Profilers

Profilers

- Profilers are tools or software programs used to measure and analyze the performance of computer applications.
- The collected data helps developers identify performance bottlenecks, areas of inefficient code, and opportunities for optimization.

Profilers

- Profilers are tools or software programs used to measure and analyze the performance of computer applications.
- The collected data helps developers identify performance bottlenecks, areas of inefficient code, and opportunities for optimization.

- There are two common techniques used in application profilers to collect data about the execution of an application:
  - Instrumentation
  - Sampling

Profilers

- **Instrumentation**:
    - Instrumentation involves modifying the application's code by inserting additional instructions or probes at specific points to gather performance-related data.
    - These probes collect information such as function or method entry and exit times, memory allocations, and I/O operations.
    - When the instrumented code is executed, the probes record the relevant data, which can be later analyzed to identify performance bottlenecks or areas for optimization.

Profilers

- **Instrumentation**:
    - Instrumentation involves modifying the application's code by inserting additional instructions or probes at specific points to gather performance-related data.
    - These probes collect information such as function or method entry and exit times, memory allocations, and I/O operations.
    - When the instrumented code is executed, the probes record the relevant data, which can be later analyzed to identify performance bottlenecks or areas for optimization.
    - Instrumentation-based profilers provide detailed information about the execution flow and behavior of an application. However, they can introduce some overhead and potentially affect the application's performance, especially in cases where a large number of probes are inserted.

Profilers

- **Sampling**:
  - Sampling profilers, collect performance data by periodically sampling the application's state during runtime.
  - Instead of instrumenting the code, these profilers periodically interrupt the application's execution and record the state of the program at that moment, such as the current function or method being executed and the call stack.
  - By sampling the state at regular intervals, the profiler builds a statistical representation of the application's behavior.

Profilers

- **Sampling**:
    - Sampling profilers, collect performance data by periodically sampling the application's state during runtime.
    - Instead of instrumenting the code, these profilers periodically interrupt the application's execution and record the state of the program at that moment, such as the current function or method being executed and the call stack.
    - By sampling the state at regular intervals, the profiler builds a statistical representation of the application's behavior.
    - Sampling profilers provide a lightweight approach to performance analysis since they do not require modifying the code. They have lower overhead compared to instrumentation-based profilers but may have slightly less detailed information about individual function timings.

Profilers

- Following are few profilers available for C/C++:
    - GPROF
    - Intel VTune Amplifier
    - VALGRIND
    - HPCToolkit
    - Perf
    - Google Performance Tools (gperftools)
    - Allinea MAP

Profilers

- Following are few profilers available for C/C++:
  - GPROF
  - Intel VTune Amplifier
  - VALGRIND
  - HPCToolkit
  - Perf
  - Google Performance Tools (gperftools)
  - Allinea MAP

Our focus will remain here

GPROF (GNU Profiler)

The GNU profiler is a widely used profiling tool for C, C++, and Fortran programs. It is part of the GNU Binutils suite and provides statistical profiling information about function timings and call graphs.

gprof provides statistical profiling information, allowing developers to identify performance bottlenecks and optimize their code.

GPROF (GNU Profiler)

The GNU profiler is a widely used profiling tool for C, C++, and Fortran programs. It is part of the GNU Binutils suite and provides statistical profiling information about function timings and call graphs.

gprof provides statistical profiling information, allowing developers to identify performance bottlenecks and optimize their code.

Features supported by GPROF:

- Sampling-based Profiling
- Function-Level Profiling
- Call Graph Analysis
- Flat Profile and Call Graph Profile
- Optimized Code Analysis
- Compiler Instrumentation

GPROF

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing.
- Since the profiler uses information collected during the actual execution of your program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.

GPROF

- Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing.
- Since the profiler uses information collected during the actual execution of your program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.
- Profiling has several steps:
  - Compile and link your program with profiling enabled
  - Execute your program to generate a profile data file
  - Run gprof to analyze the profile data

Compilation

- A program that requires profiling needs to be compiled with the command line option ***-pg***.
  This option can be used along with other compiler options.
  This option can be used for compilation as well ask linking.

```
# gcc         -pg     my_program.c
```

Compilation

- A program that requires profiling needs to be compiled with the command line option ***-pg***.
  This option can be used along with other compiler options.
  This option can be used for compilation as well ask linking.

```
# gcc         -pg     my_program.c
```

- In addition to the '-pg' and '-g' options, you may also wish to specify the '-a' option when compiling. This will instrument the program to perform basic-block counting. As the program runs, it will count how many times it executed each branch of each 'if' statement, each iteration of each 'do' loop, etc.

```
# gcc         -pg     -g     -a      my_program.c
```

Execution

- Once the program is compiled for profiling, it must be executed in order to generate the information that gprof needs.
  The program should run normally, producing the same output as usual.

```
# ./a.out
```

Execution

- Once the program is compiled for profiling, it must be executed in order to generate the information that gprof needs.
  The program should run normally, producing the same output as usual.

```
# ./a.out
```

- The program will write the profile data into a file called ‘gmon.out’ just before exiting.
  If there is already a file called ‘gmon.out’, its contents are overwritten.
  In order to write the `gmon.out' file properly, your program must exit normally:
  by returning from main or by calling exit.

Analysing the profile data

- After profiling, several forms of output are available from the analysis:
  - Flat Profile
  - Call Graph
  - Annotated source

Analysing the profile data

- After profiling, several forms of output are available from the analysis:
  - Flat Profile
  - Call Graph
  - Annotated source

- After you have a profile data file ‘gmon.out’, you can run gprof to interpret the information in it.

```
# gprof options [executable-file [profile-data-file]]
```

If you omit the executable file name, the file ‘a.out’ is used.
If you give no profile data file name, the file ‘gmon.out’ is used.

Interpreting gprof Output

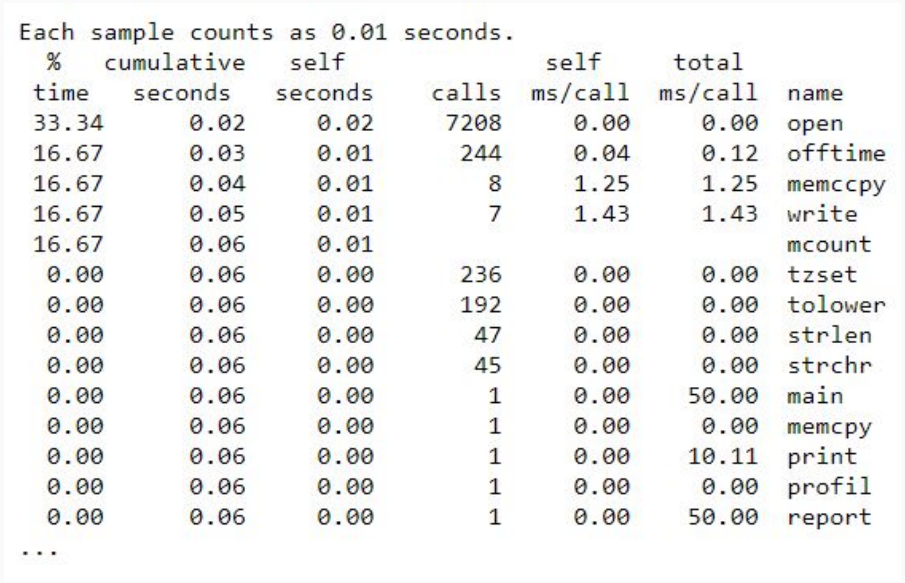
- **Flat Profile**

  The flat profile shows the total amount of time your program spent executing each function. The functions are sorted:
  - First by decreasing run-time spent
  - Then by decreasing number of calls
  - Then alphabetically by name

- The functions ‘mcount’ and ‘profil’ are part of the profiling apparatus and appear in every flat profile; their time gives a measure of the amount of overhead due to profiling.

Interpreting gprof Output

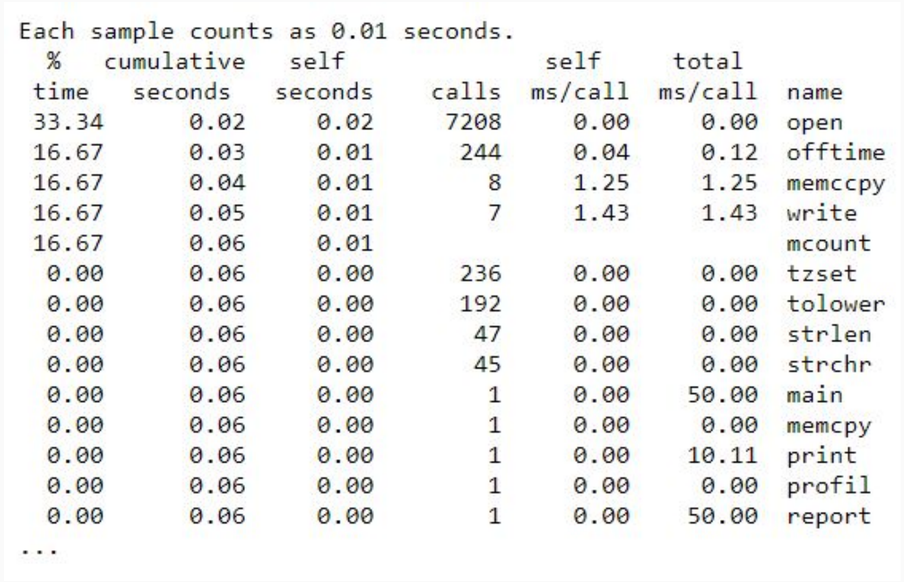
- Following is a Flat Profile for a small program:



```
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
...
```

Interpreting gprof Output

- Following is a Flat Profile for a small program:



```
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
...
```

- %time
  Percentage of the total execution time your program spent in this function
- cumulative seconds
  total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- self seconds
  number of seconds accounted for by this function alone.
- self ms/calls
  average number of milliseconds spent in this function per call
- total ms/calls
  average number of milliseconds spent in this function and its descendants per call

Interpreting gprof Output

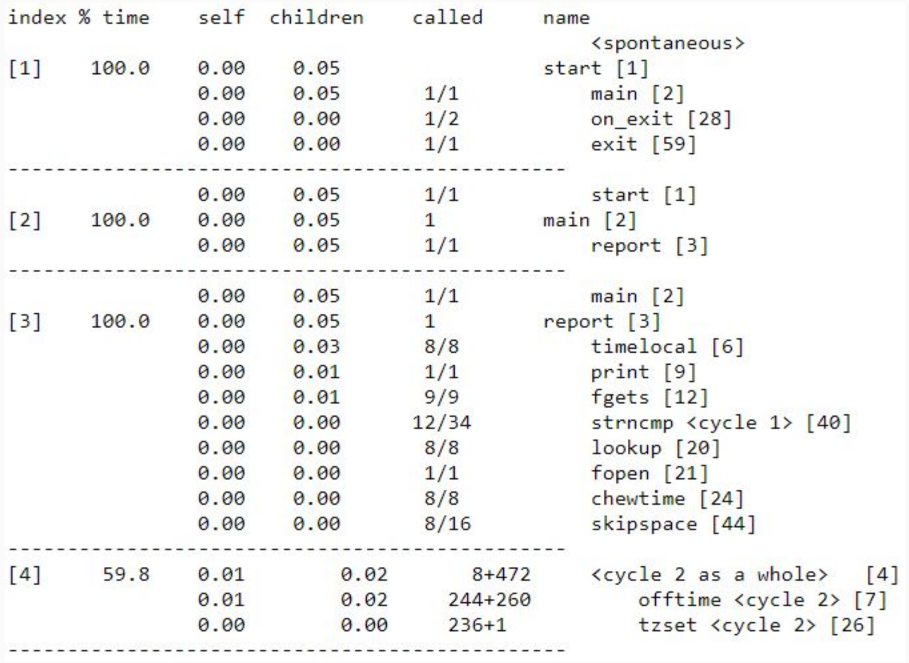
- **Call Graph**

  The call graph shows how much time was spent in each function and its children. The entries are sorted by time spent in the function and its subroutines.

- The internal profiling function ‘mcount’ is never mentioned in the call graph.

Interpreting gprof Output

- Following is the Call Graph for a small program:



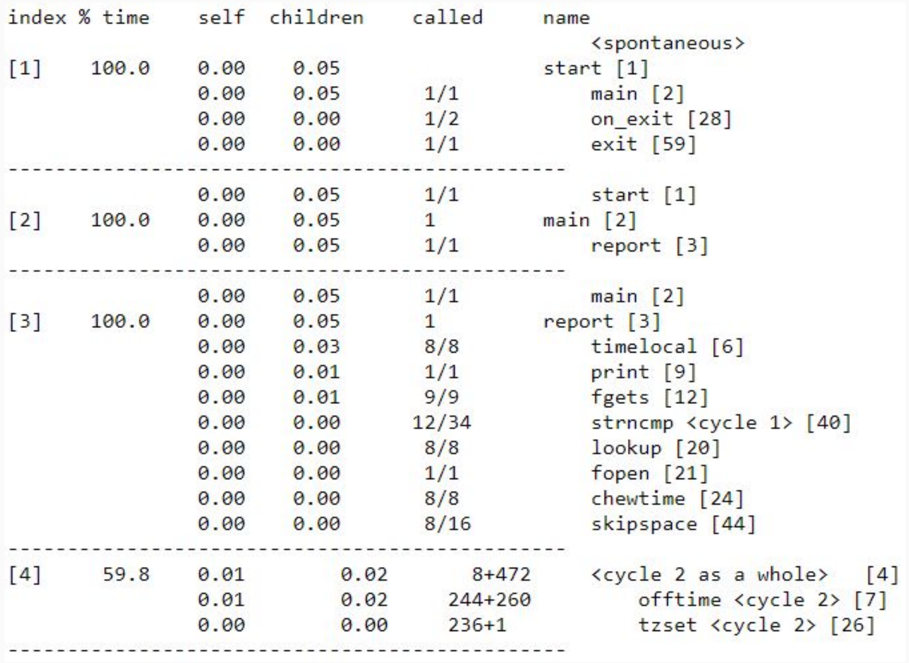
```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                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]
-----------------------------------------------
```

Interpreting gprof Output

- Following is the Call Graph

  for a small program:

  - %time
    Percentage of the total time that was spent in this function, including time spent in subroutines.
  - self
    Total amount of time spent in this function.
  - children
    Total amount of time spent in the subroutine calls made by this function.
  - called
    Number of times the function was called.
    For recursive functions, a `+' separator is used.
    The first number counts non-recursive calls,
    and the second counts recursive calls.



```
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]
-----------------------------------------------
```

Workout 1 (Part 1)

A) Profile the listed program using the techniques learnt above.

B) Note which parts of the analysis data are being used frequently.

C) List the types of profiles that you find useful in the listed program.

D) Understand how would gprof behave in case of distributed memory programs.

```
#include <stdio.h>
#include <math.h>

void do_add(int m, int n, double* A, double* B) {
  for (int i=0; i<m; i++) {
    for (int j=0;j<n; j++) {
      int idx = i*m + j;
      B[idx] += A[idx];
    }
  }
}

void do_mul(int m, int n, double* A, double* B) {
  for (int i=0; i<m; i++) {
    for (int j=0;j<n; j++) {
      int idx = i*m + j;
      B[idx] *= A[idx];
    }
  }
}
```

Workout 1 (Part 2)

A) Profile the listed program using the techniques learnt above.

B) Note which parts of the analysis data are being used frequently.

C) List the types of profiles that you find useful in the listed program.

D) Understand how would gprof behave in case of distributed memory programs.

```
void do_div(int m, int n, double* A, double* B) {
 for (int i=0; i<m; i++) {
   for (int j=0;j<n; j++) {
     int idx = i*m + j;
     B[idx] /= A[idx];
   }
 }
}

void do_sqrt(int m, int n, double* A, double* B) {
 for (int i=0; i<m; i++) {
   for (int j=0;j<n; j++) {
     int idx = i*m + j;
     B[idx] += sqrt(A[idx]);
   }
 }
}
```

Workout 1 (Part 3)

A) Profile the listed program using the techniques learnt above.

B) Note which parts of the analysis data are being used frequently.

C) List the types of profiles that you find useful in the listed program.

D) Understand how would gprof behave in case of distributed memory programs.

```
void do_pow(int m, int n, double* A, double* B) {
 for (int i=0; i<m; i++) {
   for (int j=0;j<n; j++) {
     int idx = i*m + j;
     B[idx] += pow(A[idx],0.17);
   }
 }
 do_add(m, n, A, B);
 do_add(m, n, A, B);
 do_div(m, n, A, B);
}
```

Workout 1 (Part 4)

A) Profile the listed program using the techniques learnt above.

B) Note which parts of the analysis data are being used frequently.

C) List the types of profiles that you find useful in the listed program.

D) Understand how would gprof behave in case of distributed memory programs.

```
int main() {

  const int m = 1000, n = 1000, iter = 1000;
  double A[m*n], B[m*n];

  for (int i=0; i<m*n; i++) B[i] = 0.0;
  for (int i=0; i<m*n; i++) A[i] = 1.1;

  for (int i=0; i<iter; i++) {
    do_add(m, n, A, B);
    do_mul(m, n, A, B);
    do_div(m, n, A, B);
    do_sqrt(m, n, A, B);
    do_exp(m, n, A, B);
    do_log(m, n, A, B);
    do_pow(m, n, A, B);
  }
  printf("Done!\n");
}
```

Workout 1 (Part 4)

A) Profile the listed program using the techniques learnt above.

B) Note which parts of the analysis data are being used frequently.

C) List the types of profiles that you find useful in the listed program.

D) Understand how would gprof behave in case of distributed memory programs.

```
int main() {

  const int m = 1000, n = 1000, iter = 1000;
  double A[m*n], B[m*n];

  for (int i=0; i<m*n; i++) B[i] = 0.0;
  for (int i=0; i<m*n; i++) A[i] = 1.1;

  for (int i=0; i<iter; i++) {
    do_add(m, n, A, B);
    do_mul(m, n, A, B);
    do_div(m, n, A, B);
    do_sqrt(m, n, A, B);
    do_exp(m, n, A, B);
    do_log(m, n, A, B);
    do_pow(m, n, A, B);
  }
  printf("Done!\n");
}
```

Further add the functions:

```
do_sin();
do_cos();
do_tan();
```

with the same signature and report the list of time all consuming functions.

Intel Application Performance Snapshot (APS Profiler)

Intel Application Performance Snapshot (APS) is a profiling tool provided by Intel as part of its performance analysis tools suite.

APS is designed to help developers quickly identify performance bottlenecks in their applications by providing high-level performance insights.

Features supported by GPROF:

- Integration with IDEs
- Minimal Overhead
- High-Level Analysis
- Platform Support
- Performance Metrics
- Visualization and Reporting

Intel APS

- **Setting up Intel Environment:**
  - Download Intel oneAPI toolkits: https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html

Intel APS

- **Setting up Intel Environment:**
  - Download Intel oneAPI toolkits: https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html
  - Install the package using offline installer at a custom permissible directory.

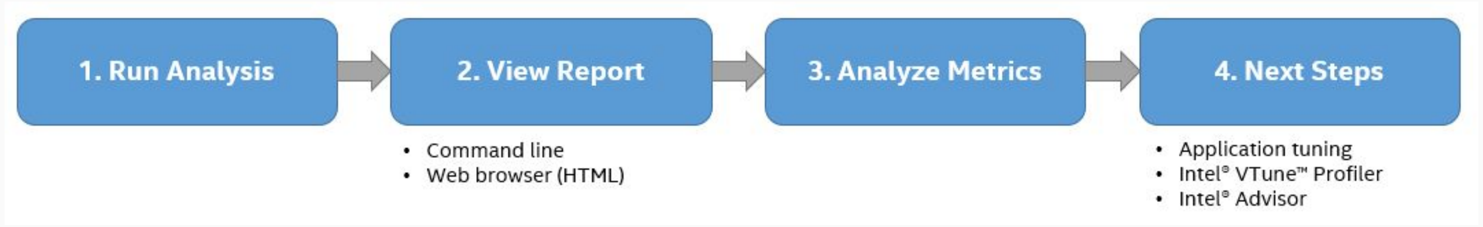
Intel APS

- **Setting up Intel Environment:**
  - Download Intel oneAPI toolkits: https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html
  - Install the package using offline installer at a custom permissible directory.
  - Enable the Intel environment using the ‘source’ command.

```
# source /path/to/oneapi/setvars.sh
```

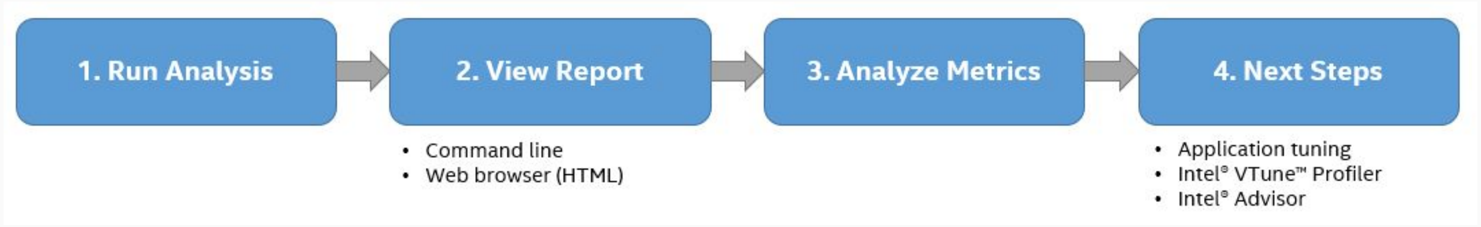
Intel APS

- **Analysing shared memory applications**



Intel APS

- **Analysing shared memory applications**



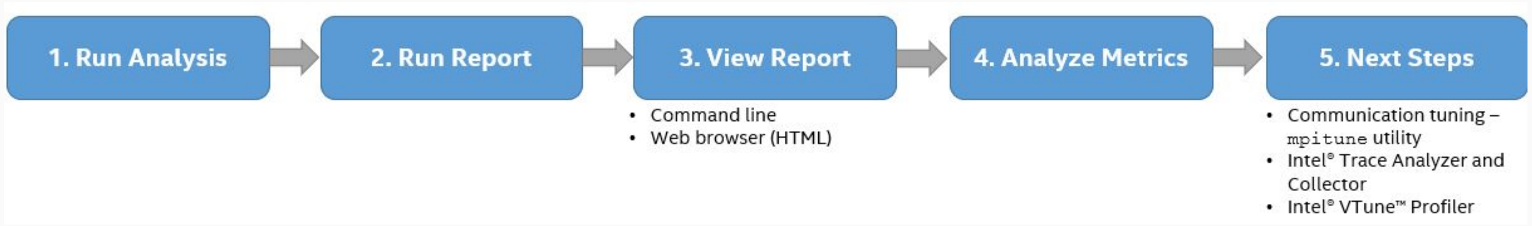
- Profile the application using the aps binary

```
# aps   executable_file
```

Application Performance Snapshot launches the application and runs the data collection. After the analysis completes, a report appears in the command window.

Intel APS

- **Analysing MPI applications**



- Profile the application using the aps binary through mpi launcher

```
# <mpi launcher>   <mpi parameters>   aps   executable_file
```

Application Performance Snapshot launches the application and runs the data collection. After the analysis completes, a aps_result_<date> directory is created.

Intel APS

- Profile the application using the aps binary through mpi launcher

```
# <mpi launcher>   <mpi parameters>   aps   executable_file
```

Application Performance Snapshot launches the application and runs the data collection. After the analysis completes, a aps_result_<date> directory is created.

- Generate the report and complete the analysis.

```
# aps   --report=aps_result_<date>
```

Workout 2

A) Use the same program developed in Workout 1 and profile it using Intel APS.

B) Notice key differences on how profile reports are generated using gprof versus Intel APS.

C) Identify which functions are listed as the bottlenecks in both the tools.

D) List which tool is more convenient to use.

```
int main() {

  const int m = 1000, n = 1000, iter = 1000;
  double A[m*n], B[m*n];

  for (int i=0; i<m*n; i++) B[i] = 0.0;
  for (int i=0; i<m*n; i++) A[i] = 1.1;

  for (int i=0; i<iter; i++) {
    do_add(m, n, A, B);
    do_mul(m, n, A, B);
    do_div(m, n, A, B);
    do_sqrt(m, n, A, B);
    do_exp(m, n, A, B);
    do_log(m, n, A, B);
    do_pow(m, n, A, B);
  }
  printf("Done!\n");
}
```

Further add the functions:

```
do_sin();
do_cos();
do_tan();
```

with the same signature and report the list of time all consuming functions.

“Profiling is like a magnifying glass for code - it helps you uncover the hidden quirks and performance monsters lurking within, turning them into amusing anecdotes for developers' gatherings.”

Questions?

Thanks!

Contact:

Vineet More
vineet.more.bfab@gmail.com
[Make sure to use HPCAP23 as the subject line for your queries]

LinkedIn Handle:
https://www.linkedin.com/in/vineet-more-c-programmer/