Scientific Computing II - Exercise 2

Professor: Antti Kuronen
Student: Caike Crepaldi

September 2015

This exercise was done using a ssh remote login through my pangolin (pangolin.it.helsinki.fi) university linux account while using my macbook pro with OS X.

To do such a thing, the following command was run in the OS X Yosemite terminal.

```
Macbook-Pro:~crepaldi $ ssh username@pangolin.it.helsinki.fi
```

The editor I used to edit files and write codes was **emacs**.

1 First Problem

The Fortran source code of the program created for the first (second and third) problem is shown bellow.

array.f90

```
1  program array
2    implicit none
3    integer :: i, it, a(10)
4
5    write(6,*) "Please give the number iteractions:"
6    read(*,*) it
7
8    do i=1,it
9       print *, i, a(i)
10   end do
11
12 end program array
```

The program should ask for the number of desired iteractions and print the value of each vector line corresponding to the iteraction counter. If the given number of iteractions is high enough the program will crash and print the value of each line of the vector until the segmentation fault occurs.

The commands for the compilation and an example of how the program works is shown bellow.

```
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ gfortran -O0 -g -o ex2p1 array.f90
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ ./ex2p1
Please give the number iteractions:
2000
```

(The output of the program's execution is really big so I will only show the last 15 lines)

```
       941  1966042741
       942   796026227
       943   926102065
       944     3225393
       945   791559519
       946  1882355813
       947  1280245809
       948  1146572868
       949  1869098813
       950  1664050541
       951  1885696611
       952     6581345
       953  2019897134
       954     3239986
       955           0
       956           0

Program received signal SIGSEGV: Segmentation fault - invalid memory reference.
Backtrace for this error:

 #0  0x7FA4B9F9B777
 #1  0x7FA4B9F9BD7E
 #2  0x7FA4B9BF3D3F
 #3  0x7FA4BA064280
```

```
#4  0x7FA4BA065820
#5  0x7FA4BA0694AE
#6  0x400A6E in array at array.f90:9

Segmentation fault
```

We can conclude that the program crashes after trying to access the value of the 957th line of the vector **a**. However, every time the program is run in the linux server, the program crashes while trying to access a different line of the vector **a**.

By running the program 2 more times in a row, we verify that the program crashes first in the 1136th line of the vector and then it crashes in the 861th line of the vector.

2 Second Problem

See bellow the commands for such task.

```
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ gdb ex2p1
(gdb) run

ex2p1
 Please give the number iteractions:
2000
           1     6311808
           2           0
           3           8
           4           0
           5           0
           6           0
           7           0
           8           0
           9       -5920
          10       32767
          11     4196384
```

(A lot of values will be skiped here)

```
         466   791770415
         467  1667444579
         468  1634755954
         469  1278174316
         470  2020961897
         471  1869567023
         472  1714254700
         473  1210937967
         474  1697596240
         475  1668441464
         476   761623401
         477  2019897138
         478     3239986
         479           0
         480           0

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7ba2280 in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
(gdb) p i
No symbol "i" in current context.
(gdb) p a
$1 = ( (0, 1045149306), 1.2904777690891933e-08 )
(gdb) up
#1  0x00007ffff7ba3821 in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
(gdb) p i
No symbol "i" in current context.
(gdb) up
#2  0x00007ffff7ba74af in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
(gdb) p i
No symbol "i" in current context.
(gdb) up
#3  0x0000000000400a6f in array () at array.f90:9
9         print *, i, a(i)
(gdb) p i
$2 = 481
(gdb) p a
$3 = (6311808, 0, 8, 0, 0, 0, 0, 0, -5920, 32767)
```

We can se above (in the last four lines) the value of the vector **a** and the last index **i** just before the program crashes.

3 Third Problem

In order to create a core dump to be investigated by the **gdb** debugger we must use the following commands.

```
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ ulimit -c 10000
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ ./ex2p1
```

```
 ex2p1
  Please give the number iteractions:
 2000
            1     6311808
            2           0
            3           8
            4           0
            5           0
            6           0
            7           0
            8           0
            9       -5920
           10       32767
           11     4196384
```

(a lot of values will be skiped here)

```
          540  1146572868
          541  1869098813
          542  1664050541
          543  1885696611
          544     6581345
          545  2019897134
          546     3239986
          547           0
          548           0

 Program received signal SIGSEGV: Segmentation fault - invalid memory reference.
```

Now we just need to use the **gdb** debugger to analyse the core dump.

```
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ ls
array.f90 core ex2p1
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ gdb ex2p1 -c core
 Core was generated by `./ex2p1'.
 Program terminated with signal SIGSEGV, Segmentation fault.
 #0  0x00007f76ec5bc280 in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
 (gdb) where
 #0  0x00007f76ec5bc280 in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
 #1  0x00007f76ec5bd821 in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
 #2  0x00007f76ec5c14af in ?? () from /usr/lib/x86_64-linux-gnu/libgfortran.so.3
 #3  0x0000000000400a6f in array () at array.f90:9
 #4  0x0000000000400ad9 in main (argc=1, argv=0x7ffcff34fca0) at array.f90:12
 #5  0x00007f76ec136ec5 in __libc_start_main (main=0x400aa5 <main>, argc=1,
     argv=0x7ffcff34f8b8, init=<optimised out>, fini=<optimised out>,
     rtld_fini=<optimised out>, stack_end=0x7ffcff34f8a8) at libc-start.c:287
 #6  0x0000000000400849 in _start ()
```

4 Fourth Problem

The program code can be seen bellow.

simpleprofiling.f90

```
1  program simpleprofiling
2    integer,parameter :: rk=selected_real_kind(5,100)
3    integer,parameter :: NMAX=100000,MMAX=1000
4    real(rk),parameter :: dx=0.00001
5    real(rk) :: sum1,sum2
6    integer :: i,j
7
8    sum1=0.0
9    sum2=0.0
10
11   do j=0,MMAX
12      do i=0,NMAX
13         sum1=sum1+dx*i*j
14      end do
15   end do
16
17   do j=0,MMAX
18      do i=0,NMAX
19         sum2=sum2+cos(exp(sin(dx*i*j)))
20      end do
21   end do
```

```
22
23    print *,sum1,sum2
24
25 end program simpleprofiling
```

According to the assignment, we must look for the time spent in the lines 13 and 19 of the source code.
In order to perform such task we must use the commands bellow.

```
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ gfortran -O0 -p -g -o simpprof simpleprofiling.f90
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ ./simpprof
 25025249617.809803 24135013.748771366
username@tktl-pangolin:~/Tools-for-HPC/exercise-2$ gprof -l simpprof
```

```
 Flat profile:

 Each sample counts as 0.01 seconds.
   %   cumulative   self              self     total
  time   seconds   seconds    calls  Ts/call  Ts/call  name
  44.30      0.92     0.92                             simpleprofiling (simpleprofiling.f90:13 @ 400a0f)
  41.87      1.78     0.87                             simpleprofiling (simpleprofiling.f90:19 @ 400a8d)
   6.09      1.91     0.13                             simpleprofiling (simpleprofiling.f90:18 @ 400b00)
   5.36      2.02     0.11                             simpleprofiling (simpleprofiling.f90:12 @ 400a37)
   0.97      2.04     0.02                             simpleprofiling (simpleprofiling.f90:18 @ 400a79)
   0.00      2.04     0.00        1     0.00     0.00  MAIN__ (simpleprofiling.f90:1 @ 4009cd)

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
       else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
       function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
       the function in the gprof listing. If the index is
       in parenthesis it shows where it would appear in
       the gprof listing if it were to be printed.


Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.


                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.49% of 2.04 seconds

index % time    self  children    called     name
                0.00    0.00       1/1           main (simpleprofiling.f90:25 @ 400bb3) [11]
[6]     0.0     0.00    0.00       1         MAIN__ (simpleprofiling.f90:1 @ 4009cd) [6]
-----------------------------------------------


 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index  A unique number given to each element of the table.
            Index numbers are sorted numerically.
            The index number is printed next to every function name so
```

it is easier to look up where the function is in the table.

% time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved.

Index by function name

```
[6] MAIN__ (simpleprofiling.f90:1 @ 4009cd) [4] simpleprofiling (simpleprofiling.f90:12 @ 400a37) [2]
    simpleprofiling (simpleprofiling.f90:19 @ 400a8d)
[1] simpleprofiling (simpleprofiling.f90:13 @ 400a0f) [5] simpleprofiling (simpleprofiling.f90:18 @ 400
    a79) [3] simpleprofiling (simpleprofiling.f90:18 @ 400b00)
```

We can see in the *"Flat profile"* the time consumption in both lines of code (13 and 19) and it is clear that the time spent in both lines is far greater than the time spent in other lines of the source code.

In the 13th line, the program spent ≈44% of the execution time (0.92 seconds). In the 19th line, the program spent ≈42% of the execution time (0.87 seconds).

If our intention was to optimize the code, we should concentrate in those two lines where most of the execution time is spent.

The reason why the time spent in those two lines is much bigger should be the two **do** loops used in each sum calculation.

Note: About this exercise

All files (and Makefiles, when applicable) created by the student and used in this exercise were packed together with this PDF documentation and shall be available to the assistant for further analysis in its own problem folder.