Tips and tricks for good (and fast) scientific programming, with and introduction to parallel computing

2 – Writing high-performance code

Lecturer: Federico Perini

Madison, 2013/12/04

Lecture series outline

1. **Basics of good programming practice**
   - Tools for good and comfortable code development and maintenance
   - Good programming practice

2. **Optimization and Profiling**
   - Basics of computer operations
   - Basic techniques for high-performance coding
   - Making KIVA faster

3. **Parallel programming**
   - Different tools for different applications
   - Examples of code parallelization with OpenMP and MPI

This schedule is open to changes upon requests!

Writing high-performance code

One of the major achievements of science in the XX century is **complexity**

Even if single phenomena can be described by
reasonably simple laws,
when many phenomena are strongly inter-linked
very small perturbations can lead to extremely complex, sometimes chaotic, behavior

⇩

Understanding and predicting their behavior would not have been possible without the development of numerical analysis and scientific computing

Writing high-performance code

Any scientist needs to find a tradeoff between

- The **resolution** of his problem → spatial and temporal scales
- The availability of computational resources

↓

1. Choose/study suitable algorithms for his class of problems
   - E.g., an efficient engine simulation with 10000 cells and 10 species = 170.000 ODEs requires different algorithms than the same simulation with 500k cells and 200 species = 10.35M ODEs

2. Fine-tune the code to optimize its performance

Even the best algorithm would be **useless** if most of the computational time is spent multiplying zeroes, or looping through memory regions to find the data, repeating the same calculations multiple times, or requiring unreasonably high accuracy

Evolution of CPU architectures

"Computers in the future may weigh no more than 1.5 tons"
Popular Mechanics, 1949

Vector processors
Pipelined CPUs
Superscalar CPUs
Out-of-order CPUs
Super-pipelined CPUs
SIMD units
SMT CPUs
Multi-core CPUs
Massively parallel GPUs

Manycore CPUs
…

$\downarrow$ time

Evolution in computing processors is towards parallelization, i.e., completing multiple tasks at the same time

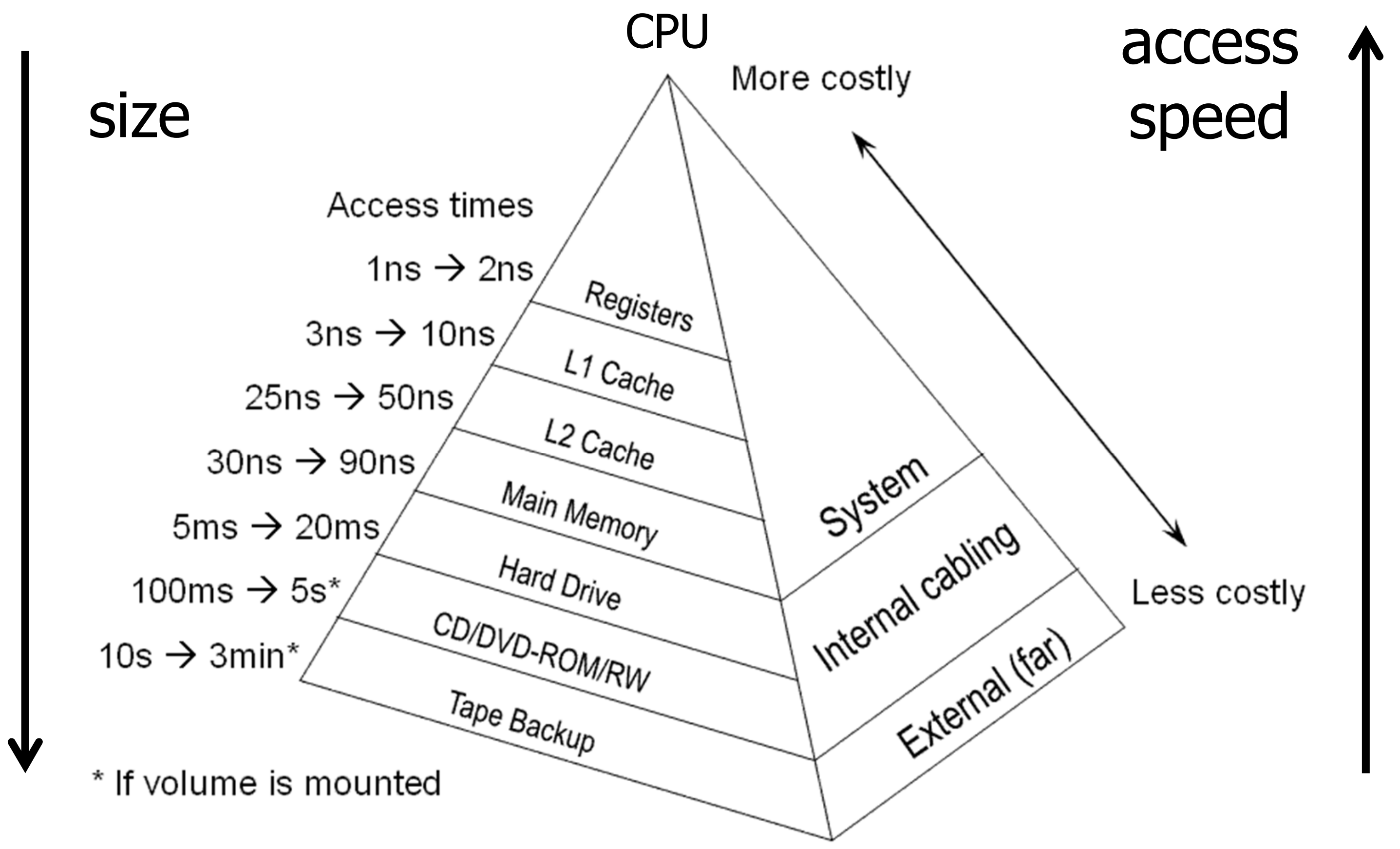
$\downarrow$

The time required for completing a single task is limited by the number of operations → the clock frequency

$\downarrow$

Increase in clock frequency has been slowing down due to transistor scaling problems (Voltage not scaling as size!)

Memory hierarchy



Finding (reading) data

- The closer to the CPU, the faster
- Only few data are needed for simple operations
- Put what’s not currently needed in the slower levels
- Caching → e.g. use of intermediate scalars

1. Look up L1 cache (1-5 cycles)
2. Look up L2 cache (20 cycles)
3. Look up RAM (more cycles)
4. Copy from RAM to L2
5. Copy from L2 to L1
6. Copy into registers for operation

Finding (reading) data

- Importance of **data contiguity**

```
do i = 1, 100
    do j = 1, 50
       c(i) = c(i) + a(i,j) * b(i,j)
    end do
end do
```

i = 1, j = 1
- Look for a(1,1) in L1 → maybe *cache miss*
- Load from RAM
- Copy from a(1,1) to a(8,1) into L1 *(cacheline)*
- Copy a(1,1) into a register
- Look for b(1,1) in L1 → maybe *cache miss*
- Load from RAM
- Copy from b(1,1) to b(8,1) into L1 *(cacheline)*
- Copy b(1,1) into a register
- Calculate a(1,1)*b(1,1)

The data block fetched from the main memory

Finding (reading) data

- Importance of **data contiguity**

```
do i = 1, 100
    do j = 1, 50
       c(i) = c(i) + a(i,j) * b(i,j)
    end do
end do
```

i = 1, j = 2
- Look for a(1,2) in L1 → *cache miss*

↓

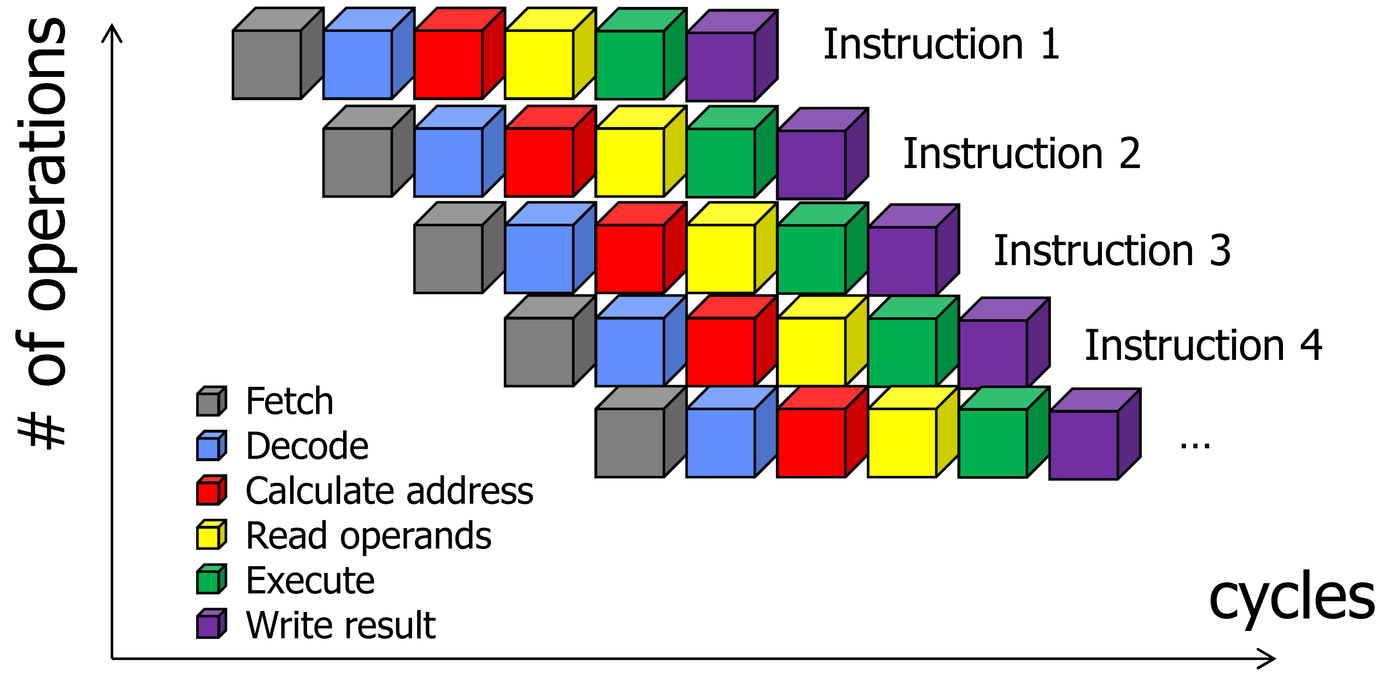
Need to load a(1,2) to a(8,2) from RAM, maybe trashing the data a(1,1) to a(8,1) previously loaded

➔ Inverting the i and j loops allows to **maximize cache hits**
➔ The compiler can do that, if we use intrinsic functions

```
c = c + sum(a*b, dim=2)
```

Pipelines

- Sequences of *independent* stages needed to complete an instruction
- In pipelined CPUs, multiple stages run at the same time for different instructions



The necessary CPU time

Operation	CPI (cycles per instruction)
Sum	12
Product	12
Division	20
sqrt	24
sin, cos	52
tan	100
log	60
exp, power	130
Condition evaluation	70

- Every operation is the result of multiple simple floating point calculations
- Complex arithmetic is expensive for computers, too
- With pipelining we can improve CPI on **parallelizable** loops

➔ No conditional clauses within loops!

The necessary accuracy

- Due to round-off and truncation in floating point arithmetic even the actual sequence of operations will change the results

$$(a+b)+c \neq a+(b+c)$$

- Real and integer numbers count!

~24 cycles	$x^3 \neq x^{3.0}$	~130 cycles
~24 cycles	$\sqrt{x} \neq x^{0.5}$	~130 cycles
~36 cycles	$x\sqrt{x} \neq \sqrt{x^3} \neq x^{\frac{3}{2}}$	~130 cycles

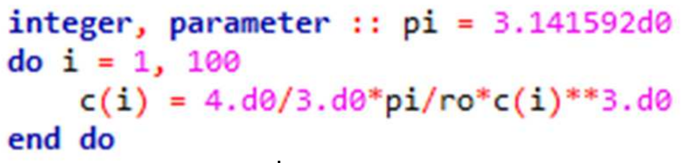
- Store and retrieve, if possible (e.g., tables)
- Importance is that the overall numerical scheme is stable, i.e., will converge even in presence of perturbations

How to start optimizing

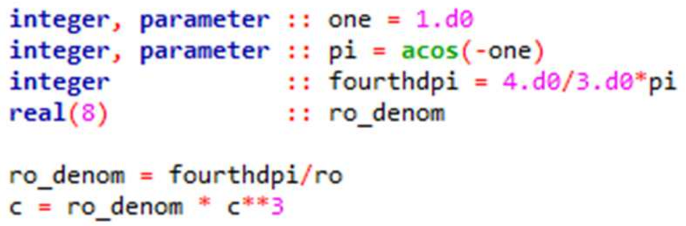
- Manual code operations can significantly speed-up the code
  - E.g., matrix product
- But for simple operations, libraries are available
  - BLAS, LAPACK, etc.
- Modern compilers can handle loop-based optimizations
  - Compiling time is not a problem anymore
  - Better to have simpler-looking code and leave these optimizations to the compiler
- Do not change operations that are already optimized, this will increase the risk of introducing errors
- Use the **profiler** to understand where are the code's bottlenecks

Simple optimization techniques

1. Avoid divisions and complex functions where possible



```
integer, parameter :: pi = 3.141592d0
do i = 1, 100
     c(i) = 4.d0/3.d0*pi/ro*c(i)**3.d0
end do
```



```
integer, parameter :: one = 1.d0
integer, parameter :: pi = acos(-one)
integer            :: fourthdpi = 4.d0/3.d0*pi
real(8)            :: ro_denom

ro_denom = fourthdpi/ro
c = ro_denom * c**3
```

200 divisions
100 powers
200 products
~ 19400 cycles

1 division
400 products
~4820 cycles

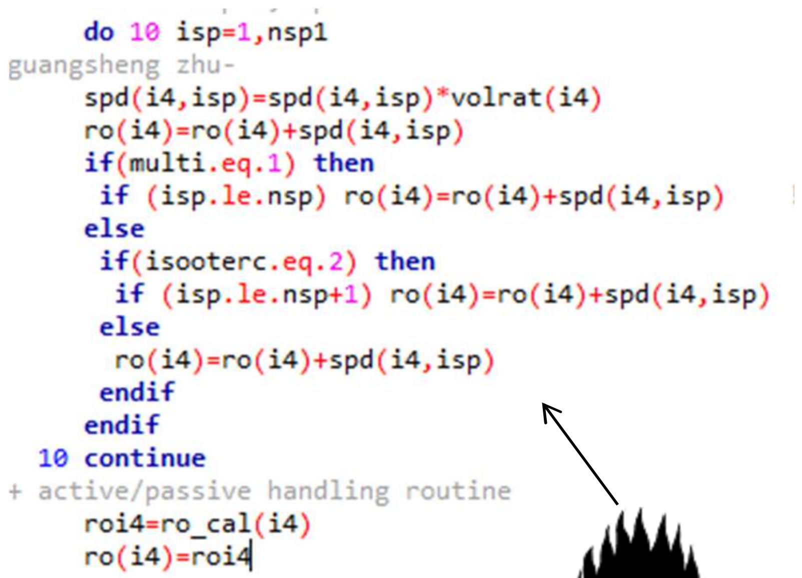
The compiler will optimize part of this

Faster, simpler to read, less prone to errors

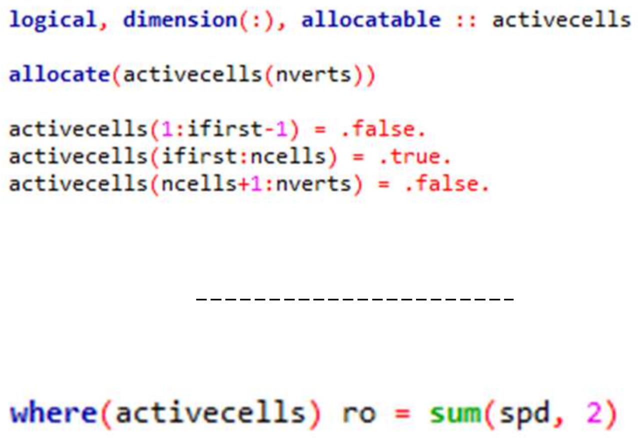
Simple optimization techniques

2. Keep data contiguity during every operation

3. Do not introduce clauses within loops



```
      do 10 isp=1,nsp1
guangsheng zhu-
      spd(i4,isp)=spd(i4,isp)*volrat(i4)
      ro(i4)=ro(i4)+spd(i4,isp)
      if(multi.eq.1) then
       if (isp.le.nsp) ro(i4)=ro(i4)+spd(i4,isp)
      else
       if(isooterc.eq.2) then
        if (isp.le.nsp+1) ro(i4)=ro(i4)+spd(i4,isp)
       else
        ro(i4)=ro(i4)+spd(i4,isp)
       endif
      endif
   10 continue
+ active/passive handling routine
      roi4=ro_cal(i4)
      ro(i4)=roi4
```



```
logical, dimension(:), allocatable :: activecells

allocate(activecells(nverts))

activecells(1:ifirst-1) = .false.
activecells(ifirst:ncells) = .true.
activecells(ncells+1:nverts) = .false.

                      ----------------------

where(activecells) ro = sum(spd, 2)
```

CODING HORROR

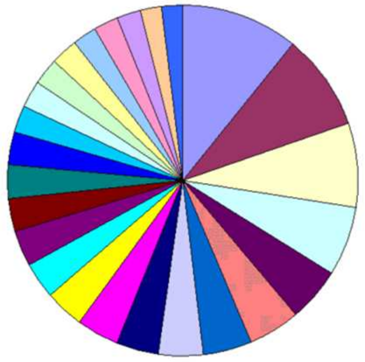
Simple optimization techniques

4. Always start from the profiler (e.g., gprof)

- Will tell what the bottlenecks are at runtime
- Different bottlenecks may arise in different runs

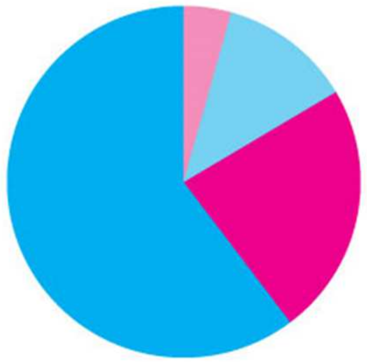
→ A "call graph" will show the CPU time needed by all functions and the ones nested into them

→ Can look at every single line of code



← Well optimized: no dominant sources of CPU time

→ Poorly optimized: few operations dominate CPU time



The KIVA spd array

- It’s the only two-dimensional array in the code

→ tip: **never** use two-dimensional arrays, if not needed!

- It contains species densities in every cell: spd(i4, isp)
- What was the correct ordering? spd(isp, i4) or spd(i4, isp)?

→ When we need to evaluate mixture-averaged properties, we need to access the species dimension

→ when we need to evaluate field properties for a same species, we need to access the cell dimension

➔ KIVA chooses to give more importance to the field properties, as each species is advected separately from each other

➔ At that time, only 10-12 species were typically used

➔ Good for the implicit solver, not for thermodynamics and equation-of-state relationships

Practice
Making KIVA faster