Performance-related aspects in the Big Data Astronomy Era: architects in software optimization

Daniele Tavagnacco - INAF-Observatory of Trieste on behalf of EUCLID SDC-IT
Design and Optimization
EUCLID mission

- ESA medium class space mission
- Universe expansion, dark energy, dark matter, gravity
- launch 2022
- 15,000 deg$^2$ survey, 6 years
- 2 instruments: VISible imager, Near Infrared SPectrograph
- $\sim 10^9$ observed sources, $\sim 10^6$ sources with spectrum
- $> 15$ PB data
- lookback $\sim 10$ billion years (z~2)

image credits: ESA
Design and Big Data

EUCLID: 15 PB of data to be processed reduced

- technical only
- technical + human: understandable

Big Data: datasets difficult to process in acceptable time frame or cost range

1- Computing power
   a) powerful machines
   b) parallelize computations

2- Design/optimize algorithms
SDC-IT level3 activities

Euclid computing infrastructure:
- 3 levels: level1 (collect), level2 (prepare), level3 (science)
- distributed infrastructure (--> rules)
  - common environment for sw
  - minimize effort in production and testing (common development tools, test tools, ...)

SDC-IT supervise level3 Galaxy Clustering software:
- integration in EC framework (C++, Python, 3rd party sw like swarp, sextractor, h5py...)
- software porting in C++ or Python
- support for refactoring and optimization
- deployment in CI environment
Two Point Correlation Function GC example

Before... on Euclid estimated size of $\sim 10^7$ objects...after measured on simulated catalogs

Before... on Euclid estimated size of $\sim 10^7$ objects

EC: technical budget note, 2015

$\max r = 200$ fixed

$\log_{\text{RAM}}(\text{Gbytes}) = A \log_{\text{N}}(\text{N}) + B$

$N_{\text{random}} = 50 \times N_{\text{objects}}$

$\text{Number of ~22h}$

$\text{Time (hrs)}$

Scaling of the peak memory usage with the data catalog size $N_{\text{p}}$ when $N_{\text{p}} = 50 \times N_{\text{q}}$

Scaling of the computation time with the data catalog size $N_{\text{p}}$ when $N_{\text{p}} = 50 \times N_{\text{q}}$

EC: LE3_GC_2PCF_SDD, 2017

$\log_{\text{RAM}}(\text{Gbytes}) = A \log_{\text{N}}(\text{N}) + B$

$N_{\text{random}} = 50 \times N_{\text{objects}}$

$\text{Number of ~22h}$

$\text{Time (hrs)}$

Scaling of the peak memory usage with the data catalog size $N_{\text{p}}$ when $N_{\text{p}} = 50 \times N_{\text{q}}$

Scaling of the computation time with the data catalog size $N_{\text{p}}$ when $N_{\text{p}} = 50 \times N_{\text{q}}$

EC: LE3_GC_2PCF_SDD, 2017
Software design

Activity performed before writing any line of code

Aimed at reducing:
- rigidity - any change affects many parts of the system
- fragility - change breaks unexpected parts of the system
- immobility - code hard to reuse because it cannot be disentangled

Based on:
- Single Responsibility - each entity has only one responsibility
- Open/Close - entities open for extension, closed for modifications
- Liskov’s Substitution - Open/Close applied to behaviour
- Interface Segregation - avoid general purpose interfaces
- Dependency Inversion - decoupling high-level /low-level modules with interfaces
Scientific software: how good is design?

Software is a collection of modules that:
- operate in harmony
- have simple APIs
- hide complexity internally

Requirements change during lifetime:
- extend functionalities
- maintain reliability when extending
- reuse parts of the code

The quantity of data to be reduced is increasing:
- code scalability
- how many data are “big data”
Optimization within Euclid GC

Scientific software:
- has special life cycle
- mainly developed by scientists
- No a priori requirements

Refactoring the code:
- more understandable
- cleaner and tidier
- removing redundancies and unused code
- generalize to allow reuse
- change internal structure
  (smooth flow, avoid nested conditions)
- improve performance

Compiler result

```cpp
void loopVectorA(vector<int> & vec) {
    int c = 0;
    // simple loop
    for (size_t ivec=0; ivec<vec.size(); ivec++) {
      c += vec[ivec];
    }
}

void loopVectorB(vector<int> vec) {
    int c = 0;
    // optimized loop
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
      c += *it;
    }
}
```

Optimized code

```cpp
loopVectorA(std::vector<int, std::allocator<int> >&): rep ret
loopVectorB(std::vector<int, std::allocator<int> >): rep ret
```
Optimization within Euclid GC

Scientific software:
- has peculiar life cycle
- mainly developed by scientists
- No a priori requirements

Refactoring the code:
- more understandable
- cleaner and tidier
- removing redundancies and unused code
- generalize to allow reuse
- change internal structure
  (smooth flow, avoid nested conditions)
- improve performance (know the tool)

```cpp
void loopVectorA(vector<int> & vec) {
    int c = 0;
    // simple loop
    for (size_t ivec=0; ivec<vec.size(); ivec++) {
        c += vec[ivec];
    }
}

void loopVectorB(vector<int> vec) {
    int c = 0;
    // optimized loop
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
        c += *it;
    }
}
```

Compiler result:
```
1 loopVectorA(std::vector<int, std::allocator<int>> &):
2 rep ret
3 loopVectorB(std::vector<int, std::allocator<int>>):
4 rep ret
```

Optimized code
Optimization within Euclid GC

Scientific software:
- has special life cycle
- mainly developed by scientists
- No a priori requirements

Refactoring the code:
- more understandable
- cleaner and tidier
- removing redundancies and unused code
- generalize to allow reuse
- change internal structure
  (smooth flow, avoid nested conditions)
- improve performance (know the tool)

```python
>>> if os.path.exists("filename"):
    ...
    os.remove("filename")
    ...
    f = open("filename","w")
# ------

>>> if not os.path.exists("dirname"):
    ...
    os.makedirs("dirname")
```

```python
f = open("filename","w+")
# ------
os.makedirs("dirname")
```
Need to become a code expert?

Source code

```c
void Test1(int a, int b, int c, int d) {
    int res = a;
    res += b;
    res += c;
    res += d;
}

void Test2(int a, int b, int c, int d) {
    int res = a + b + c + d;
}
```

Optimized code by compiler

```c
1 Test1(int, int, int, int):
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-20], ed1
  mov DWORD PTR [rbp-24], es1
  mov DWORD PTR [rbp-28], edx
  mov DWORD PTR [rbp-32], eax
  mov eax, DWORD PTR [rbp-20]
  mov DWORD PTR [rbp-24], eax
  mov eax, DWORD PTR [rbp-28], eax
  mov eax, DWORD PTR [rbp-32], eax
  add DWORD PTR [rbp-4], eax
  add DWORD PTR [rbp-4], eax
  add DWORD PTR [rbp-4], eax
  pop rbp
  ret

2 Test2(int, int, int, int):
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-20], ed1
  mov DWORD PTR [rbp-24], es1
  mov DWORD PTR [rbp-28], edx
  mov DWORD PTR [rbp-32], eax
  mov eax, DWORD PTR [rbp-20]
  mov edx, DWORD PTR [rbp-28]
  add edx, eax
  add edx, eax
  add edx, eax
  mov eax, DWORD PTR [rbp-24]
  add eax, edx
  add eax, edx
  mov DWORD PTR [rbp-4], eax
  pop rbp
  ret
```

Compiled code

1reg

2reg
What a compiler does?

From source...

Lexical analysis → Syntax analysis → Semantic analysis

Software architect can make difference here

Code generation → Code optimization → Intermediate code generation

...to machine code
Maintaining the code

Revisit the code adopting new features provided by language evolution

```
// Sum first 1000 numbers
int sum1() {
    int sum = 0;
    for (int i=0; i<=1000; ++i) {
        sum += i;
    }
    return sum;
}
```

```
sum1():
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], 0
mov DWORD PTR [rbp-8], 0
jmp .L2
.L2:
    mov eax, DWORD PTR [rbp-8]
    add DWORD PTR [rbp-4], eax
    add DWORD PTR [rbp-8], 1
.cmp DWORD PTR [rbp-8], 1000
    jle .L3
    mov eax, DWORD PTR [rbp-4]
    push rbp
    pop rbp
    ret
sum2():
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], 1000
mov eax, DWORD PTR [rbp-4]
add eax, 1
imul eax, DWORD PTR [rbp-4]
mov edx, eax
shr edx, 31
add eax, edx
sar eax
mov DWORD PTR [rbp-8], eax
mov eax, DWORD PTR [rbp-8]
pop rbp
ret
```

code

compiled

optimized C++11

optimized C++14
The role of human (scientist) architect

Design code *properly* (scalability, maintenance, extension,...)

Consider performance when *designing code* and *picking algorithms*

Adopt *optimized* features provided by language evolution

Know the *tool*: C++ is not Fortran, Python is not IDL

When optimize don’t rely only on “tips&tricks”

If you use C++, trust the compiler, it contains 45 years of improvements...
Q&A

[Comic strip showing a conversation about passing the salt.]

CAN YOU PASS THE SALT?

I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

http://xkcd.com/974