第1回 gprofゼミ

ゼミ担当者 ： 澤田 淳二，斉藤 宏樹，勝崎 俊樹
指導院生 ： 花田 良子
開催日 ： 2002年10月21日

ゼミ内容: このゼミでは，プロファイリングの必要性を説明し，プロファイリングを行うためのツールの1つである gprof の使い方を紹介する．まず，劣悪なサンプルプログラムを紹介し，そのプログラムでは何処に実行時間が消費されているかを gprof を用いて検証する．そして，その結果をもとに，プログラムの改善を行う．

1 プロファイリングの必要性

Knuth によると,「プログラムの実行時間の 90%はソースコード中の 10%の部分によって消費される」[1] と言われている．この値は言い過ぎだとしても，ソースコード上のごく一部の部分がプログラムの実行時間の大部分を占めていることが多いのは事実である．したがって，プログラムの高速化をはかるためには，そのような実行時間の大部分を占める部分 (ホットスポットと呼ばれる) を特定する作業が重要となる．この作業を**プロファイリング**と呼ぶ．プロファイリングのためのツールを，**プロファイラ**と呼ぶ．

プログラム中で 500 秒かかる処理を 450 秒で実行できるようになったとしても，その処理が 1 回しか実行されないのなら，プログラムの実行時間は 50 秒しか短縮されない．それよりも，1000 回呼ばれている処理を 15 秒から 14 秒に短縮できたとしたら，全体として 1000 秒も実行時間を短縮できることになる．プロファイラでは，どの処理がどれだけの回数呼ばれたかを知ることができるため，その部分の高速化をはかれば，プログラム全体の実行速度を向上させることが可能となる．

2 劣悪なサンプルプログラム

Fibonacci 数列の第 n 項 F_n は，式 (1) で計算できる．

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & n \geq 3 \\ 1 & n = 1, 2 \end{cases} \tag{1}$$

式 (1) を素直にプログラム化すると，Fig. 1 のようになる．

```
int fib(int n)
{
  if(n == 1 || n == 2)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```

Fig. 1 Fibonacci 数列を計算するプログラム

Fig. 1 のプログラムを実行すると，n の値が増加するにつれて実行時間が爆発的にかかるようになってしまうことがわかる．その様子を Fig. 2 に示す．

この問題を解決するためには，プログラムの何処に実行時間がかかっているかを確かめる必要がある．そこで，プロファイラの登場となる．

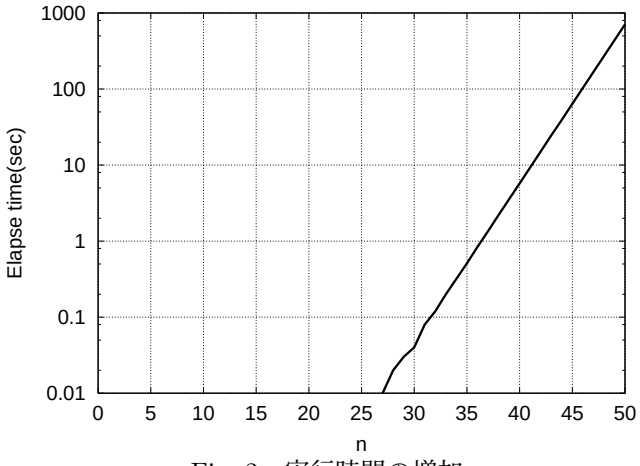

Fig. 2　実行時間の増加

3　プロファイリングの実行

3.1　gprof

gprof [2] とは，プロファイリングを行うためのツールである．プロファイリング用の処理が行われるために，通常のプログラムよりも実行時間は長くなるが，ある関数が実行時間全体で占める割合は，プロファイリングを行っている場合でも行っていない場合でもそう違いはない．

gprof を利用する場合は，コンパイル時に，-pg オプションを付加する．つまり，次のように，コンパイル & リンクを行う．こうすることで，gprof が利用するプロファイリング用のコードが生成される．

```
$ gcc -c foo.c -pg
$ gcc -o foo foo.o -pg
```

このようにして作成したプログラムを実行すると，gmon.out というファイルが作成される．プログラムを実行した後に，

```
$ gprof 実行ファイル名
```

とすると，各関数での処理の合計が全体のどれだけの実行時間を占めているかと，どの関数がどの関数をどれだけ呼び出しているかの統計値が出力される．

3.2　プロファイルの見方

Fig. 3 を例に，プロファイルの見方を説明する．まず，出力結果の Flat profile の意味を以下に示す．

- % time

  全体の総実行時間に対するその関数の総実行時間の占める割合．

- cumulative seconds

  プロファイルリストにおいて，その関数より上位に示されている全ての関数の総実行時間と，その関数の総実行時間の累積時間．

- self seconds

  その関数の総実行時間．プロファイルのリストは，この時間により各関数を降順に並べ，次に calls で降順に並べ替え，最後に name でアルファベット順に並び替えている．

- calls

  その関数が呼び出された総回数．もしその関数が呼び出されなかった場合や，呼び出される回数が決定されなかった (関数がプロファイリング可能なものでコンパイルできなかった) 場合は，空白になる．

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report

(中略)

            Call graph (explanation follows)

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole>[4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
(後略)
```

Fig. 3　プロファイリング結果の例

- self ms/call

  その関数の一回の呼び出しに対する平均実行時間 (ミリ秒)．プロファイル不可能であった場合は空白になる．

- total ms/call

  その関数の一回の呼び出しに対する，その関数とその関数に呼び出されたサブルーチンの平均実行時間 (ミリ秒).

- name

  その関数の名前.

次に， Call graph の意味を以下に示す.

- index

  関数につけられる連続した番号.

- % time

  全実行時間に対するその関数の総実行時間 (その関数から呼び出されたサブルーチンの実行時間も含む) の割合.

- self

  その関数の総実行時間．Call graph のリストも，この時間により各関数を降順に並べている．

- children

  その関数に呼び出されたサブルーチンの総実行時間.

- called

  その関数が呼び出された総回数．もしその関数自身が再帰的に呼び出される場合は，“+” で分けられた二つの数で表される．一つ目の数が再帰でなく呼び出された回数で，二つ目の数が再帰で呼び出された回数である．“/” の前が，注目している関数に呼ばれた数，後ろが呼ばれた総数である．

- name

  その関数の名前．index の番号はこの名前につけられる．

3.3 オプション

gprof では，実行時にオプションをつけることで，結果の出力をコントロールすることができる．以下に，代表的なオプションを紹介する．

- -b

  gprof を用いた場合，通常プロファイルの見方についての説明が加えられる．このオプションを使うことで，その説明部分を省略することができる．

- -p

  通常，出力される結果としては，Fig. 3 のように，Flat profile と Call graph の 2 種類の結果が得られる．Flat profile はそれぞれの関数の使用状況を羅列したものであり，Call graph はそれぞれの関数の階層構造を示す．このオプションを用いることで，Flat profile の部分のみを出力することができる．

- -P

  -p とは反対に，Call graph のみを出力することができる．

- -c

  通常，Call graph に出力される関数は，main 関数によって直接使われる関数のみである．しかし，このオプションを使うことによって，関数内部で使われる関数についての使用状況も出力することができる (Fig. 4 参照).

- -z

  このオプションを使うことで，実行時には使用されていない関数，または時間を計測できない程度しか使われない関数についても，その使用状況を出力することができる．

- -e function_name

  function_name には，実際の関数名が入る (例えば main など)．このオプションを使うことで，指定した関数の出力を結果から除くことができる．

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 92.65      0.63     0.63                             get_fitness(void)
  4.41      0.66     0.03                             ga_crossover_exx(int, int,
 int)
  1.47      0.67     0.01                             ga_newgeneration(void)
  1.47      0.68     0.01                             mcount

                     Call graph


granularity: each sample hit covers 4 byte(s) for 1.49% of 0.67 seconds

index % time    self  children    called     name
                0.00    0.00       0/0           ga_loop(int) [47]
[1]     94.0    0.63    0.00                 get_fitness(void) [1]
                0.00    0.00       0/0           mcount (55)
                0.00    0.00       0/0           sqrt [72]
-----------------------------------------------
(以下略)
```

Fig. 4　-c 使用時の出力結果

```
$ gprof -b -f main ga

(略)

granularity: each sample hit covers 4 byte(s) for 1.49% of 0.67 seconds

index % time    self  children    called     name
                                    2             main [53]
[53]     0.0    0.00    0.00       0+2        main [53]
                                    2             main [53]
-----------------------------------------------

Index by function name

   (2) ga_crossover_exx(int, int, int) (1) get_fitness(void)
   (3) ga_newgeneration(void) (55) mcount
```

Fig. 5　-f 使用時の出力結果

- -f function_name

  このオプションを使うことで，指定した関数の使用状況のみを出力することができる (Fig. 5 参照).

3.4　劣悪なプログラムのプロファイリング

2 節で作成したプログラムにプロファイリング用のコードを付加し，F_{40} の値を求めると，Fig. 6 のプロファイリング結果が得られる．

Fig. 6 より，n の値が増加すると，fib() 関数の再帰呼び出しが爆発的に増えていることがわかる．この部分の改善すれば，実行時間の向上が見られそうである．

4　劣悪なプログラムの改良

3.4 節で 2 節で作成したプログラムのホットスポットが判明したので，その部分を修正する．

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     12.26    12.26        1    12.26    12.26  fib
(中略)
            Call graph (explanation follows)


index % time    self  children    called     name
                            204668308             fib [1]
               12.26    0.00       1/1           main [2]
[1]    100.0   12.26    0.00       1+204668308 fib [1]
                            204668308             fib [1]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00   12.26                 main [2]
               12.26    0.00       1/1           fib [1]
-----------------------------------------------
(後略)
```

Fig. 6　fib 40 のプロファイリング結果

Fibonacci 数列の第 n 項を求めるには，Fig. 7 のような再帰呼び出しを使用しない書き方も可能である．[3)]

```
int fib(int n)
{
  int i, a = 1, b = 0, t;
  for(i = 1; i < n; i++){
    t = a + b; b = a; a = t;
  }
  return a;
}
```

Fig. 7　Fibonacci 数列を計算するプログラム (改良版)

このプログラムで同様に F_{40} の値を求める．そうすると，Fig. 8 のような結果が得られる．

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
  0.00      0.00     0.00        1     0.00     0.00  fib
(中略)
            Call graph (explanation follows)


index % time    self  children    called     name
                0.00    0.00       1/1           main [13]
[1]      0.0    0.00    0.00       1         fib [1]
-----------------------------------------------
(後略)
```

Fig. 8　fib 40 のプロファイリング結果 (改良版)

これで，劣悪なプログラムを改善することができた．

5　まとめ

今回は，サンプルということでわざと実行時間のかかるようなプログラムを用意した．今回の例では，ホットスポットとなる部分が自分の書いたソースコード上に存在したため，その部分を改良することで実行時間の改善をはかれた．しかし，ライブラリ関数など，自分で手出しできない部分が実行時間の大部分を占めているということも実際のプログラムではよくあることである．このような場合は，時間のかかるライブラリ関数の呼び出しをできるだけ抑えることで，実行時間の短縮をはかることが可能である．

参考文献

1) Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, Vol. 1, pp. 105–133, 1971.

2) Jay Fenlason and Richard Stallman. GNU gprof.

3) 奥村晴彦. C 言語による最新アルゴリズム辞典. 技術評論社, 1991.