PROFILING OF CODE

Q. How to understand where time is consumed in a program?

- Time complexity g(n) = (O(f(n)): g(n) <= **c**.f(n) for all n >= **n0**, Theta(f(n))) [c, n0 are ignored, as time complexity becomes harder to compute if we take all parameters.]
- **CORMEN: RAM MODEL** of computation
- Assumes that all systems take a constant amount of time for computation. [Memory hierarchy performs an important role in deciding performance.]
- If considering matrix multiplication, use libraries like intel's, python, etc. or, compute in blocks. [strassen's mult. also has some problems, when adding terms at the end., so don't use strassen's mult, use its optimized version.]
  (Quick sort is cache friendly, so it is used more often than merge sort.)

  Instead of straight-away running the program:

1) use **gpof** (gnu profiler)
   a) [g++ -PG] (flag while compiling)
   b) gpof_annotate ./a.out

   Time spent %time Line content of source code

2) **cProfiler** in python and so on…

- In python, hard to calculate execution time
  [bcz, translate into m/c code, sent to processor for execution] in python, java, but not in c++] initially it's slow, but then it speeds up, in successive executions, lastly it converges.
- Therefore, Enclose the code in a loop, and then run it for many times (100 or 50).. Then take the average over 100 runs.

3) **The valgrind tool**:

a) memcheck:

```
Int x[10];
x[10] = 2; //x[9] is the last element
        //*(x+10): this can create an error at any point in the future.
```

- THIS IS A LOGICAL ERROR, C++ MIGHT GIVE THIS ERROR SOMETIMES, AND SOMETIMES NOT.
- Running it through valgrind's memcheck will ensure that these problems do not occur.
- memcheck Identifies:
  1. Out-of-bound memory access
  2. Memory leak
  3. Double free() called
  4. Accessing memory using uninitialized pointers
     e.g. int *a;
     *a = 5; //sometimes c++ might store garbage and not give errors sometimes.

- Disadvantage: incurs time overhead, makes program slow.

b) cachegrind:
(floating point operations are significantly slower than integer operations)

- **./valgring tool = cachegrind ./a.out** //cache miss for each line of code.
- Simulation of the cache. [you can define cache sizes]
- Disadvantages: slower, doesn't represent the real environment.
  It has a simple cache model.
  assumes that all caches are organized in the same way, therefore it is less accurate sometimes.

IS MORE ACCURATE PROFILING POSSIBLE?

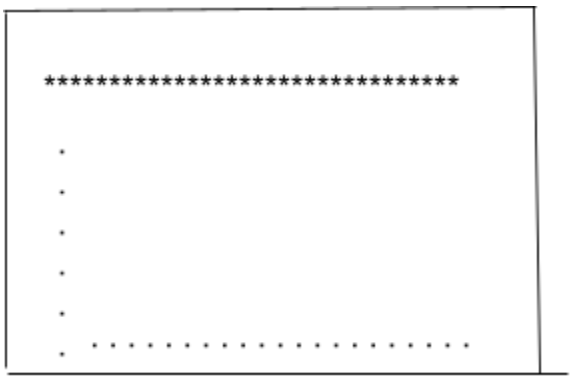
4) **perf tools:** [Performance registers]

- Allows access to these registers to identify performance.
- Advantage: Very accurate in practise, directly represents actual hardware details.
- Disadvantage: No line-by-line profiling is possible, especially in source code, bcz the binary file doesn't have the source code written anywhere in it.
  (thus, you've to get assembly code and from there you can get line-by-line profiling)
- Perf tools can even give power consumed by a program

Floating point addition(10X) and multiplication
VS
integer addition and multiplication(5X)

- floating point computations are Most widely supported operation on GPUs, bcz they provide more accuracy, which is actually important.

1) Threads **X** **(after midsem)**
2) Data-level parallelism:
   - Some sort of vector operations in CPUs, which run a total of 64-128 operations (additions/ multiplications) using a single instruction.
     e.g. A[i][j] = B[i][j] + C[i][j] (same operation)
     Single instruction Multiple Data / Data-Level Parallelism
     Using SSE/ AVX



GPUs: Graphical Processing Unit -> nVidia: CUDA [OpenCL]: Requires C, C++.