Software Development

Debugging and Profiling

Alexander B. Pacheco
Research Computing
June 30, 2021

Outline

Introduction

Debugging vs Profiling

Debugging

- a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected
- a developer activity and effective debugging is very important before testing begins to increase the quality of the system.

Profiling

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing.
- This information can show you which pieces of your program are slower than you expected,

  and might be candidates for rewriting to make your program execute faster.
- It can also tell you which functions are being called more or less often than you expected.

Available Tools

Open Source

1. GNU Debugger (gdb)
2. Data Display Debugger (ddd), visual frontend to gdb
3. GNU Profiler (gprof)
4. Valgrind

Commercial (some are free)

1. Intel VTune Profiler
2. NVIDIA Nsight Graphics
3. TotalView
4. Arm Forge toolsuite - ARM DDT and ARM MAP (formerly Allinea DDT and Allinea MAP)

gdb and ddd

What is GNU Debugger or gdb?

- most popular debugger for UNIX systems for several languages
- GNU Debugger helps you in getting information about the following:
  - If a core dump happened, then what statement or expression did the program crash on?
  - If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
  - What are the values of program variables at a particular point during execution of the program?
  - What is the result of a particular expression in a program?

Starting gdb I

- Run the command `gdb <program name>` to start `gdb` and debug program `program name`

```
[alp514.sol](1026): gdb ./md
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-12.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type ``show copying'' and ``show warranty'' for details.
This GDB was configured as ``x86_64-redhat-linux-gnu''.
Type ``show configuration'' for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type ``help''.
Type ``apropos word'' to search for commands related to ``word''...
Reading symbols from ./md...(no debugging symbols found)...done.
(gdb)
```

Starting gdb II

- If you do not enter the program to debug, then enter `file <program name>` on the `gdb` prompt

```
[alp514.sol](1027): gdb
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-12.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type ‘‘show copying’’ and ‘‘show warranty’’ for details.
This GDB was configured as ‘‘x86_64-redhat-linux-gnu’’.
Type ‘‘show configuration’’ for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type ‘‘help’’.
Type ‘‘apropos word’’ to search for commands related to ‘‘word’’.
(gdb) file ./md
Reading symbols from ./md...(no debugging symbols found)...done.
(gdb)
```

Getting Help

- Type help at the prompt to see list of available commands and their documentation

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type ‘‘help'' followed by a class name for a list of commands in that class.
Type ‘‘help all'' for the list of all commands.
Type ‘‘help'' followed by command name for full documentation.
Type ‘‘apropos word'' to search for commands related to ‘‘word''.
Command name abbreviations are allowed if unambiguous.
```

Running the program

- To run the program, type `run`, followed by command line arguments

```
(gdb) run < md.inp
Starting program: /home/alp514/Workshop/sum2015/fortran/MolDyn/srcv3/md < md.inp
[Thread debugging using libthread_db enabled]
Using host libthread_db library ‘‘/lib64/libthread_db.so.1’’.
Input Parameters:
&MOLDYN
 NATOM=4000                ,
 NPARTDIM=10                  ,
 NSTEP=10                  ,
 TEMPK=  10.000000000000000     ,
 DT=  1.0000000000000000E-003,
 POT=’’lj’’,
 /
Initial Average Temperature:    0.10033457E+01
Initial Scaled Average Temperature:    0.10000000E+02
Average Temperature:          1   0.99978284E+01 -0.33498311E+05
... skip ...
Average Temperature:         10   0.99497344E+01 -0.32058068E+05
Init Time:        0.012
Sim  Time:        5.314
[Inferior 1 (process 488073) exited normally]
```

How do I detect bugs?

- If your program has bugs, you do not want to run the code but stop at various times evaluating the functions, subroutine and various.
- `gdb` provides various commands to debug your code
    1. `list`: list the next 10 lines of the code
    2. `break n`: insert breakpoint at line n
    3. `step`: execute the next line of code
    4. `next`: same as `step` but will not step into a function or subroutine
    5. `continue`: run until until breakpoint or end of program
    6. `print var`: print the value of a variable, `var`
    7. `watch var`: watch the variable, `var` and pause the program if the value changes
    8. `backtrace`: produces a stack trace of the function calls that lead to a seg fault
    9. `where`: same as `backtrace`; you can think of this version as working even when you're still in the middle of the program
    10. `finish`: runs until the current function is finished
    11. `delete`: deletes a specified breakpoint
    12. `info breakpoints`: shows information about all declared breakpoints

Demo

- Request an interactive session on Sol or start the shell terminal app on Open OnDemand
- Navigate to the directory where you have a code that you want to debug OR
- Copy the codes from my directory
- Follow along - Please feel free to unmute of you have a question

ddd

- `ddd`: GNU DDD is a graphical front-end for command-line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake, or the Python debugger pydb.
- load the `ddd`: `module load ddd`
- run the command `ddd` or `ddd <program name>`

gprof

GNU Profiler: gprof

- Gprof is a free profiler from GNU
  - simple way to analyze runtime behaviour of an application
    (low overhead, collect various meaningful insights)
  - determine where most of the execution time is spent
    locate code regions suited for optimization
  - analyzes connections between individual functions
    helps in understanding code and suggests elimination of expensive function calls
  - part of GNU Binutils and supported by various compilers
    available as open-source, almost everywhere
  - works for C/C++, and Fortran

How it works I

- Compile and link source code with the option `-pg`

```
[alp514.sol](1069): make all
gfortran -g -pg  -c precision.f90
gfortran -g -pg  -c param.f90
gfortran -g -pg  -c potential.f90
gfortran -g -pg  -c md.f90
gfortran -g -pg  -c initialize.f90
gfortran -g -pg  -c linearmom.f90
gfortran -g -pg  -c verlet.f90
gfortran -g -pg  -c get_temp.f90
gfortran -g -pg  -o md precision.o param.o potential.o md.o initialize.o linearmom.o verlet.o
      get_temp.o
```

How it works II

- Run the code (use shorter representative input)

```
[alp514.sol](1071): ./md < md.inp
Input Parameters:
&MOLDYN
 NATOM=4000                       ,
 NPARTDIM=10                         ,
 NSTEP=5                          ,
 TEMPK=  10.000000000000000         ,
 DT=  1.0000000000000000E-003,
 POT=''lj'',
 /
Initial Average Temperature:    0.10090031E+01
Initial Scaled Average Temperature:    0.10000000E+02
Average Temperature:          1   0.99978149E+01 -0.33498234E+05
Average Temperature:          2   0.99934140E+01 -0.33458772E+05
Average Temperature:          3   0.99889220E+01 -0.33392393E+05
Average Temperature:          4   0.99842784E+01 -0.33298190E+05
Average Temperature:          5   0.99794196E+01 -0.33174874E+05
Init Time:        0.012
Sim  Time:       15.380
```

How it works III

- The Flat Profile shows how much time is spent in each function and how often each function was called

```
[alp514.sol](1072): gprof --flat-profile ./md
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 62.64      1.54     1.54        5     0.31     0.48  verlet_
 28.47      2.24     0.70 39990000     0.00     0.00  __potential_MOD_lennard_jones
  3.66      2.33     0.09 39990000     0.00     0.00  __potential_MOD_dvdr_lj
  3.46      2.42     0.09 39990000     0.00     0.00  __potential_MOD_pot_lj
  0.81      2.44     0.02                             __potential_MOD_dvdr_mp
  0.81      2.46     0.02                             __potential_MOD_morse
  0.20      2.46     0.01                             __potential_MOD_pot_mp
  0.00      2.46     0.00    12000     0.00     0.00  main
  0.00      2.46     0.00        7     0.00     0.00  get_temp_
  0.00      2.46     0.00        6     0.00     0.00  linearmom_
  0.00      2.46     0.00        1     0.00     2.42  MAIN__
  0.00      2.46     0.00        1     0.00     0.00  initialize_
```

How it works IV

- The Call Graph shows which functions called each other and how many times.

```
[alp514.sol](1073): gprof --graph ./md
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.41% of 2.46 seconds

index % time    self  children    called     name
                1.54    0.88       5/5           MAIN__ <cycle 1> [3]
[2]     98.2    1.54    0.88       5         verlet_ [2]
                0.70    0.18 39990000/39990000     __potential_MOD_lennard_jones [4]
-----------------------------------------------
                                   1             main <cycle 1> [10]
[3]     98.2    0.00    2.42       1         MAIN__ <cycle 1> [3]
                1.54    0.88       5/5           verlet_ [2]
                0.00    0.00       7/7           get_temp_ [11]
                0.00    0.00       6/6           linearmom_ [12]
                                   1             initialize_ <cycle 1> [13]
-----------------------------------------------
                0.70    0.18 39990000/39990000     verlet_ [2]
[4]     35.6    0.70    0.18 39990000         __potential_MOD_lennard_jones [4]
                0.09    0.00 39990000/39990000     __potential_MOD_dvdr_lj [5]
                0.09    0.00 39990000/39990000     __potential_MOD_pot_lj [6]
```

How it works V

```
-----------------------------------------------
                0.09    0.00 39990000/39990000     __potential_MOD_lennard_jones [4]
[5]      3.7    0.09    0.00 39990000         __potential_MOD_dvdr_lj [5]
-----------------------------------------------
                0.09    0.00 39990000/39990000     __potential_MOD_lennard_jones [4]
[6]      3.5    0.09    0.00 39990000         __potential_MOD_pot_lj [6]
-----------------------------------------------
                                                   <spontaneous>
[7]      0.8    0.02    0.00                  __potential_MOD_dvdr_mp [7]
-----------------------------------------------
                                                   <spontaneous>
[8]      0.8    0.02    0.00                  __potential_MOD_morse [8]
-----------------------------------------------
                                                   <spontaneous>
[9]      0.2    0.01    0.00                  __potential_MOD_pot_mp [9]
-----------------------------------------------
                              12000                initialize_ <cycle 1> [13]
[10]     0.0    0.00    0.00  12000           main <cycle 1> [10]
                                  1                MAIN__ <cycle 1> [3]
-----------------------------------------------
                0.00    0.00      7/7              MAIN__ <cycle 1> [3]
[11]     0.0    0.00    0.00      7           get_temp_ [11]
-----------------------------------------------
                0.00    0.00      6/6              MAIN__ <cycle 1> [3]
[12]     0.0    0.00    0.00      6           linearmom_ [12]
```

How it works VI

```
-----------------------------------------------
                                  1             MAIN__ <cycle 1> [3]
[13]     0.0    0.00    0.00      1         initialize_ <cycle 1> [13]
                              12000             main <cycle 1> [10]
-----------------------------------------------
```

- Gprof can even annotate your source code. (Add option `-g` at compile time.)

```
[alp514.sol](1082): gprof --annotated-source ./fib
*** File /home/alp514/Workshop/sum2015/fortran/Solution/fibonacci.f90:
           1 -> program fibonacci

                  implicit none
                  integer, parameter :: dp = selected_real_kind(15)
                  integer :: i, n, fib0, fib1, fib

                  print *, ``Enter the Fibonacci number''
                  read *, n

                  fib0 = 0
                  fib1 = 1

                  print *, ``n,  f(n)''
```

How it works VII

```
                ! 0 + 1 + 2 + ... + n

                open(10, file=''fib.dat'')
                do i = 2, n
                   fib = fib1 + fib0
                   write(10, *) i, fib
                   fib0 = fib1
                   fib1 = fib
                end do

     ##### -> end program fibonacci




Top 10 Lines:

     Line      Count

        1          1

Execution Summary:

        2   Executable lines in this file
```

How it works VIII

```
     2   Lines executed
100.00   Percent of the file executed

     1   Total number of line executions
  0.50   Average executions per line
```

Valgrind

Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail.
- The Valgrind distribution currently includes seven production-quality tools:
    1. a memory error detector,
    2. two thread error detectors,
    3. a cache and branch-prediction profiler,
    4. a call-graph generating cache and branch-prediction profiler, and
    5. two different heap profilers.

Why use Valgrind?

- can automatically detect many memory management and threading bugs
- can perform very detailed profiling to help find bottlenecks in your programs
- uses dynamic binary instrumentation, so you don't need to modify, recompile or relink your applications
- a debugging and profiling system for large, complex programs
- suitable for any type of software
- works with programs written in any language
  - used on programs written partly or entirely in C, C++, Java, Perl, Python, assembly code, Fortran, Ada, and many others
- can even be used on programs for which you don't have the source code

Memory Leak I

- a type of resource leak that occurs when a computer program incorrectly manages memory allocations
- can also occur when an object is stored in memory but cannot be accessed by the running code
- has symptoms similar to a number of other problems and generally can only be diagnosed by a programmer with access to the programs' source code
- are often the cause of or a contributing factor to software aging
    - software aging refers to all software's tendency to fail, or cause a system failure after running continuously for a certain time, or because of ongoing changes in systems surrounding the software
- reduces the performance of the computer by reducing the amount of available memory
- may not be serious or even detectable by normal means
- memory leak in a program that only runs for a short time may not be noticed and is rarely serious

Memory Leak II

- Much more serious leaks include those:
  - where the program runs for an extended time and consumes additional memory over time, such as background tasks on servers, but especially in embedded devices which may be left running for many years
  - where new memory is allocated frequently for one-time tasks, such as when rendering the frames of a computer game or animated video
  - where the program can request memory — such as shared memory — that is not released, even when the program terminates
  - where memory is very limited, such as in an embedded system or portable device, or where the program requires a very large amount of memory to begin with, leaving little margin for leakage
  - where the leak occurs within the operating system or memory manager
  - when a system device driver causes the leak
  - running on an operating system that does not automatically release memory on program termination.

Using Valgrind

- Usage: `valgrind [valgrind-options] <program-name> [program-options]`

```
[2021-06-25 10:35.44] ~/Workshop/2021HPC/debugging
[alp514.pavo5](1118): cat memleak.c
#include <stdlib.h>

void foo(void) {
  int* x;
  x = malloc(10 * sizeof(int));
  x[10] = 0;        // heap block overrun
  return;           // x not freed
}

int main(void) {
  foo();
  return 0;
}
```

- Compile with debug symbols enabled i.e. add `-g` flag and run using default option.

```
[2021-06-25 10:35.50] ~/Workshop/2021HPC/debugging
[alp514.pavo5](1119): gcc -g -o memleak memleak.c
[2021-06-25 10:36.02] ~/Workshop/2021HPC/debugging
[alp514.pavo5](1120): valgrind ./memleak
```

Interpreting Output I

- All lines are prepended with `==ProcessID==`
- Starts with a banner that displays version and command run

```
==63157== Memcheck, a memory error detector
==63157== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==63157== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==63157== Command: ./memleak
```

- If you code generates regular output you should see that here (not in this example)
- `valgrind` next reports a *Invalid write* i.e. writing to memory location that is not owned by the code

```
==63157== Invalid write of size 4
==63157==    at 0x401144: foo (memleak.c:6)
==63157==    by 0x401155: main (memleak.c:11)
==63157==  Address 0x5206068 is 0 bytes after a block of size 40 alloc'd
==63157==    at 0x4C34F47: malloc (vg_replace_malloc.c:309)
==63157==    by 0x401137: foo (memleak.c:5)
==63157==    by 0x401155: main (memleak.c:11)
```

Interpreting Output II

- Lastly, `valgrind` checks for any memory that was allocated and never deleted, and prints a report on this memory *in use at exit*
- If a block of memory is both in use at exit and there is no pointer to it, we have a memory leak: memory that the programcould not possibly delete

```
==63157== HEAP SUMMARY:
==63157==     in use at exit: 40 bytes in 1 blocks
==63157==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==63157==
==63157== LEAK SUMMARY:
==63157==    definitely lost: 40 bytes in 1 blocks
==63157==    indirectly lost: 0 bytes in 0 blocks
==63157==      possibly lost: 0 bytes in 0 blocks
==63157==    still reachable: 0 bytes in 0 blocks
==63157==         suppressed: 0 bytes in 0 blocks
==63157== Rerun with --leak-check=full to see details of leaked memory
==63157==
==63157== For lists of detected and suppressed errors, rerun with: -s
==63157== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

- Output if there is no memory leak

Interpreting Output III

```
[2021-06-25 11:00.13] ~/Workshop/2021HPC/debugging
[alp514.pavo5](1148): cat memleak3.f90
program memleak

  implicit none

  call foo()

contains

  subroutine foo
    integer, dimension(:), pointer :: x

    allocate(x(10))

    x(10) = 0         ! fixed heap block overrun
    deallocate(x)     ! x is deallocated freeing memory
    return
  end subroutine foo

end program memleak
[2021-06-25 11:00.51] ~/Workshop/2021HPC/debugging
[alp514.pavo5](1149): valgrind ./memleakf3
==63495== Memcheck, a memory error detector
==63495== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
```

Interpreting Output IV

```
==63495== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==63495== Command: ./memleakf3
==63495==
==63495==
==63495== HEAP SUMMARY:
==63495==     in use at exit: 0 bytes in 0 blocks
==63495==   total heap usage: 22 allocs, 22 frees, 13,560 bytes allocated
==63495==
==63495== All heap blocks were freed -- no leaks are possible
==63495==
==63495== For lists of detected and suppressed errors, rerun with: -s
==63495== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cachegrind I

- Cachegrind simulates how your program interacts with a machine's cache hierarchy and (optionally) branch predictor.
- It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2).
- Cachegrind gathers the following statistics
  - I cache reads (Ir, which equals the number of instructions executed), I1 cache read misses (I1mr) and LL cache instruction read misses (ILmr).
  - D cache reads (Dr, which equals the number of memory reads), D1 cache read misses (D1mr), and LL cache data read misses (DLmr).
  - D cache writes (Dw, which equals the number of memory writes), D1 cache write misses (D1mw), and LL cache data write misses (DLmw).
  - Conditional branches executed (Bc) and conditional branches mispredicted (Bcm).
  - Indirect branches executed (Bi) and indirect branches mispredicted (Bim).
- run Cachegrind to gather the profiling information, and
- Usage: `valgrind --tool=cachegrind <program name> <program options>`

Cachegrind II

```
[alp514.sol](1084): valgrind --tool=cachegrind ./md
==2684769== Cachegrind, a cache and branch-prediction profiler
==2684769== Copyright (C) 2002-2017, and GNU GPL’d, by Nicholas Nethercote et al.
==2684769== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==2684769== Command: ./md
==2684769==
--2684769-- warning: L3 cache found, using its data for the LL simulation.
--2684769-- warning: specified LL cache: line_size 64  assoc 20  total_size 26,214,400
--2684769-- warning: simulated LL cache: line_size 64  assoc 25  total_size 26,214,400
Initial Average Temperature:    9.97340091E-01
Initial Scaled Average Temperature:    1.00000000E+01
Average Temperature:         1   9.99780302E+00 -3.34981556E+04
Average Temperature:         2   9.99338410E+00 -3.34585091E+04
Average Temperature:         3   9.98888351E+00 -3.33918970E+04
Average Temperature:         4   9.98423690E+00 -3.32974444E+04
Average Temperature:         5   9.97937661E+00 -3.31738710E+04
Average Temperature:         6   9.97423159E+00 -3.30194709E+04
Average Temperature:         7   9.96872615E+00 -3.28320847E+04
Average Temperature:         8   9.96277887E+00 -3.26090649E+04
Average Temperature:         9   9.95630149E+00 -3.23472350E+04
Average Temperature:        10   9.94919802E+00 -3.20428429E+04
==2684769==
==2684769== Process terminating with default action of signal 27 (SIGPROF)
==2684769==    at 0x5977A63: __open_nocancel (in /usr/lib64/libc-2.28.so)
==2684769==    by 0x5983FCF: write_gmon (in /usr/lib64/libc-2.28.so)
==2684769==    by 0x59847CD: _mcleanup (in /usr/lib64/libc-2.28.so)
==2684769==    by 0x58BEF8B: __run_exit_handlers (in /usr/lib64/libc-2.28.so)
==2684769==    by 0x58BF0BF: exit (in /usr/lib64/libc-2.28.so)
==2684769==    by 0x58A87B9: (below main) (in /usr/lib64/libc-2.28.so)
==2684769==
==2684769== I   refs:      37,383,926,147
==2684769== I1  misses:         3,704,219
==2684769== LLi misses:             2,214
==2684769== I1  miss rate:           0.01%
==2684769== LLi miss rate:           0.00%
==2684769==
==2684769== D   refs:      13,437,341,766  (11,266,871,670 rd   + 2,170,470,096 wr)
```

Cachegrind III

```
==2684769== D1  misses:        58,568,835  (    58,405,295 rd   +       163,540 wr)
==2684769== LLd misses:            16,927  (         4,603 rd   +        12,324 wr)
==2684769== D1  miss rate:            0.4% (           0.5%     +           0.0%  )
==2684769== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==2684769==
==2684769== LL refs:           62,273,054  (    62,109,514 rd   +       163,540 wr)
==2684769== LL misses:             19,141  (         6,817 rd   +        12,324 wr)
==2684769== LL miss rate:             0.0% (           0.0%     +           0.0%  )
Profiling timer expired
```

- run `cg_annotate` to get a detailed presentation of that information

```
[alp514.sol](1087): cg_annotate cachegrind.out.2684769
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         26214400 B, 64 B, 25-way associative
Command:          ./md
Data file:        cachegrind.out.2684769
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir                      I1mr               ILmr           Dr                      D1mr               DLmr           Dw                      D1mw
                     DLmw
--------------------------------------------------------------------------------
37,383,926,147 (100.0%) 3,704,219 (100.0%) 2,214 (100.0%) 11,266,871,670 (100.0%) 58,405,295 (100.0%) 4,603 (100.0%) 2,170,470,096 (100.0%) 163,540 (1
         00.0%) 12,324 (100.0%)  PROGRAM TOTALS

--------------------------------------------------------------------------------
```

Cachegrind IV

```
Ir                     I1mr             ILmr        Dr                     D1mr              DLmr          Dw                     D1mw
                     DLmw             file:function
--------------------------------------------------------------------------------
30,598,675,512 (81.85%)   136,605 ( 3.69%)  99 ( 4.47%) 10,490,989,557 (93.11%) 58,339,260 (99.89%)     2 ( 0.04%) 2,082,046,072 (95.93%) 157,945 (96.
        58%) 10,501 (85.21%)  /home/alp514/Workshop/2021HPC/fortran/MolDyn/orig/md-orig.f90:MAIN__
 4,318,920,000 (11.55%)        20 ( 0.00%)   2 ( 0.09%)    239,940,000 ( 2.13%)          0              0                      0                0
                       0           ???:lround
 1,599,600,000 ( 4.28%)        10 ( 0.00%)   1 ( 0.05%)     79,980,000 ( 0.71%)          0              0                      0                0
                       0           ???:__powidf2
   506,857,492 ( 1.36%) 1,789,425 (48.31%) 546 (24.66%)    374,811,801 ( 3.33%)     18,345 ( 0.03%)    32 ( 0.70%)    30,294,235 ( 1.40%)   2,174 ( 1.
             33%)    294 ( 2.39%)  ???:???
   109,423,825 ( 0.29%)   727,410 (19.64%) 106 ( 4.79%)     21,821,524 ( 0.19%)        222 ( 0.00%)    10 ( 0.22%)    13,433,163 ( 0.62%)     150 ( 0.
             09%)      6 ( 0.05%)  ???:__printf_fp_l
    54,403,658 ( 0.15%)   215,709 ( 5.82%)  68 ( 3.07%)     14,393,152 ( 0.13%)        150 ( 0.00%)     4 ( 0.09%)     8,715,024 ( 0.40%)     150 ( 0.
             09%)      1 ( 0.01%)  ???:printf_positional
    37,589,527 ( 0.10%)     9,115 ( 0.25%)   6 ( 0.27%)     13,124,792 ( 0.12%)          0              0              6,032,456 ( 0.28%)      11 ( 0.
             01%)      1 ( 0.01%)  ???:hack_digit

--------------------------------------------------------------------------------
-- Auto-annotated source: /home/alp514/Workshop/2021HPC/fortran/MolDyn/orig/md-orig.f90
--------------------------------------------------------------------------------
Ir                    I1mr            ILmr       Dr                    D1mr              DLmr       Dw                    D1mw            DLmw

            5 ( 0.00%)      1 ( 0.00%) 1 ( 0.05%)            0                   0          0                   3 ( 0.00%)      0              0
                                  program md
... skipping the rest ..

--------------------------------------------------------------------------------
Ir                     I1mr             ILmr        Dr                     D1mr              DLmr       Dw                     D1mw
         DLmw
--------------------------------------------------------------------------------
30,599,156,508 (81.85%) 136,613 ( 3.69%) 104 ( 4.70%) 10,491,154,555 (93.12%) 58,339,263 (99.89%) 2 ( 0.04%) 2,082,150,409 (95.93%) 157,945 (96.58%) 1
         0,501 (85.21%)  events annotated
```

Callgrind I

- Callgrind is a profiling tool that records the call history among functions in a program's run as a call-graph.
- By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls.
- Optionally, cache simulation and/or branch prediction (similar to Cachegrind) can produce further information about the runtime behavior of an application.
- The profile data is written out to a file at program termination. For presentation of the data, and interactive control of the profiling, two command line tools are provided:
- `callgrind_annotate`: reads in the profile data, and prints a sorted lists of functions, optionally with source annotation.
- Use `qcachegrind` for graphical visualization of the data to navigate the large amount of data that Callgrind produces.

Callgrind II

- `callgrind_control`:enables you to interactively observe and control the status of a program currently running under Callgrind's control, without stopping the program. You can get statistics information as well as the current stack trace, and you can request zeroing of counters or dumping of profile data.
- To start a profile run for a program, execute:

```
valgrind --tool=callgrind [callgrind options] your-program [program options]
```

```
[alp514.sol](1003): valgrind --tool=callgrind ./md
==3076654== Callgrind, a call-graph generating cache profiler
==3076654== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3076654== Using Valgrind-3.16.0 and LibVEX; rerun with -h for copyright info
==3076654== Command: ./md
==3076654==
==3076654== For interactive control, run 'callgrind_control -h'.
Initial Average Temperature:    1.00606866E+00
Initial Scaled Average Temperature:    1.00000000E+01
Average Temperature:          1   9.99780325E+00 -3.34981611E+04
Average Temperature:          2   9.99338074E+00 -3.34584985E+04
Average Temperature:          3   9.98886821E+00 -3.33917992E+04
Average Temperature:          4   9.98420035E+00 -3.32971328E+04
Average Temperature:          5   9.97930766E+00 -3.31731528E+04
Average Temperature:          6   9.97411590E+00 -3.30180681E+04
Average Temperature:          7   9.96854463E+00 -3.28296057E+04
Average Temperature:          8   9.96250549E+00 -3.26049633E+04
Average Temperature:          9   9.95590058E+00 -3.23407523E+04
Average Temperature:         10   9.94862103E+00 -3.20329313E+04
==3076654==
==3076654== Process terminating with default action of signal 27 (SIGPROF)
```

Callgrind III

```
==3076654==    at 0x5977A63: __open_nocancel (in /usr/lib64/libc-2.28.so)
==3076654==    by 0x5983FCF: write_gmon (in /usr/lib64/libc-2.28.so)
==3076654==    by 0x59847CD: _mcleanup (in /usr/lib64/libc-2.28.so)
==3076654==    by 0x58BEF8B: __run_exit_handlers (in /usr/lib64/libc-2.28.so)
==3076654==    by 0x58BF0BF: exit (in /usr/lib64/libc-2.28.so)
==3076654==    by 0x58A87B9: (below main) (in /usr/lib64/libc-2.28.so)
==3076654==
==3076654== Events    : Ir
==3076654== Collected : 37380887421
==3076654==
==3076654== I   refs:      37,380,887,421
Profiling timer expired
```

- To generate a function-by-function summary from the profile data file, use

```
callgrind_annotate [options] callgrind.out.<pid>
```

Callgrind IV

```
[alp514.sol](1004): callgrind_annotate callgrind.out.3076654
--------------------------------------------------------------------------------
Profile data file 'callgrind.out.3076654' (creator: callgrind-3.16.0)
--------------------------------------------------------------------------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 3237167226
Trigger: Program termination
Profiled target:  ./md (PID 3076654, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
User annotated:
Auto-annotation:  on

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
37,380,595,563 (100.0%)  PROGRAM TOTALS

--------------------------------------------------------------------------------
Ir                       file:function
--------------------------------------------------------------------------------
30,918,860,230 (82.71%)  md-orig.f90:MAIN__ [/home/alp514/Workshop/2021HPC/fortran/MolDyn/orig/md]
 4,318,920,000 (11.55%)  ???:lround [/usr/lib64/libm-2.28.so]
 1,599,600,000 ( 4.28%)  ???:__powidf2 [/usr/lib64/libgcc_s-8-20191121.so.1]
   107,526,539 ( 0.29%)  ???:__printf_fp_l [/usr/lib64/libc-2.28.so]
    54,139,536 ( 0.14%)  ???:printf_positional [/usr/lib64/libc-2.28.so]
    37,644,281 ( 0.10%)  ???:hack_digit [/usr/lib64/libc-2.28.so]

--------------------------------------------------------------------------------
-- Auto-annotated source: md-orig.f90
--------------------------------------------------------------------------------
Ir
```

Callgrind V

```
           6 ( 0.00%)  program md
          51 ( 0.00%)  => ???:mcount (1x)
           .
           .              ! Molecular Dynamics code for equilibration of Liquid Argon
... skipping the rest ...

--------------------------------------------------------------------------------
Ir
--------------------------------------------------------------------------------
30,919,372,586 (82.72%)  events annotated
```

Other Tools available on Sol & Hawk

Intel OneAPI I

- oneAPI is an open standard of a unified application programming interface intended to be used across different compute accelerator (coprocessor) architectures.
- It is intended to eliminate the need for developers to maintain separate code bases, multiple programming languages, and different tools and workflows for each architecture.
- Intel has released production quality oneAPI toolkits that implement the specification and add migration, analysis, and debug tools.
- These include the Intel C++ compiler, Intel Fortran compiler, VTune and multiple performance libraries.
- The Intel OneAPI toolkits are available at no charge at `https://software.intel.com/`.
- Download the Intel OneAPI Base Toolkit.
- If you need access to Fortran Compiler or MPI libraries, download the HPC Toolkit also.

Intel OneAPI II

- On Sol, Intel OneAPI 2021.02 is available at `/share/Apps/intel-oneapi`.
- To put Intel OneAPI in your path, run the command

```
source /share/Apps/intel-oneapi/setvars.sh
```

- Tools available for Debugging and Profiling
  1. Intel VTune Profiling (`vtune`, `vtune-gui`): performance analysis tool for serial and multithreaded applications
  2. Intel Advisor (`advixe-cl`, `advixe-gui`): a set of tools to help ensure Fortran, C, C++, OpenCL, and Data Parallel C++ (DPC++) applications realize full performance potential on modern processors.
  3. Intel Inspector (`inspxe-cl`,`inspxe-gui`): a dynamic memory and threading error checking tool for users developing serial and multithreaded applications on Windows and Linux operating systems.
  4. Intel Trace Analyzer and Collector (`traceanalyzer`): a graphical tool for understanding MPI application behavior, quickly finding bottlenecks, improving correctness, and achieving high performance for parallel cluster applications based on Intel architecture.

Intel VTune Profiler

- Use Intel VTune Profiler to locate or determine:
  1. The most time-consuming (hot) functions in your application and/or on the whole system
  2. Sections of code that do not effectively utilize available processor time
  3. The best sections of code to optimize for sequential performance and for threaded performance
  4. Synchronization objects that affect the application performance
  5. Whether, where, and why your application spends time on input/output operations
  6. Whether your application is CPU or GPU bound and how effectively it offloads code to the GPU
  7. The performance impact of different synchronization methods, different numbers of threads, or different algorithms
  8. Thread activity and transitions
  9. Hardware-related issues in your code such as data sharing, cache misses, branch misprediction, and others
- User Guide

Intel Advisor

- Intel Advisor enables you to analyze your code from the following perspectives:
  1. Discover where vectorization will pay off the most using Vectorization and Code Insights perspective.
  2. Identify CPU-imposed performance ceilings using CPU / Memory Roofline perspective.
  3. Identify high-impact opportunities to offload to a GPU using Offload Modeling perspective.
  4. Identify GPU performance bottlenecks using GPU Roofline Insights perspective.
  5. Prototype threading design options using Threading perspective.
- Get Started Guide

Intel Inspector

- Intel Inspector offers:
  1. Preset analysis configurations (with some configurable settings), as well as the ability to create custom analysis configurations to help you control analysis scope and cost.
  2. Visibility into individual problems, problem occurrences, and call stack information, with problem prioritization and filtering by inclusion and exclusion to help you focus on items that require your attention.
  3. Problem suppressions support to help you focus on only those issues that require your attention, including the ability to:
     - Create suppression rules based on stacks
     - Convert third-party suppression files to the Intel Inspector suppression file format
     - Create and edit suppression files in a text editor
  4. Interactive debugging capability so you can investigate problems more deeply during analysis
  5. A wealth of reported memory errors, including on-demand memory leak detection
  6. Memory growth measurement to help ensure your application uses no more memory than expected
  7. Data race, deadlock, lock hierarchy violation, and cross-thread stack access error detection, including error detection on the stack
- Get Started Guide

Intel Trace Analyzer and Collector

- Use Intel Trace Analyzer and Collector to:
  1. Evaluate profiling statistics and load balancing.
  2. Learn about communication patterns, parameters, and performance data.
  3. Identify communication hotspots.
  4. Decrease time to solution and increase application efficiency.
- Get Started Guide

NVIDIA Visual Profiler I

- a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications
- Focus on the information that matters

  Quickly identify potential performance bottleneck issues in your applications using highly configurable tables and graphical views

- Automated performance analysis

  Perform automated analysis of your application to identify performance bottlenecks and get optimization suggestions that can be used to improve performance

- Unified CPU and GPU Timeline

  View CUDA activity occurring on both CPU and GPU in a unified time line, including CUDA API calls, memory transfers and CUDA launches.

- CUDA API trace

  View all memory transfers, kernel launches, and other API functions on the same timeline

- Drill down to raw data

  Gain low-level insights by looking at performance metrics collected directly from GPU hardware counters and software instrumentation.

NVIDIA Visual Profiler II

- Compare results across multiple sessions

  Confirm performance improvements by comparing against previous sessions

- Analyze data collected from remote systems

  Use the command line profiler using environment variables to collect data from multiple systems and analyze the results in Visual Profiler

- CUDA Dynamic Parallelism

  View timeline for applications that use CUDA Dynamic Parallelism including both host-launched and device-launched kernels and the parent-child relationship between kernels.

- Guided Application Analysis

  Use the guided analysis mode has to get step-by-step analysis and optimization guidance. The analysis results now include graphical visualizations to more clearly indicate the optimization opportunities.

- Power, thermal, and clock profiling

  Observe how GPU power, thermal, and clock values vary during application execution

- Included in CUDA Toolkit

NVIDIA Nsight Systems

- a low overhead performance analysis tool designed to provide nsights developers need to optimize their software.
- Unbiased activity data is visualized within the tool to help users investigate bottlenecks, avoid inferring false-positives, and pursue optimizations with higher probability of performance gains.
- Users will be able to identify issues, such as GPU starvation, unnecessary GPU synchronization, insufficient CPU parallelizing, and even unexpectedly expensive algorithms across the CPUs and GPUs of their target platform.
- NVIDIA Nsight Systems can even provide valuable insight into the behaviors and load of deep learning frameworks such as PyTorch and TensorFlow; allowing users to tune their models and parameters to increase overall single or multi-GPU utilization.
- Included in the NVIDIA HPC SDK
- User Guide