Profiling

- Gathering information about a program’s behavior, specifically its use of resources,is called profiling.
- The most important “resource” in terms of high performance computing is runtime.
- hot spots the parts of the program that require the dominant fraction of runtime.

Software Tools:

- Function Profiling and line-based runtime profiling

  Instrumentation: Compiler modify each function call, inserting some code that logs the call, its caller (or the complete call stack) and probably how much time it required.

  Sampling: the program is interrupted at periodic intervals, e.g., 10 milliseconds, and the program counter and possibly the current call stack is recorded.

Function Profiling:

- The most widely used profiling tool is gprof from the GNU binutils package.
- gprof uses both instrumentation and sampling to collect a flat function profile as well as a callgraph profile, also called a butterfly graph.
- When compiling with gcc use the -pg option,this creates a gmon.out file which can be executed with the gprof program.

Flat Profile

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
70.45	5.14	5.14	26074562	0.00	0.00	intersect
26.01	7.03	1.90	4000000	0.00	0.00	shade
3.72	7.30	0.27	100	2.71	73.03	calc_tile

- %time: Percentage of overall program runtime used exclusively by this function, i.e., not counting any of its callees.
- cumulative seconds: Cumulative sum of exclusive runtimes of all functions up to and including this one.
- self seconds: Number of seconds used by this function (exclusive). By default, the list is sorted according to this field.
- calls: The number of times this function was called.
- self ms/call: Average number of milliseconds per call that were spent in this function (exclusive).
- total ms/call: Average number of milliseconds per call that were spent in this function, including its callees (inclusive). 96th

Callgraph Profile:

- Reveals how the runtime contribution of a certain function is composed of several different callers, which other functions (callees) are called from it, and which contribution to runtime they in turn incur.
- Each section of the callgraph pertains to exactly one function, which is listed together with a running index (far left).
- The functions listed above this line are the current function’s callers, whereas those listed below are its callees.

```
index % time    self  children    called     name
                0.27    7.03     100/100         main [2]
[1]     99.9    0.27    7.03     100         calc_tile [1]
                1.90    5.14 4000000/4000000     shade [3]
-----------------------------------------------
                                                 <spontaneous>
[2]     99.9    0.00    7.30                 main [2]
                0.27    7.03     100/100         calc_tile [1]
-----------------------------------------------
                             5517592             shade [3]
                1.90    5.14 4000000/4000000     calc_tile [1]
[3]     96.2    1.90    5.14 4000000+5517592 shade [3]
                5.14    0.00 26074562/26074562     intersect [4]
                             5517592             shade [3]
-----------------------------------------------
                5.14    0.00 26074562/26074562     shade [3]
[4]     70.2    5.14    0.00 26074562         intersect [4]
```

- % time The percentage of overall runtime spent in this function, including its callees (inclusive time). This should be identical to the product of the number of calls and the time per call on the flat profile.

- self: For each indexed function, this is exclusive execution time (identical to flat profile). For its callers (callees), it denotes the inclusive time this function (each callee) contributed to each caller (this function).

- children: For each indexed function, this is inclusive minus exclusive runtime, i.e., the contribution of all its callees to inclusive time.

- called: denotes the number of times the function was called.

Line-based Profiling:

- Function profiling becomes useless when the program to be analyzed contains large functions (in terms of code lines) that contribute significantly to overall runtime:
- If the hot spot in such functions cannot be found by simple common sense, tools for line-based profiling should be used. Ex: OProfile
- With OProfile, the only prerequisite the binary has o fulfill is that debug symbols must be included (usually this is accomplished by the -g compiler option).
- A profiling daemon must be started (usually with the rights of a super user), which subsequently monitors the whole computer and collects data about all running binaries.
- The user can later extract information about a specific binary.
- This can be an annotated source listing in which each source line is accompanied by the number of sampling hits and the relative percentage of total program samples

```
1                    :        DO 215 M=1,3
2   4292   0.9317   :           bremsdir(M) = bremsdir(M) + FH(M)*Z12
3   1462   0.3174   : 215    CONTINUE
4                    :
5    682   0.1481   :        U12 = U12 + GCL12 * Upot
6                    :
7                    :        DO 230 M=1,3
8   3348   0.7268   :           F(M,I)=F(M,I)+FH(M)*Z12
9   1497   0.3250   :           Fion(M)=Fion(M)+FH(M)*Z12
10   501   0.1088   :230     CONTINUE
```

Hardware Performance Counters:

- modern processors feature a small number of performance counters (often far less than ten), which are special on-chip registers that get incremented each time a certain event occurs.

Events that can be monitored that are most useful for profiling:

- Number of bus transactions, i.e., cache line transfers.
- Number of loads and stores. Together with bus transactions, this can give an indication as to how efficiently cache lines are used for computation. If, e.g., the number of DP loads and stores per cache line is less than its length in DP words, this may signal strided memory access.
- Number of floating-point operations.
- Mispredicted branches. This counter is incremented when the CPU has predicted the outcome of a conditional branch and the prediction has proved to be wrong.
- Pipeline stalls. Dependencies between operations running in different stages of the processor pipeline can lead to cycles during which not all stages are busy, so-called stalls or bubbles.
- Number of instructions executed. Together with clock cycles, this can be a guideline for judging how effectively the superscalar hardware with its multiple execution units is utilized.

- lipfpm tool:to get a quick overview of the performance properties of an application, measure overall counts from start to finish and probably calculate some derived metrics like “instructions per cycle” or “cache misses per load or store.”

```
1  CPU Cycles.......................................... 28526301346
2  Retired Instructions................................ 15720706664
3  Average number of retired instructions per cycle..... 0.551095
4  L2 Misses........................................... 605101189
5  Bus Memory Transactions............................. 751366092
6  Average MB/s requested by L2........................ 4058.535901
7  Average Bus Bandwidth (MB/s)........................ 5028.015243
8  Retired Loads....................................... 3756854692
9  Retired Stores...................................... 2472009027
10 Retired FP Operations............................... 4800014764
11 Average MFLOP/s..................................... 252.399428
12 Full Pipe Bubbles in Main Pipe...................... 25550004147
13 Percent stall/bubble cycles........................ 89.566481
```

Manual instrumentation:

- If the overheads subjected to the application by standard compiler-based instrumentation are too large, or if only certain parts of the code should be profiled in order to get a less complex view on performance properties, manual instrumentation may be considered.
- Profiling libraries like PAPI can be used for manual profiling.