary timing data that is transformed into legible text by the program `gprof`. In order for `gprof` to be able to associate the timing data with the correct routines you need to pass it both the executable and the `gmon.out` files as input. The output is written to `gprof.out` which serves as the input to `gprof2tree` in the final command. Here I have asked `gprof2tree` to print the call tree starting from the routine `MAIN__` . This name is the symbol that `g77` has assigned to the routine that starts with the `program` statement in the Fortran code. This name is generally compiler dependent as are any trailing underscores or prefixed periods (".") with other routine names. Therefore it is advisable to look at the contents of `gprof.out` before using `gprof2tree` for the first time to see what the compiler has made of your subroutine names.
The call tree as produced by `gprof2tree` was stored in `simple_calltree.out` the contents of which are shown in Box 3. It should come as no surprise that it shows the program calling subroutine `sub_a`, calling subroutine `sub_b`, and so on. Of course there is no requirement to start printing the call tree at the top level `MAIN__` . `Gprof2tree` can print the call tree of every routine you wish.

```
MAIN__
  sub_a__
    sub_b__
      sub_c__
```

Box 3: The contents of simple_calltree.out.

Instead of looking at which routines are called by a given routine, i.e. printing the children of the routine, it may be useful to find out where a given routine is called from, i.e. printing its parents. In other words one might be interested in printing the call tree from the bottom up instead of from the top down. The flag `--parents` allows you to do that as shown in Box 4.

```
gprof2tree --parents sub_c__ < gprof.out
```

```
sub_c__
  sub_b__
    sub_a__
      MAIN__
```

Box 4: The use of the `--parents` flag (top) reverses the order of the call tree as shown above.

Pruning the call tree

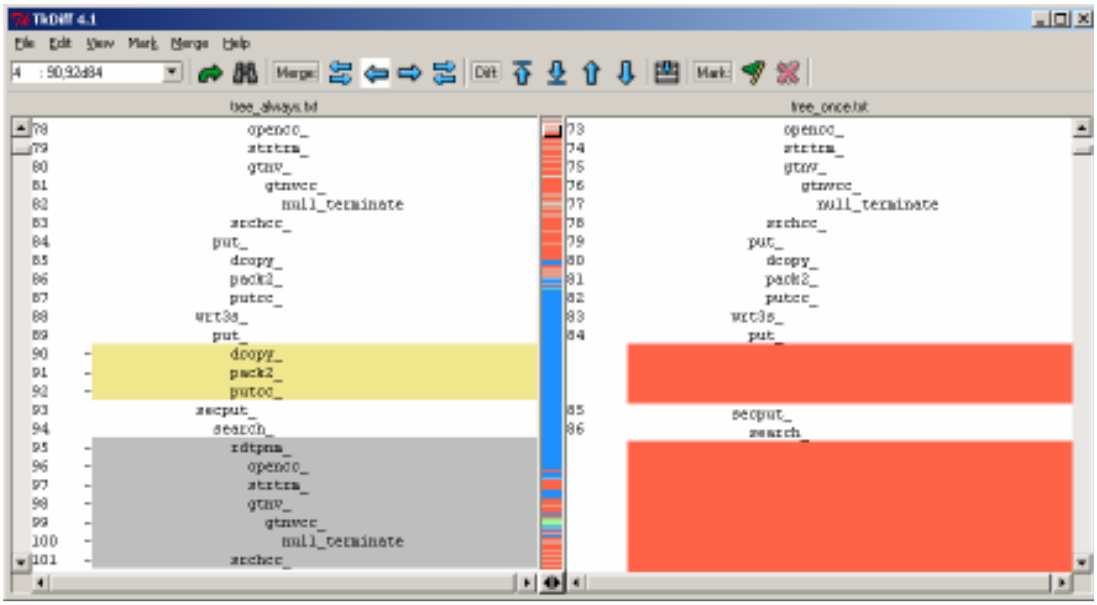
The basic problem with call trees is not the lack of data but the opposite of that. Even for modest programs the call tree quickly grows to a size that defies comprehension. Therefore any mechanisms to help prune the call tree and focus on the relevant parts in a given context are crucial. To illustrate these mechanisms I will use examples from the quantum chemistry package `GAMESS-UK` [1](http://www.cfs.dl.ac.uk). This program is mostly written in Fortran77 but some parts that are closely linked to the operating system such as memory allocators, timers and file access have been written in C. In total the program comprises approximately 1 million lines of code and almost 8000 subroutines. The program provides a wide variety of quantum chemistry methods, ranging from a low to a high algorithmic complexity.

A very powerful pruning aspect of `gprof2tree` is of course that the base data only includes those routines that are actually executed in a given run. This typically eliminates the majority of the routines. Even so, the full call tree of the simplest calculation run with `GAMESS-UK` produces a call tree of some 3746 lines, which corresponds to more that 50 pages of text assuming some 65 lines fit onto a page. Clearly, reducing this is essential if one tries to understand the code.

The simplest way to reduce the call tree is of course to choose a suitable top level routine to print the call tree of. The deeper this routine sits in the call tree the smaller its call tree will be. Although this obvious approach can easily reduce the call tree by a factor 2 or 3 the remaining data is still far too much.

Another way to drastically reduce the call tree is to exploit the fact that many routines are invoked multiple times during a run. In the call tree, strictly speaking, the structure of every routine has to appear only once. The options `--always` and `--once` control this, where the first requests the structure of a subroutine to be printed in full in every instance, the second requests the structure to be printed only once. Using the latter option on the example mentioned above reduces the call tree from 3746 lines to only 906 lines, as illustrated in Box 5.

```
gprof2tree --always MAIN__ < gprof.out > tree_always.txt
gprof2tree --once   MAIN__ < gprof.out > tree_once.txt
```



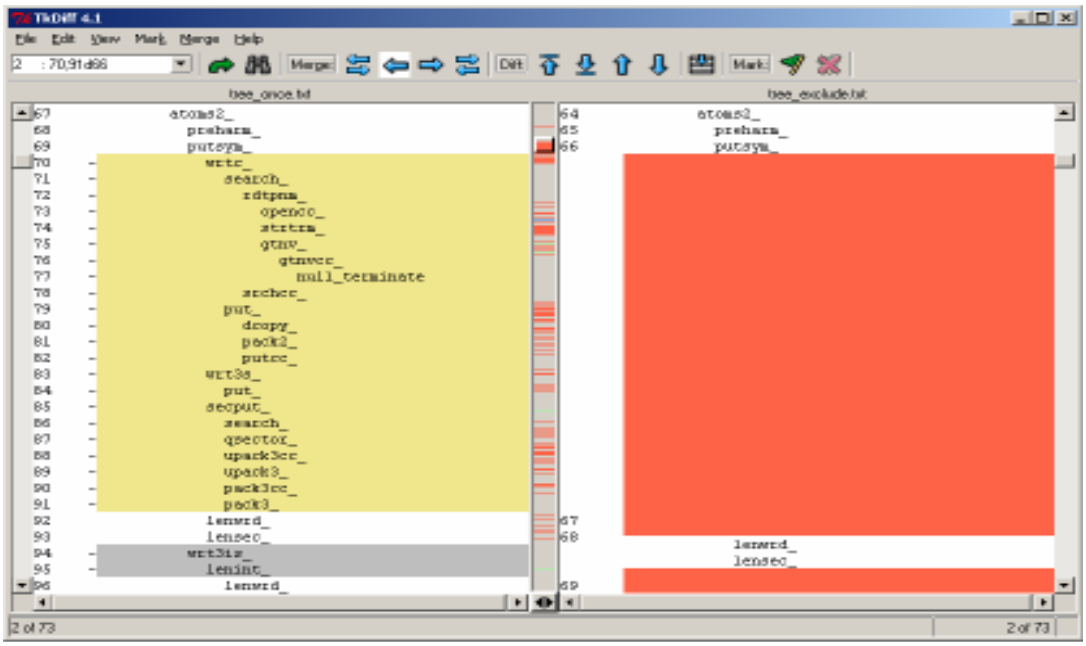
Box 5: The use of the `--always` and `--once` flags, note how in `tree_always.txt` the structure of `put_` appears twice where in `tree_once.txt` the second instance is suppressed

The above option achieves a significant reduction but the resulting 906 lines still equate to 14 pages.

The next step is based on the realization that many large scale programs rely on using libraries. Here a library is considered a collection of routines whose functionality is closely linked. For example, a program may use a collection of routines related to file I/O. These libraries may be an integral part of the program or they might be third party libraries. The point is that most often when a code uses a library you are actually not interested how this library works. So a mechanism that identifies named routines as library routines to `gprof2tree` could be used to stop it from printing the internal structure of those routines. As libraries often contain many routines I chose to have a mechanism where one would store all the subroutine names of a single library in a file. The option `--stopat` followed by a colon separated list of file names instructs `gprof2tree` to terminate the call tree at each of the routines mentioned in each of the files. Taking this one step further the option `--exclude` will remove each of the mentioned subroutines altogether. To illustrate the use of these options lets assume we want to suppress printing the structure of the file I/O routines. In a file called `io` we store the names of the various routines such as `rdedx_`, `rdchr_`, `wrtc_`, `wrt3_`, etc. Running `gprof2tree` with the `--stopat` option in combination with the `--once` option reduces the call tree to 860 lines. Using the `--exclude` option instead the call

tree is reduced to 739 lines, see Box 6. This is a reasonable result given that the `io` library contained only 16 routines, the structure of which was printed at most once due to the `--once` option. The impact of this option can of course be increased by including more and more libraries.

```
gprof2tree --once MAIN__ < gprof.out > tree_once.txt
gprof2tree --once --exclude io MAIN__ < gprof.out >
tree_exclude.txt
```



Box 6. The effect of the `--exclude` option, note how the printing of the structure of the routines `wrtc_`, `wrt3s_` and `secput_` has been suppressed.

After having reduced the call tree to a comprehensible size of perhaps a few pages only, one might be able to identify a few key routines. Focussing only on these might well condense the call tree as much as possible without losing any relevant information. The option `--to` allows to do just this. It takes a space separated list of subroutine names and prints all the pathways that lead from the top level routine to one of the routines in the list. Everything else is suppressed. When this is applied for example to answer the question where symmetry related subroutines are used in the code the call tree is reduced to just 32 lines as shown in Box 7.

```
gprof2tree --to "ssymb_ symm_ setsym_ symh_ symass_
symsvq_ symana_ symsrt_ sym1e_ symtrn_ symtrv_ symtrd_"
MAIN__ < gprof.out > tree_sym.txt
```

```
MAIN__
  driver_
    start_
      rdgeom_
        sget_
          ssymb_
      symm_
        setsym_
    scfgvb_
      hfscf_
        scf_
          rhfclm_
            symh_
            symass_
              symas2_
                symsvq_
                symana_
                symsrt_
          denscf_
            symh_
        standv_
          stvstv_
            sym1e_
      hfprop_
        lowdin_
          symtrn_
          symtrv_
```

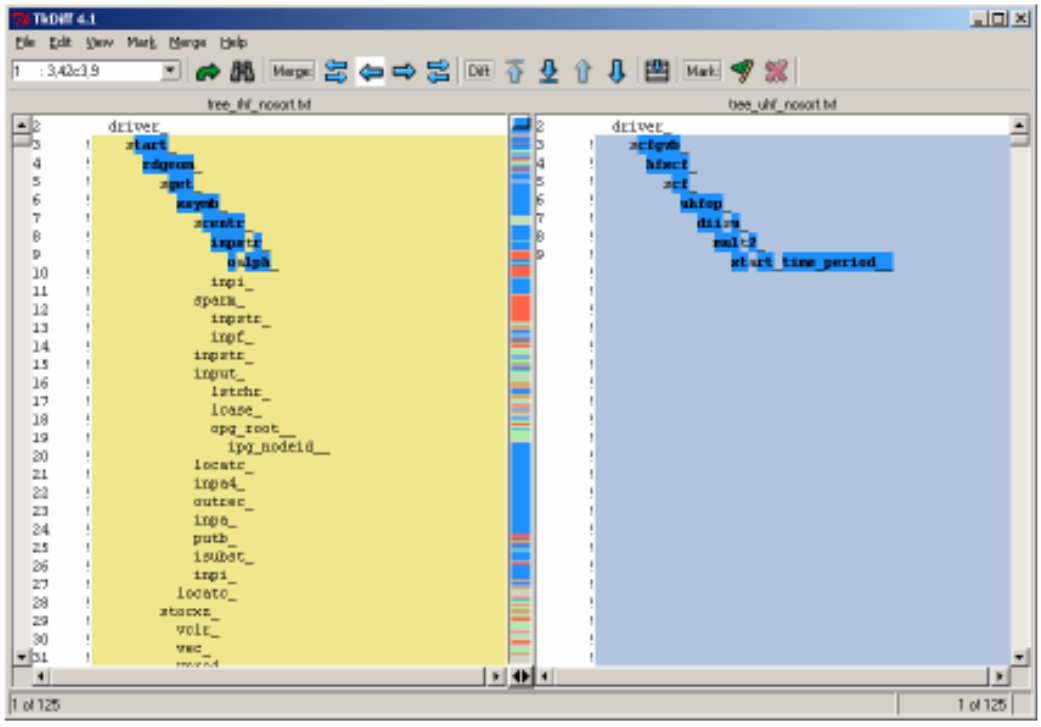
Box 7: The use of the `--to` flag to limit the call tree to selected branches only.

The above presents how a call tree can be condensed to focus on the most relevant parts. However perhaps even more important is the comparison of call trees of slightly different runs of a program. This gives more insight in how certain options change the pathway through the code. Options to help with this analysis are given below.

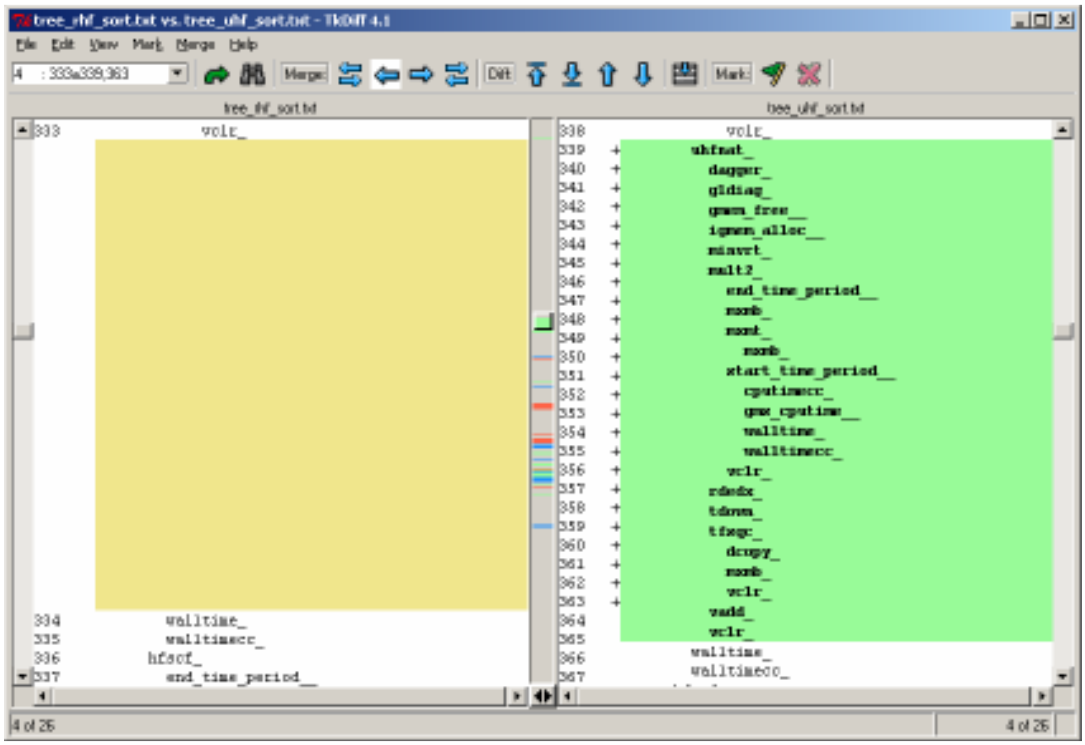
Call tree printing

Ordering the branches for code comparison

Personally I think `gprof2tree` is most useful for comparing call trees of calculations run with different options. Using `tkdiff` the differences in the pathway through the code are highlighted straightaway helping to focus on how a particular option is implemented. For this to work the call trees produced by 2 calculations have to be fairly similar on the whole. In the data produced by `gprof` the subroutines are listed in the order of decreasing CPU time usage. This order can differ wildly depending on the options given to the program or the size of the calculation, causing problems in the comparison. A simple solution to this problem is to sort the list of parents or children of every subroutine in alphabetical order. The option `--sort` will request this sorting to take place, the option `--nosort` explicitly requests this sorting to be omitted. The latter may be desirable if you want to focus on performance issues rather than algorithm comparison. The effect of the `--sort` option is illustrated in Box 8a,b.



Box 8a: The call tree comparison obtained with `--nosort`.



Box 8b: The same call tree comparison obtained with `--sort`.

From the above example it is clear that `tkdiff` is much more successful identifying the crucial differences in the code when the call tree is sorted than when its not.

Options to aid the interpretation

The last option to consider in this section is most useful for large call trees. In particular, when printing large call trees it can be difficult to match up the depth of the tree. The chances are that you misjudge the level of indentation and hence misinterpret the structure of the code. To counter this problem the option `--level` is provided to request the depth level of the call tree to be printed as shown in Box 9. Clearly this option is useful only if you want to store a non-trivial call tree for future reference.

```
gprof2tree --level MAIN__ < gprof.out
```

```
1 MAIN__
2   sub_a__
3     sub_b__
4       sub_c__
```

Box 9: The use of the `--level` flag to print the depth levels in the call tree.

Implementation Guide

Motivation

The creation of this little script was motivated by the need to quickly familiarize myself with a number of quantum chemistry codes including `CADPAC`, `HONDO`, and `NWChem [2,3]` in the course of a project. Specifically I needed a quick way of pin-pointing the particular parts of each of these codes that were involved in performing certain calculations.

Requirements

There were many requirements the most important of which were:

- Platform independence requiring both the tool as any other utilities to be used in conjunction with the tool to run on any platform that we support including but not limited to `Linux`, `MacOS`, `Windows` and any of the commercial `UNIX`'s.
- Simplicity of porting to all the above platforms.
- The production of text based call trees that can be stored in a revision control system for future reference.
- The ability to work on large codes without excessive overheads.
- The availability of options to effectively reduce the size of call trees to focus on the relevant parts.

Limits

The implementation of this tool was limited in several ways:

- All the data involved obtained directly from a `gprof` output and kept in memory. No data is kept between runs.
- No time was allocated for the creation of this tool in the project plan, so it had to be written within the time that one could expect it would save. This severly

limited the options available. All the input data as well as the capability to analyse the output had to be available from existing tools.

Outline of the implementation

The essential idea behind the implementation is the recognition that three things need to happen:

1. The call tree information needs to be extracted from the `gprof` output.
2. The data needs to be massaged in certain ways to effect the various options whether they be sorting the subroutine names or limiting the amount of data being printed.
3. Finally the call tree needs to be written out.

The main reason for separating steps 2 and 3 was to achieve a high efficiency. In step 3 one particular subroutine may be encountered many times. Instead of performing various operations at that stage they can be performed in step 2 where every subroutine is encountered only once. This way the scalability of the script as a function of the size of the codes being analysed is improved.

Reading the gprof data into tables

The `gprof` data is being read and parsed by the `python` function `parse_call_tree`. This function stores the data in three dictionaries; `parents`, `children` and `times`. The subroutine names are used as keys in the first two dictionaries. For each subroutine they hold a list of parent subroutines or child subroutines. The dictionary `times` uses tuples of subroutine names and child names as keys. For each subroutine-child tuple it holds a tuple of cpu-time and call-counts. Special arrangements have been made for routines that are called from uninstrumented routines which are marked by `gprof` as having a parent "`<spontaneous>`", and for recursive routines.

```
-----------------------------------------------
                0.00    0.16       4/25          minit_ [4]
                0.00    0.84      21/25          sfun1_ [6]
[5]     97.6    0.00    1.00      25         calcfg_ [5]
                0.00    0.75      10/10          grad_ [9]
                0.00    0.26      13/14          newpt_ [15]
                0.00    0.00      24/24          mindum_ [255]
                0.00    0.00      23/26533       dcopy_ [94]
                0.00    0.00      14/2370        cpulft_ [112]
                0.00    0.00       1/23          mnter_ [264]
-----------------------------------------------
```

Box 10. An example of a section of gprof output

The general parsing procedure knows 2 phases named `parse_parents` and `parse_children`. The procedure starts in the `parse_parents` phase. In the example in Box 10 it builds a list containing the subroutines `minit_` and `sfun1_` . The phase is terminated at the subroutine `calcfg_` . At this point the parent list is stored in the `parents` dictionary. The phase is switched to the `parse_children` phase. Next a list is build up containing `grad_`, `newpt_`, ..., `mnter_` . This phase is terminated at the "`----`" line, at which point the list is stored in the children dictionary and phase is switched to `parse_parents` for the next section. The parse routine terminates at the end-of-file or the string "` This table:`".

Manipulating the data tables

The main two table manipulations are related to the `--stopat` and `--exclude` options. The `--stopat` option is effected simply by replacing the list of children in the `children` dictionary with an empty list for a given subroutine. The same thing is done in the `parents` dictionary.
The `--exclude` option is effected by deleting the whole entry of a given subroutine in the `children` and `parents` dictionaries.
The `--sort` option is effected by iterating through the `children` and `parents` dictionaries and sorting the lists stored with the keys.

Writing the call tree

Writing the call tree out is a matter of simply traversing the stored data recursively. The only critical point is that a subroutine name should not be printed if it does not appear as a key in dictionary. This is required to make the `--exclude` option work correctly.
The `--to` option presents slight challenge in that whether a subroutine name should be printed depends on whether it has a child somewhere down the stack. This challenge is met by building up a list of subroutine names as the tree is being traversed. If a "to" subroutine is found then the list of subroutines is printed. Essentially this is a simple modification of the straightforward tree traversion.

References

1. GAMESS-UK is a package of ab initio programs. See: "http://www.cfs.dl.ac.uk/gamess-uk/index.shtml", M.F. Guest, I. J. Bush, H.J.J. van Dam, P. Sherwood, J.M.H. Thomas, J.H. van Lenthe, R.W.A Havenith, J. Kendrick, "The GAMESS-UK electronic structure package: algorithms, developments and applications",Molecular Physics, Vol. 103, No. 6-8, 719-747.

2. Aprà, E.; Windus, T.L.; Straatsma, T.P.; Bylaska, E.J.; de Jong, W.; Hirata, S.; Valiev, M.; Hackler, M.; Pollack, L.; Kowalski, K.; Harrison, R.; Dupuis, M.; Smith, D.M.A; Nieplocha, J.; Tipparaju V.; Krishnan, M.; Auer, A.A.; Brown, E.; Cisneros, G.; Fann, G.; Fruchtl, H.; Garza, J.; Hirao, K.; Kendall, R.; Nichols, J.; Tsemekhman, K.; Wolinski, K.; Anchell, J.; Bernholdt, D.; Borowski, P.; Clark, T.; Clerc, D.; Dachsel, H.; Deegan, M.; Dyall, K.; Elwood, D.; Glendening, E.; Gutowski, M.; Hess, A.; Jaffe, J.; Johnson, B.; Ju, J.; Kobayashi, R.; Kutteh, R.; Lin, Z.; Littlefield, R.; Long, X.; Meng, B.; Nakajima, T.; Niu, S.; Rosing, M.; Sandrone, G.; Stave, M.; Taylor, H.; Thomas, G.; van Lenthe, J.; Wong, A.; Zhang, Z.; "NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.7" (2005), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA.
3. "High Performance Computational Chemistry: An Overview of NWChem a Distributed Parallel Application," Kendall, R.A.; Apra, E.; Bernholdt, D.E.; Bylaska, E.J.; Dupuis, M.; Fann, G.I.; Harrison, R.J.; Ju, J.; Nichols, J.A.; Nieplocha, J.; Straatsma, T.P.; Windus, T.L.; Wong, A.T.; Computer Phys. Comm. 2000, 128, 260-283.