CO339 - Performance Engineering (60017)

Introduction

Performance is how well a person / machine does work or activity. However we need to define **well**, which is something that is perceived by the user of the code. We first assume that the software is bug-free (functional), and the perceived quality is therefore usually associated with speed of execution (treat it as end-to-end execution for now).

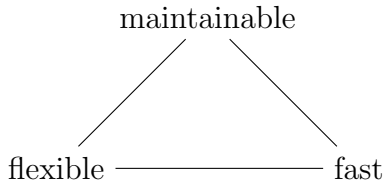
High performance computing is usually focused on a single problem (such as fluid dynamics, weather simulation, etc). The problem is usually 'high value', with some significant benefit, is relatively simple (narrowly scoped, however speed is difficult), and is occasionally supported by special hardware, such as GPUs or tensor processing units built to support ML workloads.

However, performance engineering focuses on systems (complex, flexible pieces of software). Examples of systems include big data systems, DBMS, AI systems (such as *Tensorflow*, *Torch*, etc), as well as operating system kernels and operating system tools (such as `awk`, `grep`, `sed`).

A system is often made up of **components** which interact to achieve a greater goal. These are usually applicable to multiple problems or domains (they are generic); this differs from a well-designed application as the goal is domain-agnostic, hence they are designed to be flexible at runtime - for example, we don't want to recompile `grep` every time we want to search for a string.

The need for flexibility arises from the fact that we do not know the exact operating conditions for a system during development. For example, a data management system will not know the schema beforehand (could be sensor data, or user data etc). Another example is an OS not knowing the number of users, *Tensorflow* doesn't know the dimensionality of the data, or `grep` not knowing what regular expression we intend to search for.

Software systems are complex and developed over years, and therefore must be maintainable. They also need to be fast. The challenge is to build a system which strikes a balance, within the classic trade-off triangle (where we choose two of the three);



Instead of having only flexible and fast code, we need to think about what parts should be prioritised for flexibility, what parts for speed, and what parts should be maintainable.

Example

The challenge here is to count lines in a CSV file which contain the phrase " FAUST:" - two spaces, followed by `FAUST`, followed by a colon. This could be done with the command;

```
egrep -c '^  FAUST:$' faust.txt
```

An example for this, in C++, would be as follows;

```
class Operator {
public:
  virtual char const* getNextString() = 0; // abstract class
};
```

```
5
6  class ArrayReader : public Operator {
7    char const* inputArray;
8    bool isRead = false;
9  public:
10   ArrayReader(char const* array):inputArray(array){};
11   char const* getNextString() { // only return once
12     if (isRead)
13       return nullptr;
14
15     isRead = true;
16     return inputArray;
17   }
18 };
19
20 class LineSeparator : public Operator {
21   Operator&& input;
22   char const* currentString = nullptr; // string from input
23 public:
24   LineSeparator(Operator&& input):input(std::move(input)){};
25   char const* getNextString() {
26     if (currentString == nullptr)
27       currentString = input.getNextString();
28     auto result = currentString; // keep a backup
29     if (currentString != nullptr) { // we want to split this
30       // keep iterating until new line or end of string
31       while (*currentString != '\n' && *currentString != '\0')
32         currentString++
33
34       if (*currentString == '\n')
35         currentString++; // advance to string after
36       else if (*currentString == '\0')
37         return nullptr; // reached end of string
38     }
39     return result
40   }
41 };
42
43 class LineMatcher : public Operator {
44   Operator&& input;
45 public:
46   LineMatcher(Operator&& input):input(std::move(input)){};
47   char const* getNextString() {
48     for (auto candidate = input.getNextString(); candidate != nullptr; candidate =
         input.getNextString()) {
49       if (candidate[0] == ' ' && candidate[1] == ' ' && candidate[2] == 'F' &&
50           candidate[3] == 'A' && candidate[4] == 'U' && candidate[5] == 'S' &&
51           candidate[6] == 'T' && candidate[7] == ':' && candidate[8] == '\n')
52         return candidate;
53     }
54     return nullptr;
55   }
56 }
```

```
57
58 class LineCounter {
59   Operator&& input;
60 public
61   LineCounter(Operator&& input):input(std::move(input)){};
62   unsigned long getCount() {
63     auto result = 0UL;
64     for (auto inputLine = input.getNextString(); inputLine != nullptr, inputLine =
           input.getNextString()) {
65       result++;
66     }
67     return result;
68   }
69 }
70
71 int main(int argc, char *argv[]) {
72   auto fd = open(argv[1], O_RDONLY);
73   auto fileSize = lseek(fs, 0L, SEEK_END);
74   lseek(fd, 0UL, SEEK_SET); // reset to beginning
75   auto data = (char*)mmap(nullptr, fileSize, PROT_READ, MAP_SHARED, fd, 0);
76   auto result = LineCounter(LineMatcher(LineSeparator(ArrayReader(data)))).
       getCount();
77   printf("%lu\n", result);
78   return 0;
79 }
```

However, this is a significant amount slower than `grep`. While this was a good implementation from an object oriented view, it's bad for performance.

```
1 int main(int argc, char *argv[]) {
2   auto fd = open(argv[1], O_RDONLY);
3   auto fileSize = lseek(fs, 0L, SEEK_END);
4   lseek(fd, 0UL, SEEK_SET); // reset to beginning
5   auto data = (char*)mmap(nullptr, fileSize, PROT_READ, MAP_SHARED, fd, 0);
6   auto result = 0;
7   // stop 9 chars before end
8   for (auto i =0; i < fileSize - 9; i++) {
9     if (data[i] == '\n' &&
10        data[i+1] == ' ' && data[i+2] == ' ' && data[i+3] == 'F' &&
11        data[i+4] == 'A' && data[i+5] == 'U' && data[i+6] == 'S' &&
12        data[i+7] == 'T' && data[i+8] == ':' && data[i+9] == '\n')
13       result++;
14   }
15
16   printf("%lu\n", result);
17
18   return 0;
19 }
```

While this code, compiled with `-O3` is better than the first implementation, it's still slightly slower than `grep`. We can do better (only the loop this time);

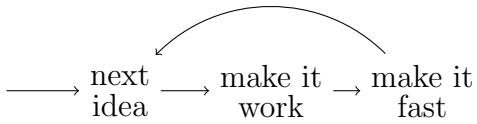
```
1 for (auto i =0; i < fileSize - 9; i++) {
2   long word = *(long*)"  FAUST:"; // gives a bit pattern representing the string
3   if (*(long*)(data + i) == word && data[i+8] == '\n')
4     result++;
```

```
5 }
```

This can be compiled for native (and also gives better performance than `grep`);

```
clang++ -O3 -march=native -mtune=native count.cpp
```

This follows the following process, where we have an idea, make a functional solution (well designed), and then improving the performance;



Target

One question is to define how fast is fast enough.

1. **define a target metric**

   - **throughput** how many runs can be performed in a given time
   - **latency** trade-off with throughput (what we've been looking at)
   - **scalability** how well it works with more machines
   - **memory usage** stay within memory constraints
   - **energy consumption** energy related to cooling in data centres
   - **TCO** total ownership cost
   - **efficiency** work over time

2. **decide when requirements are met**

   An easy approach to this it to set an optimisation budget (in terms of developer time / cost). For example, we can ask how much a customer is willing to spend to optimise something.

   On the other hand, we can set a target, requirement, or threshold (at least this fast). This might have real-time requirements; soft (if it misses the requirements, it's an error), or hard (if it misses the requirement, it's considered a failure). These are often referred to as **Quality-of-Service (QoS)** objectives.

   These objectives are statistical properties of a metric that must hold for a system. For example, we can put a condition for a game to run at 60 FPS or higher on a GPU with 50 GFlops (or higher) - here we have have something that must be fulfilled once a certain condition is satisfied. This can sometimes conflict with functional requirements (the previous example may conflict with the realism of the AI in a game).

   Service-Level Agreements (SLAs) are related to (and more important than) QoS objectives. These are legal contracts specifying QoS objectives as well as **penalties for violations**. Since these are non-functional requirements, it can be difficult to tell (for example execution time), but still should be enforced. Requirements should be defined with the SMART acronym in mind;

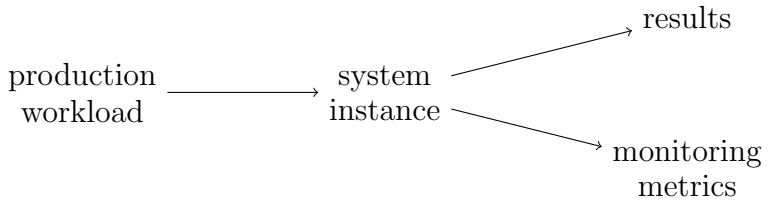
   - **s**pecific acceptable must be stated exactly in numerical terms ('fast' is insufficient)
   - **m**easurable can be measured (time unit being below clock granularity is impossible)
   - **a**cceptable guarantees user's success in reality
   - **r**ealisable must be possible to implement
   - **t**horough all necessary aspects are specified (consider trade-offs between speed and memory)

Performance Evaluation Techniques

Measuring comes in two forms; **monitoring** and **benchmarking**. Another technique is **analytical modelling**, which is related to **simulation**. Hybrids also exist.

Measuring can be performed on the actual system, either on a prototype or the final system. If this is done properly, it can achieve good accuracy (closest thing to quantifiable performance). Based on **instrumentation**, and can be difficult as we want to reduce the effect of the benchmarking on the actual performance of the production system.

Monitoring is measuring in production. Constant monitoring is required to enforce SLAs, by observing and collecting statistics about system performance. The data is then analysed and violations reported. However, by performing the monitoring, it can be costly on performance, hence this isn't always done continuously (samples are taken and then extrapolated).



The alternative is **benchmarking**, which is measuring in the lab. Since some systems run continuously, they can store state, therefore the first step in benchmarking is to put the system into a **pre-defined / steady state**, which is reasonably close to what a system would be in production. After this, a series of operations (the workload) is performed with relevant metrics measured. For example, to benchmark a database system, some data is typically generated and loaded to put the system into a steady state, and a query set is run (which we actually measure metrics for).